

Draft Standard for Information Technology— Portable Operating System Interface (POSIX®)

Draft Technical Standard: System Interfaces, Issue 7

Prepared by the Austin Group (www.opengroup.org/austin)

Copyright © 2006 The Institute of Electrical & Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Copyright © 2006 The Open Group
Thames Tower, Station Road, Reading, Berkshire RG1 1LX, UK

All rights reserved.

Except as permitted below, no part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior permission of the copyright owners. This is an unapproved draft, subject to change. Permission is hereby granted for Austin Group participants to reproduce this document for purposes of IEEE, The Open Group, and JTC1 standardization activities. Other entities seeking permission to reproduce this document for standardization purposes or other activities must contact the copyright owners for an appropriate license. Use of information contained within this unapproved draft is at your own risk.

Portions of this document are derived with permission from copyrighted material owned by Hewlett-Packard Company, International Business Machines Corporation, Novell Inc., The Open Software Foundation, and Sun Microsystems, Inc.

Abstract

This standard defines a standard operating system interface and environment, including a command interpreter (or “shell”), and common utility programs to support applications portability at the source code level. This standard is intended to be used by both applications developers and system implementors and comprises four major components (each in an associated volume):

- General terms, concepts, and interfaces common to all volumes of this standard, including utility conventions and C-language header definitions, are included in the Base Definitions volume.
- Definitions for system service functions and subroutines, language-specific system services for the C programming language, function issues, including portability, error handling, and error recovery, are included in the System Interfaces volume.
- Definitions for a standard source code-level interface to command interpretation services (a “shell”) and common utility programs for application programs are included in the Shell and Utilities volume.
- Extended rationale that did not fit well into the rest of the document structure, which contains historical information concerning the contents of this standard and why features were included or discarded by the standard developers, is included in the Rationale (Informative) volume.

The following areas are outside the scope of this standard:

- Graphics interfaces
- Database management system interfaces
- Record I/O considerations
- Object or binary code portability
- System configuration and resource availability

This standard describes the external characteristics and facilities that are of importance to applications developers, rather than the internal construction techniques employed to achieve these capabilities. Special emphasis is placed on those functions and facilities that are needed in a wide variety of commercial applications.

Keywords

application program interface (API), argument, asynchronous, basic regular expression (BRE), batch job, batch system, built-in utility, byte, child, command language interpreter, CPU, extended regular expression (ERE), FIFO, file access control mechanism, input/output (I/O), job control, network, portable operating system interface (POSIX[®]), parent, shell, stream, string, synchronous, system, thread, X/Open System Interface (XSI)

Feedback

This standard has been prepared by the Austin Group. Feedback relating to the material contained in this standard may be submitted using the Austin Group web site at www.opengroup.org/austin/bugreport.html.



Contents

1	Chapter 1	Introduction	1
2	1.1	Scope	1
3	1.2	Conformance.....	1
4	1.3	Normative References	1
5	1.4	Change History	1
6	1.5	Terminology	1
7	1.6	Definitions	3
8	1.7	Relationship to Other Formal Standards.....	3
9	1.8	Portability.....	3
10	1.8.1	Codes	3
11	1.9	Format of Entries.....	10
12	Chapter 2	General Information	13
13	2.1	Use and Implementation of Functions.....	13
14	2.2	The Compilation Environment	14
15	2.2.1	POSIX.1 Symbols.....	14
16	2.2.1.1	The <code>_POSIX_C_SOURCE</code> Feature Test Macro.....	14
17	2.2.1.2	The <code>_XOPEN_SOURCE</code> Feature Test Macro	14
18	2.2.2	The Name Space.....	15
19	2.3	Error Numbers.....	21
20	2.3.1	Additional Error Numbers	28
21	2.4	Signal Concepts	28
22	2.4.1	Signal Generation and Delivery.....	28
23	2.4.2	Realtime Signal Generation and Delivery	29
24	2.4.3	Signal Actions	30
25	2.4.4	Signal Effects on Other Functions.....	33
26	2.5	Standard I/O Streams	34
27	2.5.1	Interaction of File Descriptors and Standard I/O Streams	35
28	2.5.2	Stream Orientation and Encoding Rules	36
29	2.6	STREAMS	38
30	2.6.1	Accessing STREAMS	39
31	2.7	XSI Interprocess Communication	39
32	2.7.1	IPC General Description	39
33	2.8	Realtime.....	40
34	2.8.1	Realtime Signals	41
35	2.8.2	Asynchronous I/O.....	41
36	2.8.3	Memory Management	42
37	2.8.3.1	Memory Locking.....	42
38	2.8.3.2	Memory Mapped Files	43
39	2.8.3.3	Memory Protection	43
40	2.8.3.4	Typed Memory Objects	43
41	2.8.4	Process Scheduling.....	44
42	2.8.5	Clocks and Timers.....	48
43	2.9	Threads	50

44	2.9.1	Thread-Safety.....	50
45	2.9.2	Thread IDs.....	51
46	2.9.3	Thread Mutexes.....	51
47	2.9.4	Thread Scheduling.....	52
48	2.9.5	Thread Cancellation.....	54
49	2.9.5.1	Cancelability States.....	54
50	2.9.5.2	Cancellation Points.....	55
51	2.9.5.3	Thread Cancellation Cleanup Handlers.....	57
52	2.9.5.4	Async-Cancel Safety.....	58
53	2.9.6	Thread Read-Write Locks.....	58
54	2.9.7	Thread Interactions with Regular File Operations.....	59
55	2.9.8	Use of Application-Managed Thread Stacks.....	59
56	2.10	Sockets.....	60
57	2.10.1	Address Families.....	60
58	2.10.2	Addressing.....	60
59	2.10.3	Protocols.....	60
60	2.10.4	Routing.....	60
61	2.10.5	Interfaces.....	60
62	2.10.6	Socket Types.....	61
63	2.10.7	Socket I/O Mode.....	62
64	2.10.8	Socket Owner.....	62
65	2.10.9	Socket Queue Limits.....	62
66	2.10.10	Pending Error.....	62
67	2.10.11	Socket Receive Queue.....	62
68	2.10.12	Socket Out-of-Band Data State.....	63
69	2.10.13	Connection Indication Queue.....	63
70	2.10.14	Signals.....	63
71	2.10.15	Asynchronous Errors.....	64
72	2.10.16	Use of Options.....	64
73	2.10.17	Use of Sockets for Local UNIX Connections.....	67
74	2.10.17.1	Headers.....	67
75	2.10.18	Use of Sockets over Internet Protocols.....	67
76	2.10.19	Use of Sockets over Internet Protocols Based on IPv4.....	68
77	2.10.19.1	Headers.....	68
78	2.10.20	Use of Sockets over Internet Protocols Based on IPv6.....	68
79	2.10.20.1	Addressing.....	68
80	2.10.20.2	Compatibility with IPv4.....	69
81	2.10.20.3	Interface Identification.....	70
82	2.10.20.4	Options.....	70
83	2.10.20.5	Headers.....	71
84	2.11	Tracing.....	71
85	2.11.1	Tracing Data Definitions.....	73
86	2.11.1.1	Structures.....	73
87	2.11.1.2	Trace Stream Attributes.....	77
88	2.11.2	Trace Event Type Definitions.....	77
89	2.11.2.1	System Trace Event Type Definitions.....	77
90	2.11.2.2	User Trace Event Type Definitions.....	80
91	2.11.3	Trace Functions.....	81
92	2.12	Data Types.....	82
93	2.12.1	Defined Types.....	82
94	2.12.2	The char Type.....	83
95	2.12.3	Pointer Types.....	83

96	Chapter 3	System Interfaces.....	85
97		<i>FD_CLR()</i>	86
98		<i>_Exit()</i>	87
99		<i>_longjmp()</i>	92
100		<i>_tolower()</i>	94
101		<i>_toupper()</i>	95
102		<i>a64l()</i>	96
103		<i>abort()</i>	98
104		<i>abs()</i>	100
105		<i>accept()</i>	101
106		<i>access()</i>	103
107		<i>acos()</i>	106
108		<i>acosh()</i>	108
109		<i>acosl()</i>	110
110		<i>aio_cancel()</i>	111
111		<i>aio_error()</i>	113
112		<i>aio_fsync()</i>	114
113		<i>aio_read()</i>	116
114		<i>aio_return()</i>	119
115		<i>aio_suspend()</i>	121
116		<i>aio_write()</i>	123
117		<i>alarm()</i>	126
118		<i>alphasort()</i>	128
119		<i>asctime()</i>	130
120		<i>asin()</i>	133
121		<i>asinh()</i>	135
122		<i>asinl()</i>	137
123		<i>assert()</i>	138
124		<i>atan()</i>	139
125		<i>atan2()</i>	141
126		<i>atanf()</i>	143
127		<i>atanh()</i>	144
128		<i>atanl()</i>	146
129		<i>atexit()</i>	147
130		<i>atof()</i>	149
131		<i>atoi()</i>	150
132		<i>atol()</i>	152
133		<i>basename()</i>	153
134		<i>bind()</i>	155
135		<i>bsearch()</i>	157
136		<i>btowc()</i>	160
137		<i>cabs()</i>	161
138		<i>cacos()</i>	162
139		<i>cacosh()</i>	163
140		<i>cacosl()</i>	164
141		<i>calloc()</i>	165
142		<i>carg()</i>	167
143		<i>casin()</i>	168
144		<i>casinh()</i>	169
145		<i>casinl()</i>	170
146		<i>catan()</i>	171
147		<i>catanh()</i>	172

148	<i>catanl()</i>	173
149	<i>catclose()</i>	174
150	<i>catgets()</i>	175
151	<i>catopen()</i>	177
152	<i>cbrt()</i>	179
153	<i>ccos()</i>	180
154	<i>ccosh()</i>	181
155	<i>ccosl()</i>	182
156	<i>ceil()</i>	183
157	<i>cexp()</i>	185
158	<i>cfgetispeed()</i>	186
159	<i>cfgetospeed()</i>	188
160	<i>cfsetispeed()</i>	189
161	<i>cfsetospeed()</i>	190
162	<i>chdir()</i>	191
163	<i>chmod()</i>	193
164	<i>chown()</i>	197
165	<i>cimag()</i>	201
166	<i>clearerr()</i>	202
167	<i>clock()</i>	203
168	<i>clock_getcpuclid()</i>	204
169	<i>clock_getres()</i>	205
170	<i>clock_nanosleep()</i>	208
171	<i>clock_settime()</i>	211
172	<i>clog()</i>	212
173	<i>close()</i>	213
174	<i>closedir()</i>	216
175	<i>closelog()</i>	218
176	<i>confstr()</i>	222
177	<i>conj()</i>	225
178	<i>connect()</i>	226
179	<i>copysign()</i>	229
180	<i>cos()</i>	230
181	<i>cosh()</i>	232
182	<i>cosl()</i>	234
183	<i>cpow()</i>	235
184	<i>cproj()</i>	236
185	<i>creal()</i>	237
186	<i>creat()</i>	238
187	<i>crypt()</i>	240
188	<i>csin()</i>	242
189	<i>csinh()</i>	243
190	<i>csinl()</i>	244
191	<i>csqrt()</i>	245
192	<i>ctan()</i>	246
193	<i>ctanh()</i>	247
194	<i>ctanl()</i>	248
195	<i>ctermid()</i>	249
196	<i>ctime()</i>	251
197	<i>daylight</i>	253
198	<i>dbm_clearerr()</i>	254
199	<i>difftime()</i>	258

Contents

200	<i>dirfd()</i>	259
201	<i>dirname()</i>	261
202	<i>div()</i>	263
203	<i>dlclose()</i>	264
204	<i>dlderror()</i>	266
205	<i>dlopen()</i>	268
206	<i>dlsym()</i>	271
207	<i>dprintf()</i>	273
208	<i>drand48()</i>	274
209	<i>dup()</i>	277
210	<i>duplocale()</i>	279
211	<i>encrypt()</i>	281
212	<i>endgrent()</i>	283
213	<i>endhostent()</i>	285
214	<i>endnetent()</i>	287
215	<i>endprotoent()</i>	289
216	<i>endpwent()</i>	291
217	<i>endservent()</i>	293
218	<i>endutxent()</i>	295
219	<i>environ</i>	298
220	<i>erand48()</i>	299
221	<i>erf()</i>	300
222	<i>erfc()</i>	302
223	<i>erff()</i>	304
224	<i>errno</i>	305
225	<i>exec</i>	307
226	<i>exit()</i>	319
227	<i>exp()</i>	320
228	<i>exp2()</i>	322
229	<i>expm1()</i>	324
230	<i>fabs()</i>	326
231	<i>faccessat()</i>	328
232	<i>fattach()</i>	329
233	<i>fchdir()</i>	332
234	<i>fchmod()</i>	333
235	<i>fchmodat()</i>	335
236	<i>fchown()</i>	336
237	<i>fchownat()</i>	338
238	<i>fclose()</i>	339
239	<i>fcntl()</i>	341
240	<i>fdatasync()</i>	349
241	<i>fdetach()</i>	350
242	<i>fdim()</i>	352
243	<i>fdopen()</i>	354
244	<i>fdopendir()</i>	356
245	<i>feclearexcept()</i>	359
246	<i>fegetenv()</i>	360
247	<i>fegetexceptflag()</i>	361
248	<i>fegetround()</i>	362
249	<i>feholdexcept()</i>	364
250	<i>feof()</i>	365
251	<i>feraiseexcept()</i>	366

252	<i>ferror()</i>	367
253	<i>fesetenv()</i>	368
254	<i>fesetexceptflag()</i>	369
255	<i>fesetround()</i>	370
256	<i>fetestexcept()</i>	371
257	<i>feupdateenv()</i>	373
258	<i>fexecve</i>	375
259	<i>fflush()</i>	376
260	<i>ffs()</i>	379
261	<i>fgetc()</i>	380
262	<i>fgetpos()</i>	382
263	<i>fgets()</i>	384
264	<i>fgetwc()</i>	386
265	<i>fgetws()</i>	388
266	<i>fileno()</i>	390
267	<i>flockfile()</i>	391
268	<i>floor()</i>	393
269	<i>fma()</i>	395
270	<i>fmax()</i>	397
271	<i>fmemopen()</i>	398
272	<i>fmin()</i>	401
273	<i>fmod()</i>	402
274	<i>fntmsg()</i>	404
275	<i>fnmatch()</i>	407
276	<i>fopen()</i>	409
277	<i>fork()</i>	413
278	<i>fpathconf()</i>	418
279	<i>fpclassify()</i>	423
280	<i>fprintf()</i>	424
281	<i>fputc()</i>	436
282	<i>fputs()</i>	438
283	<i>fputwc()</i>	440
284	<i>fputws()</i>	442
285	<i>fread()</i>	443
286	<i>free()</i>	445
287	<i>freeaddrinfo()</i>	446
288	<i>freelocale()</i>	450
289	<i>freopen()</i>	452
290	<i>frexp()</i>	456
291	<i>fscanf()</i>	458
292	<i>fseek()</i>	465
293	<i>fsetpos()</i>	468
294	<i>fstat()</i>	470
295	<i>fstatat()</i>	473
296	<i>fstatofs()</i>	478
297	<i>fsync()</i>	481
298	<i>ftell()</i>	483
299	<i>ftok()</i>	485
300	<i>ftruncate()</i>	487
301	<i>ftrylockfile()</i>	489
302	<i>ftw()</i>	490
303	<i>funlockfile()</i>	493

Contents

304	<i>futimesat()</i>	494
305	<i>fwide()</i>	495
306	<i>fwprintf()</i>	496
307	<i>fwrite()</i>	503
308	<i>fwscanf()</i>	505
309	<i>gai_strerror()</i>	511
310	<i>getaddrinfo()</i>	512
311	<i>getc()</i>	513
312	<i>getc_unlocked()</i>	514
313	<i>getchar()</i>	516
314	<i>getchar_unlocked()</i>	517
315	<i>getcwd()</i>	518
316	<i>getdate()</i>	520
317	<i>getdelim()</i>	525
318	<i>getegid()</i>	527
319	<i>getenv()</i>	528
320	<i>geteuid()</i>	531
321	<i>getgid()</i>	532
322	<i>getgrent()</i>	533
323	<i>getgrgid()</i>	534
324	<i>getgrnam()</i>	537
325	<i>getgroups()</i>	539
326	<i>gethostent()</i>	541
327	<i>gethostid()</i>	542
328	<i>gethostname()</i>	543
329	<i>getitimer()</i>	544
330	<i>getline()</i>	546
331	<i>getlogin()</i>	547
332	<i>getmsg()</i>	550
333	<i>getnameinfo()</i>	554
334	<i>getnetbyaddr()</i>	557
335	<i>getopt()</i>	558
336	<i>getpeername()</i>	563
337	<i>getpgid()</i>	565
338	<i>getpgrp()</i>	566
339	<i>getpid()</i>	567
340	<i>getpmsg()</i>	568
341	<i>getppid()</i>	569
342	<i>getpriority()</i>	570
343	<i>getprotobyname()</i>	573
344	<i>getpwent()</i>	574
345	<i>getpwnam()</i>	575
346	<i>getpwuid()</i>	578
347	<i>getrlimit()</i>	581
348	<i>getrusage()</i>	584
349	<i>gets()</i>	586
350	<i>getserobyname()</i>	588
351	<i>getsid()</i>	589
352	<i>getsockname()</i>	590
353	<i>getsockopt()</i>	591
354	<i>getsubopt()</i>	594
355	<i>gettimeofday()</i>	598

356	<i>getuid()</i>	599
357	<i>getutxent()</i>	600
358	<i>getwc()</i>	601
359	<i>getwchar()</i>	602
360	<i>glob()</i>	603
361	<i>gmtime()</i>	607
362	<i>grantpt()</i>	609
363	<i>hcreate()</i>	610
364	<i>htonl()</i>	613
365	<i>hypot()</i>	614
366	<i>iconv()</i>	616
367	<i>iconv_close()</i>	619
368	<i>iconv_open()</i>	620
369	<i>if_freenameindex()</i>	622
370	<i>if_indextoname()</i>	623
371	<i>if_nameindex()</i>	624
372	<i>if_nametoindex()</i>	625
373	<i>ilogb()</i>	626
374	<i>imaxabs()</i>	628
375	<i>imaxdiv()</i>	629
376	<i>inet_addr()</i>	630
377	<i>inet_ntop()</i>	632
378	<i>initstate()</i>	634
379	<i>insque()</i>	636
380	<i>ioctl()</i>	639
381	<i>isalnum()</i>	650
382	<i>isalpha()</i>	652
383	<i>isascii()</i>	654
384	<i>isastream()</i>	655
385	<i>isatty()</i>	656
386	<i>isblank()</i>	657
387	<i>iscntrl()</i>	658
388	<i>isdigit()</i>	660
389	<i>isfinite()</i>	662
390	<i>isgraph()</i>	663
391	<i>isgreater()</i>	665
392	<i>isgreaterequal()</i>	666
393	<i>isinf()</i>	667
394	<i>isless()</i>	668
395	<i>islessequal()</i>	669
396	<i>islessgreater()</i>	670
397	<i>islower()</i>	671
398	<i>isnan()</i>	673
399	<i>isnormal()</i>	674
400	<i>isprint()</i>	675
401	<i>ispunct()</i>	677
402	<i>isspace()</i>	679
403	<i>isunordered()</i>	681
404	<i>isupper()</i>	682
405	<i>iswalnum()</i>	684
406	<i>iswalpha()</i>	686
407	<i>iswblank()</i>	688

Contents

408	<i>iswcntrl()</i>	690
409	<i>iswctype()</i>	692
410	<i>iswdigit()</i>	694
411	<i>iswgraph()</i>	696
412	<i>iswlower()</i>	698
413	<i>iswprint()</i>	700
414	<i>iswpunct()</i>	702
415	<i>iswspace()</i>	704
416	<i>iswupper()</i>	706
417	<i>iswxdigit()</i>	708
418	<i>isxdigit()</i>	710
419	<i>j0()</i>	712
420	<i>rand48()</i>	714
421	<i>kill()</i>	715
422	<i>killpg()</i>	718
423	<i>l64a()</i>	720
424	<i>labs()</i>	721
425	<i>lchown()</i>	722
426	<i>lcong48()</i>	724
427	<i>ldexp()</i>	725
428	<i>ldiv()</i>	727
429	<i>lfind()</i>	728
430	<i>lgamma()</i>	729
431	<i>link()</i>	731
432	<i>linkat()</i>	735
433	<i>lio_listio()</i>	736
434	<i>listen()</i>	740
435	<i>llabs()</i>	742
436	<i>lldiv()</i>	743
437	<i>llrint()</i>	744
438	<i>llround()</i>	746
439	<i>localeconv()</i>	748
440	<i>localtime()</i>	752
441	<i>lockf()</i>	755
442	<i>log()</i>	758
443	<i>log10()</i>	760
444	<i>log1p()</i>	762
445	<i>log2()</i>	764
446	<i>logb()</i>	766
447	<i>logf()</i>	768
448	<i>longjmp()</i>	769
449	<i>rand48()</i>	771
450	<i>lrint()</i>	772
451	<i>lround()</i>	774
452	<i>lsearch()</i>	776
453	<i>lseek()</i>	778
454	<i>lstat()</i>	780
455	<i>malloc()</i>	781
456	<i>mblen()</i>	783
457	<i>mbrlen()</i>	785
458	<i>mbrtowc()</i>	787
459	<i>mbsinit()</i>	789

460	<i>mbsnrto wcs()</i>	790
461	<i>mbsrtowcs()</i>	791
462	<i>mbstowcs()</i>	793
463	<i>mbtowc()</i>	795
464	<i>memccpy()</i>	797
465	<i>memchr()</i>	798
466	<i>memcmp()</i>	799
467	<i>memcpy()</i>	800
468	<i>memmove()</i>	801
469	<i>memset()</i>	802
470	<i>mkdir()</i>	803
471	<i>mkdirat()</i>	806
472	<i>mkdtemp()</i>	807
473	<i>mkfifo()</i>	809
474	<i>mkfifoat()</i>	812
475	<i>mknod()</i>	813
476	<i>mknodat()</i>	817
477	<i>mkstemp()</i>	818
478	<i>mktime()</i>	819
479	<i>mlock()</i>	821
480	<i>mlockall()</i>	823
481	<i>mmap()</i>	825
482	<i>modf()</i>	833
483	<i>mprotect()</i>	835
484	<i>mq_close()</i>	837
485	<i>mq_getattr()</i>	838
486	<i>mq_notify()</i>	840
487	<i>mq_open()</i>	842
488	<i>mq_receive()</i>	845
489	<i>mq_send()</i>	848
490	<i>mq_setattr()</i>	850
491	<i>mq_timedreceive()</i>	852
492	<i>mq_timedsend()</i>	853
493	<i>mq_unlink()</i>	854
494	<i>rand48()</i>	856
495	<i>msgctl()</i>	857
496	<i>msgget()</i>	859
497	<i>msgrcv()</i>	861
498	<i>msgsnd()</i>	864
499	<i>msync()</i>	867
500	<i>munlock()</i>	870
501	<i>munlockall()</i>	871
502	<i>munmap()</i>	872
503	<i>nan()</i>	874
504	<i>nanosleep()</i>	875
505	<i>nearbyint()</i>	877
506	<i>newlocale()</i>	879
507	<i>nextafter()</i>	882
508	<i>nftw()</i>	884
509	<i>nice()</i>	888
510	<i>nl_langinfo()</i>	890
511	<i>rand48()</i>	892

Contents

512	<i>ntohl()</i>	893
513	<i>open()</i>	894
514	<i>open_memstream()</i>	902
515	<i>open_wmemstream()</i>	904
516	<i>openat()</i>	905
517	<i>opendir()</i>	906
518	<i>openlog()</i>	907
519	<i>optarg</i>	908
520	<i>pathconf()</i>	909
521	<i>pause()</i>	910
522	<i>pclose()</i>	911
523	<i>perror()</i>	913
524	<i>pipe()</i>	915
525	<i>poll()</i>	917
526	<i>popen()</i>	921
527	<i>posix_fadvise()</i>	924
528	<i>posix_fallocate()</i>	926
529	<i>posix_madvise()</i>	928
530	<i>posix_mem_offset()</i>	930
531	<i>posix_memalign()</i>	932
532	<i>posix_openpt()</i>	933
533	<i>posix_spawn()</i>	935
534	<i>posix_spawn_file_actions_addclose()</i>	943
535	<i>posix_spawn_file_actions_adddup2()</i>	946
536	<i>posix_spawn_file_actions_addopen()</i>	948
537	<i>posix_spawn_file_actions_destroy()</i>	949
538	<i>posix_spawnattr_destroy()</i>	950
539	<i>posix_spawnattr_getflags()</i>	952
540	<i>posix_spawnattr_getpgroup()</i>	954
541	<i>posix_spawnattr_getschedparam()</i>	956
542	<i>posix_spawnattr_getschedpolicy()</i>	958
543	<i>posix_spawnattr_getsigdefault()</i>	960
544	<i>posix_spawnattr_getsigmask()</i>	962
545	<i>posix_spawnattr_init()</i>	964
546	<i>posix_spawnattr_setflags()</i>	965
547	<i>posix_spawnattr_setpgroup()</i>	966
548	<i>posix_spawnattr_setschedparam()</i>	967
549	<i>posix_spawnattr_setschedpolicy()</i>	968
550	<i>posix_spawnattr_setsigdefault()</i>	969
551	<i>posix_spawnattr_setsigmask()</i>	970
552	<i>posix_spawnwp()</i>	971
553	<i>posix_trace_attr_destroy()</i>	972
554	<i>posix_trace_attr_getclockres()</i>	974
555	<i>posix_trace_attr_getinherited()</i>	976
556	<i>posix_trace_attr_getlogsize()</i>	979
557	<i>posix_trace_attr_getname()</i>	982
558	<i>posix_trace_attr_getstreamfullpolicy()</i>	983
559	<i>posix_trace_attr_getstreamsize()</i>	984
560	<i>posix_trace_attr_init()</i>	985
561	<i>posix_trace_attr_setinherited()</i>	986
562	<i>posix_trace_attr_setlogsize()</i>	987
563	<i>posix_trace_attr_setname()</i>	988

564	<i>posix_trace_attr_setstreamfullpolicy()</i>	989
565	<i>posix_trace_attr_setstreamsize()</i>	990
566	<i>posix_trace_clear()</i>	991
567	<i>posix_trace_close()</i>	993
568	<i>posix_trace_create()</i>	995
569	<i>posix_trace_event()</i>	999
570	<i>posix_trace_eventid_equal()</i>	1001
571	<i>posix_trace_eventid_open()</i>	1003
572	<i>posix_trace_eventset_add()</i>	1004
573	<i>posix_trace_eventtypelist_getnext_id()</i>	1006
574	<i>posix_trace_flush()</i>	1008
575	<i>posix_trace_get_attr()</i>	1009
576	<i>posix_trace_get_filter()</i>	1011
577	<i>posix_trace_get_status()</i>	1013
578	<i>posix_trace_getnext_event()</i>	1014
579	<i>posix_trace_open()</i>	1017
580	<i>posix_trace_set_filter()</i>	1018
581	<i>posix_trace_shutdown()</i>	1019
582	<i>posix_trace_start()</i>	1020
583	<i>posix_trace_timedgetnext_event()</i>	1022
584	<i>posix_trace_trid_eventid_open()</i>	1023
585	<i>posix_trace_trygetnext_event()</i>	1024
586	<i>posix_typed_mem_get_info()</i>	1025
587	<i>posix_typed_mem_open()</i>	1027
588	<i>pow()</i>	1030
589	<i>pread()</i>	1033
590	<i>printf()</i>	1034
591	<i>pselect()</i>	1035
592	<i>psiginfo()</i>	1040
593	<i>psignal()</i>	1041
594	<i>pthread_atfork()</i>	1042
595	<i>pthread_attr_destroy()</i>	1044
596	<i>pthread_attr_getdetachstate()</i>	1047
597	<i>pthread_attr_getguardsize()</i>	1049
598	<i>pthread_attr_getinheritsched()</i>	1052
599	<i>pthread_attr_getschedparam()</i>	1054
600	<i>pthread_attr_getschedpolicy()</i>	1056
601	<i>pthread_attr_getscope()</i>	1058
602	<i>pthread_attr_getstack()</i>	1060
603	<i>pthread_attr_getstacksize()</i>	1062
604	<i>pthread_attr_init()</i>	1064
605	<i>pthread_attr_setdetachstate()</i>	1065
606	<i>pthread_attr_setguardsize()</i>	1066
607	<i>pthread_attr_setinheritsched()</i>	1067
608	<i>pthread_attr_setschedparam()</i>	1068
609	<i>pthread_attr_setschedpolicy()</i>	1069
610	<i>pthread_attr_setscope()</i>	1070
611	<i>pthread_attr_setstack()</i>	1071
612	<i>pthread_attr_setstacksize()</i>	1072
613	<i>pthread_barrier_destroy()</i>	1073
614	<i>pthread_barrier_wait()</i>	1075
615	<i>pthread_barrierattr_destroy()</i>	1077

Contents

616	<i>pthread_barrierattr_getpshared()</i>	1079
617	<i>pthread_barrierattr_init()</i>	1081
618	<i>pthread_barrierattr_setpshared()</i>	1082
619	<i>pthread_cancel()</i>	1083
620	<i>pthread_cleanup_pop()</i>	1085
621	<i>pthread_cond_broadcast()</i>	1090
622	<i>pthread_cond_destroy()</i>	1093
623	<i>pthread_cond_signal()</i>	1096
624	<i>pthread_cond_timedwait()</i>	1097
625	<i>pthread_condattr_destroy()</i>	1103
626	<i>pthread_condattr_getclock()</i>	1105
627	<i>pthread_condattr_getpshared()</i>	1107
628	<i>pthread_condattr_init()</i>	1109
629	<i>pthread_condattr_setclock()</i>	1110
630	<i>pthread_condattr_setpshared()</i>	1111
631	<i>pthread_create()</i>	1112
632	<i>pthread_detach()</i>	1115
633	<i>pthread_equal()</i>	1117
634	<i>pthread_exit()</i>	1118
635	<i>pthread_getconcurrency()</i>	1120
636	<i>pthread_getcpuclockid()</i>	1122
637	<i>pthread_getschedparam()</i>	1123
638	<i>pthread_getspecific()</i>	1126
639	<i>pthread_join()</i>	1128
640	<i>pthread_key_create()</i>	1131
641	<i>pthread_key_delete()</i>	1134
642	<i>pthread_kill()</i>	1136
643	<i>pthread_mutex_consistent()</i>	1137
644	<i>pthread_mutex_destroy()</i>	1139
645	<i>pthread_mutex_getprioceiling()</i>	1144
646	<i>pthread_mutex_init()</i>	1146
647	<i>pthread_mutex_lock()</i>	1147
648	<i>pthread_mutex_setprioceiling()</i>	1151
649	<i>pthread_mutex_timedlock()</i>	1152
650	<i>pthread_mutex_trylock()</i>	1155
651	<i>pthread_mutexattr_destroy()</i>	1156
652	<i>pthread_mutexattr_getprioceiling()</i>	1161
653	<i>pthread_mutexattr_getprotocol()</i>	1163
654	<i>pthread_mutexattr_getpshared()</i>	1166
655	<i>pthread_mutexattr_getrobust()</i>	1168
656	<i>pthread_mutexattr_gettype()</i>	1170
657	<i>pthread_mutexattr_init()</i>	1172
658	<i>pthread_mutexattr_setprioceiling()</i>	1173
659	<i>pthread_mutexattr_setprotocol()</i>	1174
660	<i>pthread_mutexattr_setpshared()</i>	1175
661	<i>pthread_mutexattr_setrobust()</i>	1176
662	<i>pthread_mutexattr_settype()</i>	1177
663	<i>pthread_once()</i>	1178
664	<i>pthread_rwlock_destroy()</i>	1180
665	<i>pthread_rwlock_rdlock()</i>	1183
666	<i>pthread_rwlock_timedrdlock()</i>	1186
667	<i>pthread_rwlock_timedwrlock()</i>	1188

668	<i>pthread_rwlock_tryrdlock()</i>	1190
669	<i>pthread_rwlock_trywrlock()</i>	1191
670	<i>pthread_rwlock_unlock()</i>	1193
671	<i>pthread_rwlock_wrlock()</i>	1195
672	<i>pthread_rwlockattr_destroy()</i>	1196
673	<i>pthread_rwlockattr_getpshared()</i>	1198
674	<i>pthread_rwlockattr_init()</i>	1200
675	<i>pthread_rwlockattr_setpshared()</i>	1201
676	<i>pthread_self()</i>	1202
677	<i>pthread_setcancelstate()</i>	1203
678	<i>pthread_setconcurrency()</i>	1205
679	<i>pthread_setschedparam()</i>	1206
680	<i>pthread_setschedprio()</i>	1207
681	<i>pthread_setspecific()</i>	1209
682	<i>pthread_sigmask()</i>	1210
683	<i>pthread_spin_destroy()</i>	1214
684	<i>pthread_spin_lock()</i>	1216
685	<i>pthread_spin_unlock()</i>	1218
686	<i>pthread_testcancel()</i>	1219
687	<i>ptsname()</i>	1220
688	<i>putc()</i>	1221
689	<i>putc_unlocked()</i>	1222
690	<i>putchar()</i>	1223
691	<i>putchar_unlocked()</i>	1224
692	<i>putenv()</i>	1225
693	<i>putmsg()</i>	1227
694	<i>puts()</i>	1231
695	<i>pututxline()</i>	1233
696	<i>putwc()</i>	1234
697	<i>putwchar()</i>	1235
698	<i>pwrite()</i>	1236
699	<i>qsort()</i>	1237
700	<i>raise()</i>	1239
701	<i>rand()</i>	1241
702	<i>random()</i>	1244
703	<i>read()</i>	1245
704	<i>readdir()</i>	1252
705	<i>readlink()</i>	1256
706	<i>readlinkat()</i>	1259
707	<i>readv()</i>	1260
708	<i>realloc()</i>	1262
709	<i>realpath()</i>	1264
710	<i>recv()</i>	1266
711	<i>recvfrom()</i>	1268
712	<i>recvmsg()</i>	1271
713	<i>regcomp()</i>	1274
714	<i>remainder()</i>	1281
715	<i>remove()</i>	1283
716	<i>remque()</i>	1285
717	<i>remquo()</i>	1286
718	<i>rename()</i>	1288
719	<i>renameat()</i>	1293

Contents

720	<i>rewind()</i>	1294
721	<i>rewinddir()</i>	1295
722	<i>rint()</i>	1296
723	<i>rmdir()</i>	1298
724	<i>round()</i>	1301
725	<i>scalbn()</i>	1303
726	<i>scandir()</i>	1305
727	<i>scanf()</i>	1306
728	<i>sched_get_priority_max()</i>	1307
729	<i>sched_getparam()</i>	1308
730	<i>sched_getscheduler()</i>	1309
731	<i>sched_rr_get_interval()</i>	1310
732	<i>sched_setparam()</i>	1311
733	<i>sched_setscheduler()</i>	1313
734	<i>sched_yield()</i>	1315
735	<i>seed48()</i>	1316
736	<i>seekdir()</i>	1317
737	<i>select()</i>	1319
738	<i>sem_close()</i>	1320
739	<i>sem_destroy()</i>	1322
740	<i>sem_getvalue()</i>	1324
741	<i>sem_init()</i>	1326
742	<i>sem_open()</i>	1328
743	<i>sem_post()</i>	1331
744	<i>sem_timedwait()</i>	1333
745	<i>sem_trywait()</i>	1335
746	<i>sem_unlink()</i>	1337
747	<i>sem_wait()</i>	1339
748	<i>semctl()</i>	1340
749	<i>semget()</i>	1343
750	<i>semop()</i>	1346
751	<i>send()</i>	1351
752	<i>sendmsg()</i>	1353
753	<i>sendto()</i>	1356
754	<i>setbuf()</i>	1359
755	<i>setegid()</i>	1360
756	<i>setenv()</i>	1361
757	<i>seteuid()</i>	1363
758	<i>setgid()</i>	1364
759	<i>setgrent()</i>	1366
760	<i>sethostent()</i>	1367
761	<i>setitimer()</i>	1368
762	<i>setjmp()</i>	1369
763	<i>setkey()</i>	1371
764	<i>setlocale()</i>	1372
765	<i>setlogmask()</i>	1376
766	<i>setnetent()</i>	1377
767	<i>setpgid()</i>	1378
768	<i>setpgrp()</i>	1380
769	<i>setpriority()</i>	1381
770	<i>setprotoent()</i>	1382
771	<i>setpwent()</i>	1383

772	<i>setregid()</i>	1384
773	<i>setreuid()</i>	1386
774	<i>setrlimit()</i>	1388
775	<i>setseroent()</i>	1389
776	<i>setsid()</i>	1390
777	<i>setsockopt()</i>	1392
778	<i>setstate()</i>	1395
779	<i>setuid()</i>	1396
780	<i>setutxent()</i>	1399
781	<i>setvbuf()</i>	1400
782	<i>shm_open()</i>	1402
783	<i>shm_unlink()</i>	1406
784	<i>shmat()</i>	1408
785	<i>shmctl()</i>	1410
786	<i>shmdt()</i>	1412
787	<i>shmget()</i>	1413
788	<i>shutdown()</i>	1415
789	<i>sigaction()</i>	1417
790	<i>sigaddset()</i>	1424
791	<i>sigaltstack()</i>	1425
792	<i>sigdelset()</i>	1427
793	<i>sigemptyset()</i>	1428
794	<i>sigfillset()</i>	1430
795	<i>sighold()</i>	1431
796	<i>siginterrupt()</i>	1434
797	<i>sigismember()</i>	1436
798	<i>siglongjmp()</i>	1437
799	<i>signal()</i>	1438
800	<i>signbit()</i>	1440
801	<i>sigpause()</i>	1441
802	<i>sigpending()</i>	1442
803	<i>sigprocmask()</i>	1443
804	<i>sigqueue()</i>	1444
805	<i>sigrelse()</i>	1446
806	<i>sigsetjmp()</i>	1447
807	<i>sigsuspend()</i>	1449
808	<i>sigtimedwait()</i>	1451
809	<i>sigwait()</i>	1455
810	<i>sigwaitinfo()</i>	1457
811	<i>sin()</i>	1458
812	<i>sinh()</i>	1460
813	<i>sinl()</i>	1462
814	<i>sleep()</i>	1463
815	<i>snprintf()</i>	1465
816	<i>socketatmark()</i>	1466
817	<i>socket()</i>	1468
818	<i>socketpair()</i>	1470
819	<i>sprintf()</i>	1472
820	<i>sqrt()</i>	1473
821	<i>srand()</i>	1475
822	<i>srand48()</i>	1476
823	<i>srandom()</i>	1477

Contents

824	<i>sscanf()</i>	1478
825	<i>stat()</i>	1479
826	<i>statvfs()</i>	1480
827	<i>stdin</i>	1481
828	<i>strcpy()</i>	1483
829	<i>stpncpy()</i>	1484
830	<i>strcasecmp()</i>	1485
831	<i>strcat()</i>	1487
832	<i>strchr()</i>	1488
833	<i>strcmp()</i>	1489
834	<i>strcoll()</i>	1491
835	<i>strcpy()</i>	1493
836	<i>strcspn()</i>	1496
837	<i>strdup()</i>	1497
838	<i>strerror()</i>	1499
839	<i>strfmon()</i>	1501
840	<i>strftime()</i>	1505
841	<i>strlen()</i>	1511
842	<i>strncasecmp()</i>	1513
843	<i>strncat()</i>	1514
844	<i>strncmp()</i>	1515
845	<i>strncpy()</i>	1516
846	<i>strndup()</i>	1518
847	<i>strnlen()</i>	1519
848	<i>strpbrk()</i>	1520
849	<i>strptime()</i>	1521
850	<i>strrchr()</i>	1526
851	<i>strsignal()</i>	1527
852	<i>strspn()</i>	1528
853	<i>strstr()</i>	1529
854	<i>strtod()</i>	1530
855	<i>strtoimax()</i>	1534
856	<i>strtok()</i>	1535
857	<i>strtol()</i>	1538
858	<i>strtold()</i>	1540
859	<i>strtoll()</i>	1541
860	<i>strtoul()</i>	1542
861	<i>strtoumax()</i>	1545
862	<i>strxfrm()</i>	1546
863	<i>swab()</i>	1548
864	<i>swprintf()</i>	1549
865	<i>swscanf()</i>	1550
866	<i>symlink()</i>	1551
867	<i>symlinkat()</i>	1554
868	<i>sync()</i>	1555
869	<i>sysconf()</i>	1556
870	<i>syslog()</i>	1563
871	<i>system()</i>	1564
872	<i>tan()</i>	1569
873	<i>tanh()</i>	1571
874	<i>tanl()</i>	1573
875	<i>tcdrain()</i>	1574

876	<i>tcflow()</i>	1576
877	<i>tcflush()</i>	1578
878	<i>tcgetattr()</i>	1580
879	<i>tcgetpgrp()</i>	1582
880	<i>tcgetsid()</i>	1584
881	<i>tcsendbreak()</i>	1585
882	<i>tcsetattr()</i>	1587
883	<i>tcsetpgrp()</i>	1590
884	<i>tdelete()</i>	1592
885	<i>telldir()</i>	1596
886	<i>tempnam()</i>	1597
887	<i>tfind()</i>	1599
888	<i>tgamma()</i>	1600
889	<i>time()</i>	1602
890	<i>timer_create()</i>	1604
891	<i>timer_delete()</i>	1607
892	<i>timer_getoverrun()</i>	1608
893	<i>times()</i>	1611
894	<i>timezone()</i>	1613
895	<i>tmpfile()</i>	1614
896	<i>tmpnam()</i>	1616
897	<i>toascii()</i>	1618
898	<i>tolower()</i>	1619
899	<i>toupper()</i>	1621
900	<i>towctrans()</i>	1622
901	<i>towlower()</i>	1624
902	<i>towupper()</i>	1626
903	<i>trunc()</i>	1628
904	<i>truncate()</i>	1629
905	<i>truncf()</i>	1631
906	<i>tsearch()</i>	1632
907	<i>ttyname()</i>	1633
908	<i>twalk()</i>	1635
909	<i>tzset()</i>	1636
910	<i>ulimit()</i>	1638
911	<i>umask()</i>	1640
912	<i>uname()</i>	1642
913	<i>ungetc()</i>	1644
914	<i>ungetwc()</i>	1646
915	<i>unlink()</i>	1648
916	<i>unlinkat()</i>	1653
917	<i>unlockpt()</i>	1654
918	<i>unsetenv()</i>	1655
919	<i>uselocale()</i>	1656
920	<i>utime()</i>	1658
921	<i>utimes()</i>	1660
922	<i>va_arg()</i>	1663
923	<i>vfprintf()</i>	1664
924	<i>vfscanf()</i>	1665
925	<i>vwprintf()</i>	1666
926	<i>vfwscanf()</i>	1667
927	<i>vprintf()</i>	1668

Contents

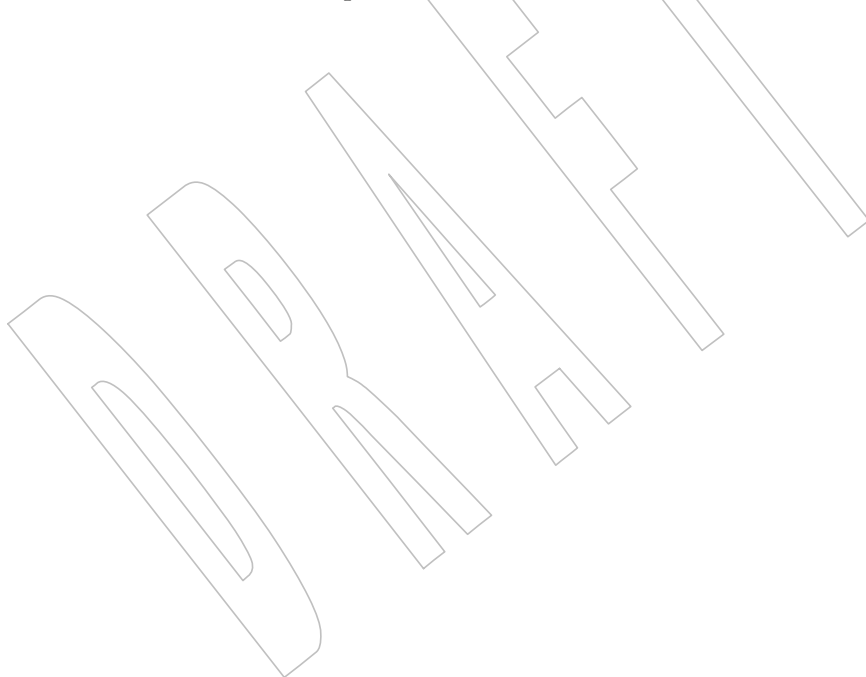
928	<i>vscanf()</i>	1669
929	<i>vsnprintf()</i>	1670
930	<i>vsscanf()</i>	1671
931	<i>vsprintf()</i>	1672
932	<i>vsscanf()</i>	1673
933	<i>vprintf()</i>	1674
934	<i>vscanf()</i>	1675
935	<i>wait()</i>	1676
936	<i>waitid()</i>	1683
937	<i>waitpid()</i>	1685
938	<i>wcpcpy()</i>	1686
939	<i>wcpncpy()</i>	1687
940	<i>wcrtomb()</i>	1688
941	<i>wscasecmp()</i>	1690
942	<i>wscat()</i>	1692
943	<i>wcschr()</i>	1693
944	<i>wscmp()</i>	1694
945	<i>wscoll()</i>	1695
946	<i>wscopy()</i>	1697
947	<i>wscspn()</i>	1698
948	<i>wcsdup()</i>	1699
949	<i>wcsftime()</i>	1700
950	<i>wcslen()</i>	1702
951	<i>wcsncasecmp()</i>	1703
952	<i>wcsncat()</i>	1704
953	<i>wcsncmp()</i>	1705
954	<i>wcsncpy()</i>	1706
955	<i>wcsnlen()</i>	1708
956	<i>wcsnrtombs()</i>	1709
957	<i>wcspbrk()</i>	1710
958	<i>wcsrchr()</i>	1711
959	<i>wcsrtombs()</i>	1712
960	<i>wcsspn()</i>	1714
961	<i>wcsstr()</i>	1715
962	<i>wcstod()</i>	1716
963	<i>wcstoimax()</i>	1720
964	<i>wcstok()</i>	1722
965	<i>wcstol()</i>	1724
966	<i>wcstold()</i>	1727
967	<i>wcstoll()</i>	1728
968	<i>wcstombs()</i>	1729
969	<i>wcstoul()</i>	1731
970	<i>wcstoumax()</i>	1734
971	<i>wcswidth()</i>	1735
972	<i>wcsxfrm()</i>	1736
973	<i>wctob()</i>	1738
974	<i>wctomb()</i>	1739
975	<i>wctrans()</i>	1741
976	<i>wctype()</i>	1743
977	<i>wcwidth()</i>	1745
978	<i>wmemchr()</i>	1746
979	<i>wmemcmp()</i>	1747

980	<i>wmemcpy()</i>	1748
981	<i>wmemmove()</i>	1749
982	<i>wmemset()</i>	1750
983	<i>wordexp()</i>	1751
984	<i>wprintf()</i>	1755
985	<i>write()</i>	1756
986	<i>writew()</i>	1764
987	<i>wscanf()</i>	1766
988	<i>y0()</i>	1767

989	Index	1769
-----	--------------------	-------------

List of Tables

991	2-1	Value of Level for Socket Options	65
992	2-2	Socket-Level Options.....	65
993	2-3	Trace Option: System Trace Events.....	79
994	2-4	Trace and Trace Event Filter Options: System Trace Events	79
995	2-5	Trace and Trace Log Options: System Trace Events.....	80
996	2-6	Trace, Trace Log, and Trace Event Filter Options: System Trace Events.....	80
997	2-7	Trace Option: User Trace Event.....	81



998

Foreword

999

Structure of the Standard

1000

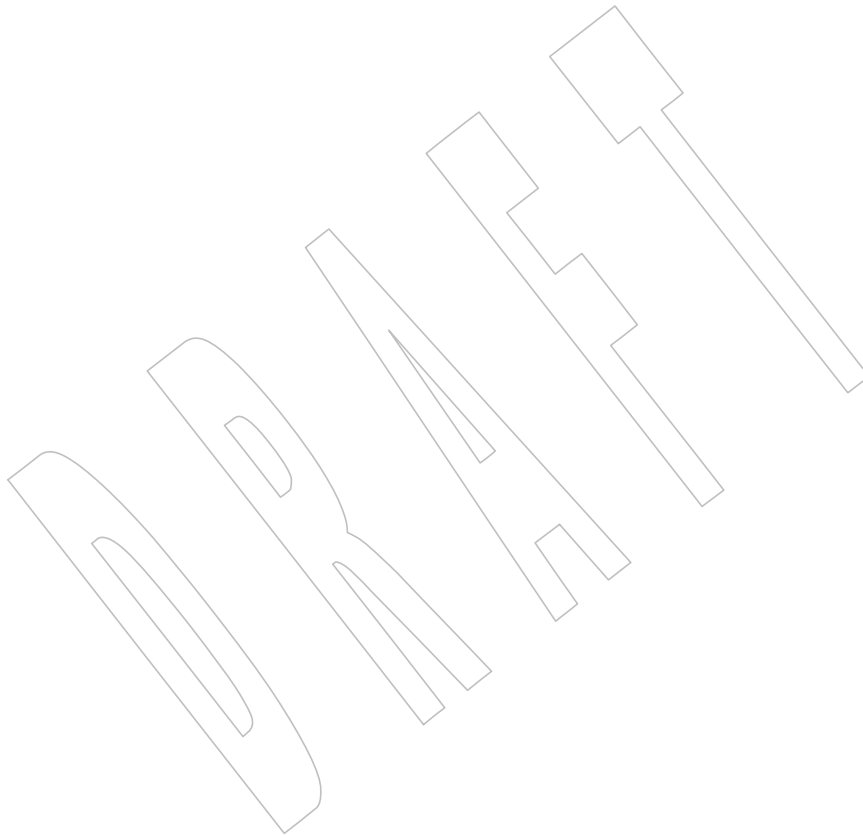
Notes to Reviewers

1001

This section with side shading will not appear in the final copy. - Ed.

1002

This section will be completed in a later draft.



Introduction

1003

1004 **Note:** This introduction is not part of IEEE Std 1003.1-200x, Standard for Information Technology —
1005 Portable Operating System Interface (POSIX).

1006 This draft standard was developed, and is maintained, by a joint working group of members of
1007 the IEEE Portable Applications Standards Committee, members of The Open Group, and
1008 members of ISO/IEC Joint Technical Committee 1. This joint working group is known as the
1009 Austin Group.¹

1010 The Austin Group arose out of discussions amongst the parties which started in early 1998,
1011 leading to an initial meeting and formation of the group in September 1998. The purpose of the
1012 Austin Group is to develop and maintain the core open systems interfaces that are the POSIX[®]
1013 1003.1 (and former 1003.2) standards, ISO/IEC 9945 Parts 1 to 4, and the core of the Single UNIX
1014 Specification.

1015 The approach to specification development has been one of “write once, adopt everywhere”,
1016 with the deliverables being a set of specifications that carry the IEEE POSIX designation, The
1017 Open Group’s Technical Standard designation, and an ISO/IEC designation.

1018 This unique development has combined both the industry-led efforts and the formal
1019 standardization activities into a single initiative, and included a wide spectrum of participants.
1020 The Austin Group continues as the maintenance body for this document.

1021 Anyone wishing to participate in the Austin Group should contact the chair with their request.
1022 There are no fees for participation or membership. You may participate as an observer or as a
1023 contributor. You do not have to attend face-to-face meetings to participate; electronic
1024 participation is most welcome. For more information on the Austin Group and how to
1025 participate, see www.opengroup.org/austin.

1026 Background

1027 The developers of this standard represent a cross section of hardware manufacturers, vendors of
1028 operating systems and other software development tools, software designers, consultants,
1029 academics, authors, applications programmers, and others.

1030 Conceptually, this standard describes a set of fundamental services needed for the efficient
1031 construction of application programs. Access to these services has been provided by defining an
1032 interface, using the C programming language, a command interpreter, and common utility
1033 programs that establish standard semantics and syntax. Since this interface enables application
1034 writers to write portable applications—it was developed with that goal in mind—it has been
1035 designated POSIX,² an acronym for Portable Operating System Interface.

1036 Although originated to refer to the original IEEE Std 1003.1-1988, the name POSIX more
1037 correctly refers to a *family* of related standards: IEEE Std 1003.*n* and the parts of ISO/IEC 9945.
1038 In earlier editions of the IEEE standard, the term POSIX was used as a synonym for
1039 IEEE Std 1003.1-1988. A preferred term, POSIX.1, emerged. This maintained the advantages of
1040 readability of the symbol “POSIX” without being ambiguous with the POSIX family of

-
- 1041 1. The Austin Group is named after the location of the inaugural meeting held at the IBM facility in Austin, Texas in September 1998.
- 1042 2. The name POSIX was suggested by Richard Stallman. It is expected to be pronounced *pahz-icks*, as in *positive*, not *poh-six*, or other
1043 variations. The pronunciation has been published in an attempt to promulgate a standardized way of referring to a standard operating
1044 system interface.

Introduction

1045 standards.

1046 **Audience**

1047 The intended audience for this standard is all persons concerned with an industry-wide
1048 standard operating system based on the UNIX system. This includes at least four groups of
1049 people:

- 1050 1. Persons buying hardware and software systems
- 1051 2. Persons managing companies that are deciding on future corporate computing directions
- 1052 3. Persons implementing operating systems, and especially
- 1053 4. Persons developing applications where portability is an objective

1054 **Purpose**

1055 Several principles guided the development of this standard:

- 1056 • **Application-Oriented**

1057 The basic goal was to promote portability of application programs across UNIX system
1058 environments by developing a clear, consistent, and unambiguous standard for the
1059 interface specification of a portable operating system based on the UNIX system
1060 documentation. This standard codifies the common, existing definition of the UNIX
1061 system.

- 1062 • **Interface, Not Implementation**

1063 This standard defines an interface, not an implementation. No distinction is made between
1064 library functions and system calls; both are referred to as functions. No details of the
1065 implementation of any function are given (although historical practice is sometimes
1066 indicated in the RATIONALE section). Symbolic names are given for constants (such as
1067 signals and error numbers) rather than numbers.

- 1068 • **Source, Not Object, Portability**

1069 This standard has been written so that a program written and translated for execution on
1070 one conforming implementation may also be translated for execution on another
1071 conforming implementation. This standard does not guarantee that executable (object or
1072 binary) code will execute under a different conforming implementation than that for which
1073 it was translated, even if the underlying hardware is identical.

- 1074 • **The C Language**

1075 The system interfaces and header definitions are written in terms of the standard C
1076 language as specified in the ISO C standard.

- 1077 • **No Superuser, No System Administration**

1078 There was no intention to specify all aspects of an operating system. System
1079 administration facilities and functions are excluded from this standard, and functions
1080 usable only by the superuser have not been included. Still, an implementation of the
1081 standard interface may also implement features not in this standard. This standard is also
1082 not concerned with hardware constraints or system maintenance.

- 1083 • **Minimal Interface, Minimally Defined**

1084 In keeping with the historical design principles of the UNIX system, the mandatory core
1085 facilities of this standard have been kept as minimal as possible. Additional capabilities
1086 have been added as optional extensions.

1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128

- Broadly Implementable

The developers of this standard endeavored to make all specified functions implementable across a wide range of existing and potential systems, including:

1. All of the current major systems that are ultimately derived from the original UNIX system code (Version 7 or later)
2. Compatible systems that are not derived from the original UNIX system code
3. Emulations hosted on entirely different operating systems
4. Networked systems
5. Distributed systems
6. Systems running on a broad range of hardware

No direct references to this goal appear in this standard, but some results of it are mentioned in the Rationale (Informative) volume.

- Minimal Changes to Historical Implementations

When the original version—IEEE Std 1003.1-1988—was published, there were no known historical implementations that did not have to change. However, there was a broad consensus on a set of functions, types, definitions, and concepts that formed an interface that was common to most historical implementations.

The adoption of the 1988 and 1990 IEEE system interface standards, the 1992 IEEE shell and utilities standard, the various Open Group (formerly X/Open) specifications, and the 2001 Edition of this standard and its technical corrigenda have consolidated this consensus, and this revision reflects the significantly increased level of consensus arrived at since the original versions. The authors of the original versions tried, as much as possible, to follow the principles below when creating new specifications:

1. By standardizing an interface like one in an historical implementation; for example, directories
2. By specifying an interface that is readily implementable in terms of, and backwards-compatible with, historical implementations, such as the extended *tar* format defined in the *pax* utility
3. By specifying an interface that, when added to an historical implementation, will not conflict with it; for example, the *sigaction()* function

This standard is specifically not a codification of a particular vendor's product.

It should be noted that implementations will have different kinds of extensions. Some will reflect "historical usage" and will be preserved for execution of pre-existing applications. These functions should be considered "obsolescent" and the standard functions used for new applications. Some extensions will represent functions beyond the scope of this standard. These need to be used with careful management to be able to adapt to future extensions of this standard and/or port to implementations that provide these services in a different manner.

- Minimal Changes to Existing Application Code

A goal of this standard was to minimize additional work for the developers of applications. However, because every known historical implementation will have to change at least slightly to conform, some applications will have to change.

*Introduction*1129 **This Standard**

1130 This standard defines the Portable Operating System Interface (POSIX) requirements and
 1131 consists of the following volumes:

- 1132 • Base Definitions
- 1133 • Shell and Utilities
- 1134 • System Interfaces (this volume)
- 1135 • Rationale (Informative)

1136 **This Volume**

1137 The System Interfaces volume describes the interfaces offered to application programs by
 1138 POSIX-conformant systems. Readers are expected to be experienced C language programmers,
 1139 and to be familiar with the Base Definitions volume.

1140 This volume is structured as follows:

- 1141 • [Chapter 1](#) explains the status of this volume and its relationship to other formal standards.
- 1142 • [Chapter 2](#) contains important concepts, terms, and caveats relating to the rest of this
 1143 volume.
- 1144 • [Chapter 3](#) defines the functional interfaces to the POSIX-conformant system.

1145 Comprehensive references are available in the index.

1146 **Typographical Conventions** The following typographical conventions are used throughout this
 1147 standard. In the text, this standard is referred to as IEEE Std 1003.1-200x, which is technically
 1148 identical to The Open Group Base Specifications, Issue 7.

1149 The typographical conventions listed here are for ease of reading only. Editorial inconsistencies
 1150 in the use of typography are unintentional and have no normative meaning in this standard.

1151	Reference	Example	Notes
1152	C-Language Data Structure	aiocb	
1153	C-Language Data Structure Member	<i>aio_lio_opcode</i>	
1154	C-Language Data Type	long	
1155	C-Language External Variable	<i>errno</i>	
1156	C-Language Function	<i>system()</i>	
1157	C-Language Function Argument	<i>arg1</i>	
1158	C-Language Function Family	<i>exec</i>	
1159	C-Language Header	<sys/stat.h>	
1160	C-Language Keyword	return	
1161	C-Language Macro with Argument	<i>assert()</i>	
1162	C-Language Macro with No Argument	INET_ADDRSTRLEN	
1163	C-Language Preprocessing Directive	#define	
1164	Commands within a Utility	a, c	
1165	Conversion Specification, Specifier/Modifier Character	<i>%A, g, E</i>	1
1166	Environment Variable	<i>PATH</i>	
1167	Error Number	[EINTR]	
1168	Example Output	Hello, World	
1169	Filename	/tmp	
1170	Literal Character	<i>'c', '\r', '\'</i>	2
1171	Literal String	<i>"abcde"</i>	2

Reference	Example	Notes
Optional Items in Utility Syntax	[]	
Parameter	<directory pathname>	
Special Character	<newline>	3
Symbolic Constant	_POSIX_VDISABLE	
Symbolic Limit, Configuration Value	{LINE_MAX}	4
Syntax	#include <sys/stat.h>	
User Input and Example Code	echo Hello, World	5
Utility Name	awk	
Utility Operand	file_name	
Utility Option	-c	
Utility Option with Option-Argument	-w width	

Notes:

1. Conversion specifications, specifier characters, and modifier characters are used primarily in date-related functions and utilities and the *fprintf* and *fscanf* formatting functions.
2. Unless otherwise noted, the quotes shall not be used as input or output. When used in a list item, the quotes are omitted. For literal characters, `'\'` (or any of the other sequences such as `''`) is the same as the C constant `'\\'` (or `'\'`).
3. The style selected for some of the special characters, such as `<newline>`, matches the form of the input given to the *localedef* utility. Generally, the characters selected for this special treatment are those that are not visually distinct, such as the control characters `<tab>` or `<newline>`.
4. Names surrounded by braces represent symbolic limits or configuration values which may be declared in appropriate headers by means of the C `#define` construct.
5. Brackets shown in this font, "[]", are part of the syntax and do *not* indicate optional items. In syntax the `'|'` symbol is used to separate alternatives, and ellipses ("`...`") are used to show that additional arguments are optional.

Shading is used to identify extensions and options; see [Section 1.8.1](#) (on page 3).

Footnotes and notes within the body of the normative text are for information only (informative).

Informative sections (such as Rationale, Change History, Application Usage, and so on) are denoted by continuous shading bars in the margins.

Ranges of values are indicated with parentheses or brackets as follows:

- (a,b) means the range of all values from a to b , including neither a nor b .
- $[a,b]$ means the range of all values from a to b , including a and b .
- $[a,b)$ means the range of all values from a to b , including a , but not b .
- $(a,b]$ means the range of all values from a to b , including b , but not a .

Notes:

1. Symbolic limits are used in this volume instead of fixed values for portability. The values of most of these constants are defined in the Base Definitions volume, `<limits.h>` or `<unistd.h>`.
2. The values of errors are defined in the Base Definitions volume, `<errno.h>`.

1215

Participants

1216

1217

1218

IEEE Std 1003.1-200x was prepared by the Austin Group, sponsored by the Portable Applications Standards Committee of the IEEE Computer Society, The Open Group, and ISO/SC22.

1219

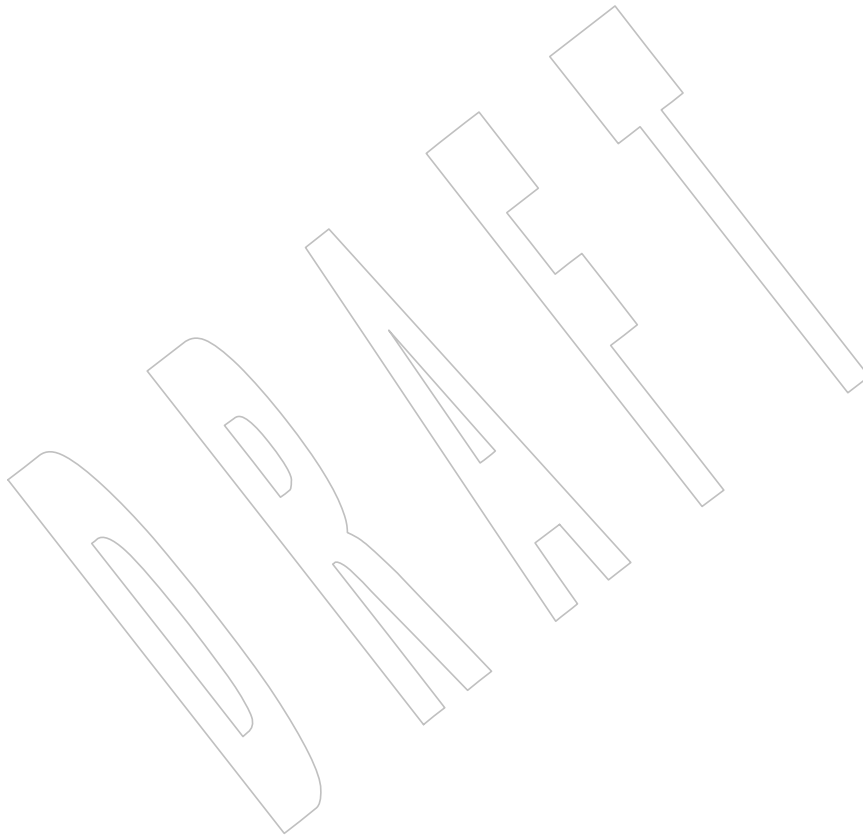
1220

1221

Notes to Reviewers

This section with side shading will not appear in the final copy. - Ed.

This section will be completed once the standard is approved.





Trademarks

1222

1223

1224

1225

1226

The following information is given for the convenience of users of this standard and does not constitute endorsement of these products by The Open Group or the IEEE. There may be other products mentioned in the text that might be covered by trademark protection and readers are advised to verify them independently.

1227

1003.1™ is a trademark of the Institute of Electrical and Electronic Engineers, Inc.

1228

AIX® is a registered trademark of IBM Corporation.

1229

AT&T® is a registered trademark of AT&T in the U.S.A. and other countries.

1230

BSD™ is a trademark of the University of California, Berkeley, U.S.A.

1231

Hewlett-Packard®, HP®, and HP-UX® are registered trademarks of Hewlett-Packard Company.

1232

IBM® is a registered trademark of International Business Machines Corporation.

1233

1234

1235

Boundaryless Information Flow™ and TOGAF™ are trademarks and Motif®, Making Standards Work®, OSF/1®, The Open Group®, UNIX®, and the “X” device are registered trademarks of The Open Group in the United States and other countries.

1236

POSIX® is a registered trademark of the Institute of Electrical and Electronic Engineers, Inc.

1237

Sun® and Sun Microsystems® are registered trademarks of Sun Microsystems, Inc.

1238

1239

/usr/group® is a registered trademark of UniForum, the International Network of UNIX System Users.

1240

Acknowledgements

1241

The contributions of the following organizations to the development of IEEE Std 1003.1-200x are gratefully acknowledged:

1242

1243

- AT&T for permission to reproduce portions of its copyrighted System V Interface Definition (SVID) and material from the UNIX System V Release 2.0 documentation.

1244

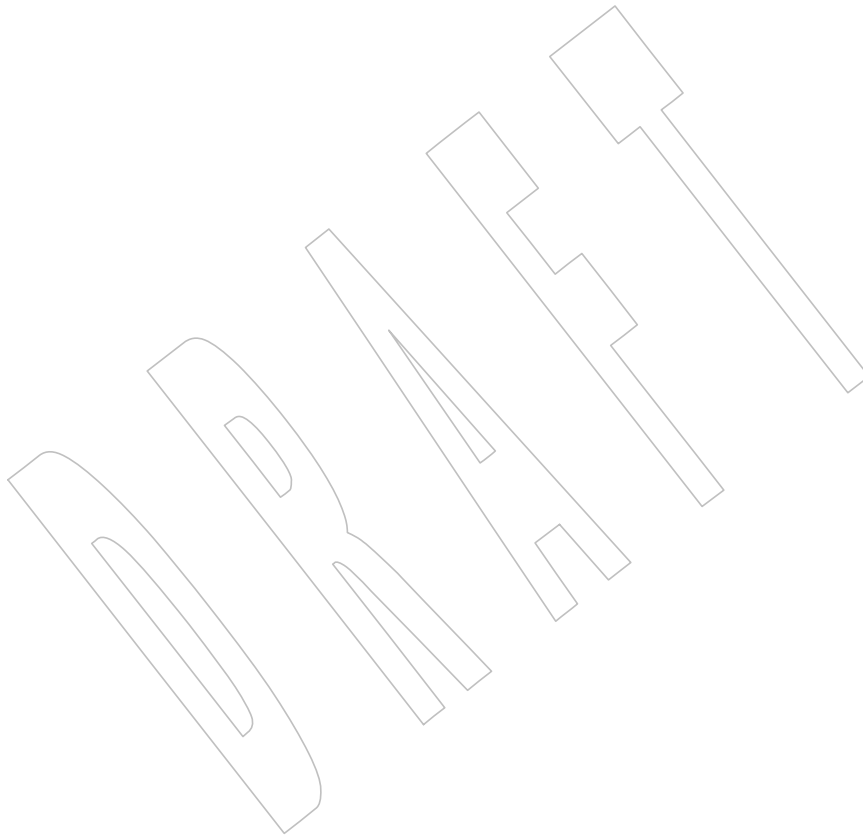
1245

- ISO/IEC JTC 1/SC 22/WG 14 C Language Committee

1246

This standard was prepared by the Austin Group, a joint working group of the IEEE, The Open Group, and ISO/IEC JTC 1/SC 22.

1247



Referenced Documents

1248

1249

Normative References

1250

Normative references for this standard are defined in the Base Definitions volume.

1251

Informative References

1252

The following documents are referenced in this standard:

1253

1984 /usr/group Standard

1254

/usr/group Standards Committee, Santa Clara, CA, UniForum 1984.

1255

Almasi and Gottlieb

1256

George S. Almasi and Allan Gottlieb, *Highly Parallel Computing*, The Benjamin/Cummings Publishing Company, Inc., 1989, ISBN: 0-8053-0177-1.

1257

1258

ANSI C

1259

American National Standard for Information Systems: Standard X3.159-1989, Programming Language C.

1260

1261

ANSI X3.226-1994

1262

American National Standard for Information Systems: Standard X3.226-1994, Programming Language Common LISP.

1263

1264

Brawer

1265

Steven Brawer, *Introduction to Parallel Programming*, Academic Press, 1989, ISBN: 0-12-128470-0.

1266

1267

DeRemer and Pennello Article

1268

DeRemer, Frank and Pennello, Thomas J., *Efficient Computation of LALR(1) Look-Ahead Sets*, SigPlan Notices, Volume 15, No. 8, August 1979.

1269

1270

Draft ANSI X3J11.1

1271

IEEE Floating Point draft report of ANSI X3J11.1 (NCEG).

1272

FIPS 151-1

1273

Federal Information Procurement Standard (FIPS) 151-1. Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language].

1274

1275

FIPS 151-2

1276

Federal Information Procurement Standards (FIPS) 151-2, Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language].

1277

1278

HP-UX Manual

1279

Hewlett-Packard HP-UX Release 9.0 Reference Manual, Third Edition, August 1992.

1280

IEC 60559: 1989

1281

IEC 60559: 1989, Binary Floating-Point Arithmetic for Microprocessor Systems (previously designated IEC 559: 1989).

1282

1283

IEEE Std 754-1985

1284

IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic.

Referenced Documents

- IEEE Std 854-1987
IEEE Std 854-1987, IEEE Standard for Radix-Independent Floating-Point Arithmetic.
- IEEE Std 1003.9-1992
IEEE Std 1003.9-1992, IEEE Standard for Information Technology — POSIX FORTRAN 77 Language Interfaces — Part 1: Binding for System Application Program Interface API.
- IETF RFC 791
Internet Protocol, Version 4 (IPv4), September 1981.
- IETF RFC 819
The Domain Naming Convention for Internet User Applications, Z. Su, J. Postel, August 1982.
- IETF RFC 822
Standard for the Format of ARPA Internet Text Messages, D.H. Crocker, August 1982.
- IETF RFC 919
Broadcasting Internet Datagrams, J. Mogul, October 1984.
- IETF RFC 920
Domain Requirements, J. Postel, J. Reynolds, October 1984.
- IETF RFC 921
Domain Name System Implementation Schedule, J. Postel, October 1984.
- IETF RFC 922
Broadcasting Internet Datagrams in the Presence of Subnets, J. Mogul, October 1984.
- IETF RFC 1034
Domain Names — Concepts and Facilities, P. Mockapetris, November 1987.
- IETF RFC 1035
Domain Names — Implementation and Specification, P. Mockapetris, November 1987.
- IETF RFC 1123
Requirements for Internet Hosts — Application and Support, R. Braden, October 1989.
- IETF RFC 1886
DNS Extensions to Support Internet Protocol, Version 6 (IPv6), C. Huitema, S. Thomson, December 1995.
- IETF RFC 2045
Multipurpose Internet Mail Extensions (MIME), Part 1: Format of Internet Message Bodies, N. Freed, N. Borenstein, November 1996.
- IETF RFC 2181
Clarifications to the DNS Specification, R. Elz, R. Bush, July 1997.
- IETF RFC 2373
Internet Protocol, Version 6 (IPv6) Addressing Architecture, S. Deering, R. Hinden, July 1998.
- IETF RFC 2460
Internet Protocol, Version 6 (IPv6), S. Deering, R. Hinden, December 1998.
- Internationalisation Guide
Guide, July 1993, Internationalisation Guide, Version 2 (ISBN: 1-859120-02-4, G304), published by The Open Group.

- 1327 ISO C (1990)
 1328 ISO/IEC 9899:1990, Programming Languages — C, including Amendment 1:1995 (E), C
 1329 Integrity (Multibyte Support Extensions (MSE) for ISO C).
- 1330 ISO 2375:1985
 1331 ISO 2375:1985, Data Processing — Procedure for Registration of Escape Sequences.
- 1332 ISO 8652:1987
 1333 ISO 8652:1987, Programming Languages — Ada (technically identical to ANSI standard
 1334 1815A-1983).
- 1335 ISO/IEC 1539:1990
 1336 ISO/IEC 1539:1990, Information Technology — Programming Languages — Fortran
 1337 (technically identical to the ANSI X3.9-1978 standard [FORTRAN 77]).
- 1338 ISO/IEC 4873:1991
 1339 ISO/IEC 4873:1991, Information Technology — ISO 8-bit Code for Information Interchange
 1340 — Structure and Rules for Implementation.
- 1341 ISO/IEC 6429:1992
 1342 ISO/IEC 6429:1992, Information Technology — Control Functions for Coded Character
 1343 Sets.
- 1344 ISO/IEC 6937:1994
 1345 ISO/IEC 6937:1994, Information Technology — Coded Character Set for Text
 1346 Communication — Latin Alphabet.
- 1347 ISO/IEC 8802-3:1996
 1348 ISO/IEC 8802-3:1996, Information Technology — Telecommunications and Information
 1349 Exchange Between Systems — Local and Metropolitan Area Networks — Specific
 1350 Requirements — Part 3: Carrier Sense Multiple Access with Collision Detection
 1351 (CSMA/CD) Access Method and Physical Layer Specifications.
- 1352 ISO/IEC 8859
 1353 ISO/IEC 8859, Information Technology — 8-Bit Single-Byte Coded Graphic Character Sets:
 1354 Part 1: Latin Alphabet No. 1
 1355 Part 2: Latin Alphabet No. 2
 1356 Part 3: Latin Alphabet No. 3
 1357 Part 4: Latin Alphabet No. 4
 1358 Part 5: Latin/Cyrillic Alphabet
 1359 Part 6: Latin/Arabic Alphabet
 1360 Part 7: Latin/Greek Alphabet
 1361 Part 8: Latin/Hebrew Alphabet
 1362 Part 9: Latin Alphabet No. 5
 1363 Part 10: Latin Alphabet No. 6
 1364 Part 11: Latin/Thai Alphabet
 1365 Part 13: Latin Alphabet No. 7
 1366 Part 14: Latin Alphabet No. 8
 1367 Part 15: Latin Alphabet No. 9
 1368 Part 16: Latin Alphabet No. 10
- 1369 ISO POSIX-1:1996
 1370 ISO/IEC 9945-1:1996, Information Technology — Portable Operating System Interface
 1371 (POSIX) — Part 1: System Application Program Interface (API) [C Language] (identical to
 1372 ANSI/IEEE Std 1003.1-1996). Incorporating ANSI/IEEE Stds 1003.1-1990, 1003.1b-1993,
 1373 1003.1c-1995, and 1003.1i-1995.

Referenced Documents

- 1374 ISO POSIX-2: 1993
 1375 ISO/IEC 9945-2: 1993, Information Technology — Portable Operating System Interface
 1376 (POSIX) — Part 2: Shell and Utilities (identical to ANSI/IEEE Std 1003.2-1992, as amended
 1377 by ANSI/IEEE Std 1003.2a-1992).
- 1378 Issue 1
 1379 X/Open Portability Guide, July 1985 (ISBN: 0-444-87839-4).
- 1380 Issue 2
 1381 X/Open Portability Guide, January 1987:
- 1382 • Volume 1: XVS Commands and Utilities (ISBN: 0-444-70174-5)
 - 1383 • Volume 2: XVS System Calls and Libraries (ISBN: 0-444-70175-3)
- 1384 Issue 3
 1385 X/Open Specification, 1988, 1989, February 1992:
- 1386 • Commands and Utilities, Issue 3 (ISBN: 1-872630-36-7, C211); this specification was
 1387 formerly X/Open Portability Guide, Issue 3, Volume 1, January 1989, XSI Commands
 1388 and Utilities (ISBN: 0-13-685835-X, XO/XPG/89/002)
 - 1389 • System Interfaces and Headers, Issue 3 (ISBN: 1-872630-37-5, C212); this specification
 1390 was formerly X/Open Portability Guide, Issue 3, Volume 2, January 1989, XSI System
 1391 Interface and Headers (ISBN: 0-13-685843-0, XO/XPG/89/003)
 - 1392 • Curses Interface, Issue 3, contained in Supplementary Definitions, Issue 3
 1393 (ISBN: 1-872630-38-3, C213), Chapters 9 to 14 inclusive; this specification was formerly
 1394 X/Open Portability Guide, Issue 3, Volume 3, January 1989, XSI Supplementary
 1395 Definitions (ISBN: 0-13-685850-3, XO/XPG/89/004)
 - 1396 • Headers Interface, Issue 3, contained in Supplementary Definitions, Issue 3
 1397 (ISBN: 1-872630-38-3, C213), Chapter 19, Cpio and Tar Headers; this specification was
 1398 formerly X/Open Portability Guide Issue 3, Volume 3, January 1989, XSI
 1399 Supplementary Definitions (ISBN: 0-13-685850-3, XO/XPG/89/004)
- 1400 Issue 4
 1401 CAE Specification, July 1992, published by The Open Group:
- 1402 • System Interface Definitions (XBD), Issue 4 (ISBN: 1-872630-46-4, C204)
 - 1403 • Commands and Utilities (XCU), Issue 4 (ISBN: 1-872630-48-0, C203)
 - 1404 • System Interfaces and Headers (XSH), Issue 4 (ISBN: 1-872630-47-2, C202)
- 1405 Issue 4, Version 2
 1406 CAE Specification, August 1994, published by The Open Group:
- 1407 • System Interface Definitions (XBD), Issue 4, Version 2 (ISBN: 1-85912-036-9, C434)
 - 1408 • Commands and Utilities (XCU), Issue 4, Version 2 (ISBN: 1-85912-034-2, C436)
 - 1409 • System Interfaces and Headers (XSH), Issue 4, Version 2 (ISBN: 1-85912-037-7, C435)
- 1410 Issue 5
 1411 Technical Standard, February 1997, published by The Open Group:
- 1412 • System Interface Definitions (XBD), Issue 5 (ISBN: 1-85912-186-1, C605)
 - 1413 • Commands and Utilities (XCU), Issue 5 (ISBN: 1-85912-191-8, C604)

- 1414 • System Interfaces and Headers (XSH), Issue 5 (ISBN: 1-85912-181-0, C606)
- 1415 Issue 6
- 1416 Technical Standard, April 2004, published by The Open Group:
- 1417 • Base Definitions (XBD), Issue 6 (ISBN: 1-931624-43-7, C046)
- 1418 • System Interfaces (XSH), Issue 6 (ISBN: 1-931624-44-5, C047)
- 1419 • Shell and Utilities (XCU), Issue 6 (ISBN: 1-931624-45-3, C048)
- 1420 Knuth Article
- 1421 Knuth, Donald E., *On the Translation of Languages from Left to Right*, Information and Control, Volume 8, No. 6, October 1965.
- 1422
- 1423 KornShell
- 1424 Bolsky, Morris I. and Korn, David G., *The New KornShell Command and Programming Language*, March 1995, Prentice Hall.
- 1425
- 1426 MSE Working Draft
- 1427 Working draft of ISO/IEC 9899:1990/Add3:Draft, Addendum 3 — Multibyte Support
- 1428 Extensions (MSE) as documented in the ISO Working Paper SC22/WG14/N205 dated 31
- 1429 March 1992.
- 1430 POSIX.0: 1995
- 1431 IEEE Std 1003.0-1995, IEEE Guide to the POSIX Open System Environment (OSE) (identical
- 1432 to ISO/IEC TR 14252).
- 1433 POSIX.1: 1988
- 1434 IEEE Std 1003.1-1988, IEEE Standard for Information Technology — Portable Operating
- 1435 System Interface (POSIX) — Part 1: System Application Program Interface (API) [C
- 1436 Language].
- 1437 POSIX.1: 1990
- 1438 IEEE Std 1003.1-1990, IEEE Standard for Information Technology — Portable Operating
- 1439 System Interface (POSIX) — Part 1: System Application Program Interface (API) [C
- 1440 Language].
- 1441 POSIX.1a
- 1442 P1003.1a, Standard for Information Technology — Portable Operating System Interface
- 1443 (POSIX) — Part 1: System Application Program Interface (API) — (C Language)
- 1444 Amendment.
- 1445 POSIX.1d: 1999
- 1446 IEEE Std 1003.1d-1999, IEEE Standard for Information Technology — Portable Operating
- 1447 System Interface (POSIX) — Part 1: System Application Program Interface (API) —
- 1448 Amendment 4: Additional Realtime Extensions [C Language].
- 1449 POSIX.1g: 2000
- 1450 IEEE Std 1003.1g-2000, IEEE Standard for Information Technology — Portable Operating
- 1451 System Interface (POSIX) — Part 1: System Application Program Interface (API) —
- 1452 Amendment 6: Protocol-Independent Interfaces (PII).
- 1453 POSIX.1j: 2000
- 1454 IEEE Std 1003.1j-2000, IEEE Standard for Information Technology — Portable Operating
- 1455 System Interface (POSIX) — Part 1: System Application Program Interface (API) —
- 1456 Amendment 5: Advanced Realtime Extensions [C Language].

Referenced Documents

- 1457 POSIX.1q: 2000
 1458 IEEE Std 1003.1q-2000, IEEE Standard for Information Technology — Portable Operating
 1459 System Interface (POSIX) — Part 1: System Application Program Interface (API) —
 1460 Amendment 7: Tracing [C Language].
- 1461 POSIX.2b
 1462 P1003.2b, Standard for Information Technology — Portable Operating System Interface
 1463 (POSIX) — Part 2: Shell and Utilities — Amendment.
- 1464 POSIX.2d:-1994
 1465 IEEE Std 1003.2d-1994, IEEE Standard for Information Technology — Portable Operating
 1466 System Interface (POSIX) — Part 2: Shell and Utilities — Amendment 1: Batch Environment.
- 1467 POSIX.13:-1998
 1468 IEEE Std 1003.13:1998, IEEE Standard for Information Technology — Standardized
 1469 Application Environment Profile (AEP) — POSIX Realtime Application Support.
- 1470 Sarwate Article
 1471 Sarwate, Dilip V., *Computation of Cyclic Redundancy Checks via Table Lookup*, Communications
 1472 of the ACM, Volume 30, No. 8, August 1988.
- 1473 Sprunt, Sha, and Lehoczky
 1474 Sprunt, B., Sha, L., and Lehoczky, J.P., *Aperiodic Task Scheduling for Hard Real-Time Systems*,
 1475 The Journal of Real-Time Systems, Volume 1, 1989, Pages 27-60.
- 1476 SVID, Issue 1
 1477 American Telephone and Telegraph Company, System V Interface Definition (SVID), Issue
 1478 1; Morristown, NJ, UNIX Press, 1985.
- 1479 SVID, Issue 2
 1480 American Telephone and Telegraph Company, System V Interface Definition (SVID), Issue
 1481 2; Morristown, NJ, UNIX Press, 1986.
- 1482 SVID, Issue 3
 1483 American Telephone and Telegraph Company, System V Interface Definition (SVID), Issue
 1484 3; Morristown, NJ, UNIX Press, 1989.
- 1485 The AWK Programming Language
 1486 Aho, Alfred V., Kernighan, Brian W., and Weinberger, Peter J., *The AWK Programming*
 1487 *Language*, Reading, MA, Addison-Wesley 1988.
- 1488 UNIX Programmer's Manual
 1489 American Telephone and Telegraph Company, *UNIX Time-Sharing System: UNIX*
 1490 *Programmer's Manual*, 7th Edition, Murray Hill, NJ, Bell Telephone Laboratories, January
 1491 1979.
- 1492 XNS, Issue 4
 1493 CAE Specification, August 1994, Networking Services, Issue 4 (ISBN: 1-85912-049-0, C438),
 1494 published by The Open Group.
- 1495 XNS, Issue 5
 1496 CAE Specification, February 1997, Networking Services, Issue 5 (ISBN: 1-85912-165-9, C523),
 1497 published by The Open Group.
- 1498 XNS, Issue 5.2
 1499 Technical Standard, January 2000, Networking Services (XNS), Issue 5.2
 1500 (ISBN: 1-85912-241-8, C808), published by The Open Group.

- 1501 X/Open Curses, Issue 4, Version 2
1502 CAE Specification, May 1996, X/Open Curses, Issue 4, Version 2 (ISBN: 1-85912-171-3,
1503 C610), published by The Open Group.
- 1504 Yacc
1505 *Yacc: Yet Another Compiler Compiler*, Stephen C. Johnson, 1978.

1506 Source Documents

1507 Parts of the following documents were used to create the base documents for this standard:

- 1508 AIX 3.2 Manual
1509 AIX Version 3.2 For RISC System/6000, Technical Reference: Base Operating System and
1510 Extensions, 1990, 1992 (Part No. SC23-2382-00).
- 1511 OSF/1
1512 OSF/1 Programmer's Reference, Release 1.2 (ISBN: 0-13-020579-6).
- 1513 OSF AES
1514 Application Environment Specification (AES) Operating System Programming Interfaces
1515 Volume, Revision A (ISBN: 0-13-043522-8).
- 1516 System V Release 2.0
1517 — UNIX System V Release 2.0 Programmer's Reference Manual (April 1984 - Issue 2).
1518 — UNIX System V Release 2.0 Programming Guide (April 1984 - Issue 2).
- 1519 System V Release 4.2
1520 Operating System API Reference, UNIX[®] SVR4.2 (1992) (ISBN: 0-13-017658-3).

1.1 Scope

The scope of IEEE Std 1003.1-200x is described in the Base Definitions volume of IEEE Std 1003.1-200x.

1.2 Conformance

Conformance requirements for IEEE Std 1003.1-200x are defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance.

1.3 Normative References

Normative references for IEEE Std 1003.1-200x are defined in the Base Definitions volume of IEEE Std 1003.1-200x.

1.4 Change History

Change history is described in the Rationale (Informative) volume of IEEE Std 1003.1-200x, and in the CHANGE HISTORY section of reference pages.

1.5 Terminology

This section appears in the Base Definitions volume of IEEE Std 1003.1-200x, but is repeated here for convenience:

For the purposes of IEEE Std 1003.1-200x, the following terminology definitions apply:

can

Describes a permissible optional feature or behavior available to the user or application. The feature or behavior is mandatory for an implementation that conforms to IEEE Std 1003.1-200x. An application can rely on the existence of the feature or behavior.

implementation-defined

Describes a value or behavior that is not defined by IEEE Std 1003.1-200x but is selected by an implementor. The value or behavior may vary among implementations that conform to IEEE Std 1003.1-200x. An application should not rely on the existence of the value or behavior. An application that relies on such a value or behavior cannot be assured to be portable across conforming implementations.

The implementor shall document such a value or behavior so that it can be used correctly by an application.

29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

legacy

Describes a feature or behavior that is being retained for compatibility with older applications, but which has limitations which make it inappropriate for developing portable applications. New applications should use alternative means of obtaining equivalent functionality.

may

Describes a feature or behavior that is optional for an implementation that conforms to IEEE Std 1003.1-200x. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations.

To avoid ambiguity, the opposite of *may* is expressed as *need not*, instead of *may not*.

shall

For an implementation that conforms to IEEE Std 1003.1-200x, describes a feature or behavior that is mandatory. An application can rely on the existence of the feature or behavior.

For an application or user, describes a behavior that is mandatory.

should

For an implementation that conforms to IEEE Std 1003.1-200x, describes a feature or behavior that is recommended but not mandatory. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations.

For an application, describes a feature or behavior that is recommended programming practice for optimum portability.

undefined

Describes the nature of a value or behavior not defined by IEEE Std 1003.1-200x which results from use of an invalid program construct or invalid data input.

The value or behavior may vary among implementations that conform to IEEE Std 1003.1-200x. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

unspecified

Describes the nature of a value or behavior not specified by IEEE Std 1003.1-200x which results from use of a valid program construct or valid data input.

The value or behavior may vary among implementations that conform to IEEE Std 1003.1-200x. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

1.6 Definitions

Concepts and definitions are defined in the Base Definitions volume of IEEE Std 1003.1-200x.

1.7 Relationship to Other Formal Standards

Great care has been taken to ensure that this volume of IEEE Std 1003.1-200x is fully aligned with the following standards:

ISO C (1999)

ISO/IEC 9899: 1999, Programming Languages — C.

Parts of the ISO/IEC 9899: 1999 standard (hereinafter referred to as the ISO C standard) are referenced to describe requirements also mandated by this volume of IEEE Std 1003.1-200x. Some functions and headers included within this volume of IEEE Std 1003.1-200x have a version in the ISO C standard; in this case CX markings are added as appropriate to show where the ISO C standard has been extended (see [Section 1.8.1](#) (on page 3)). Any conflict between this volume of IEEE Std 1003.1-200x and the ISO C standard is unintentional.

This volume of IEEE Std 1003.1-200x also allows, but does not require, mathematics functions to support IEEE Std 754-1985 and IEEE Std 854-1987.

1.8 Portability

Some of the utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x and functions in the System Interfaces volume of IEEE Std 1003.1-200x describe functionality that might not be fully portable to systems meeting the requirements for POSIX conformance (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance).

Where optional, enhanced, or reduced functionality is specified, the text is shaded and a code in the margin identifies the nature of the option, extension, or warning (see [Section 1.8.1](#) (on page 3)). For maximum portability, an application should avoid such functionality.

1.8.1 Codes

Margin codes and their meanings are listed in the Base Definitions volume of IEEE Std 1003.1-200x, but are repeated here for convenience:

ADV Advisory Information

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the ADV margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the ADV margin legend.

BE Batch Environment Services and Utilities

The functionality described is optional.

Where applicable, utilities are marked with the BE margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the BE margin legend.

CD C-Language Development Utilities

The functionality described is optional.

Where applicable, utilities are marked with the CD margin legend in the SYNOPSIS section.

Where additional semantics apply to a utility, the material is identified by use of the CD margin legend.

CPT

Process CPU-Time Clocks



151		This is a shorthand notation for combinations of multiple option codes.
152		Where applicable, functions are marked with the MC1 margin legend in the SYNOPSIS section.
153		Where additional semantics apply to a function, the material is identified by use of the MC1
154		margin legend.
155		Refer to the Base Definitions volume of IEEE Std 1003.1-200x, Section 1.5.2, Margin Code
156		Notation.
157	ML	Process Memory Locking
158		The functionality described is optional. The functionality described is also an extension to the
159		ISO C standard.
160		Where applicable, functions are marked with the ML margin legend in the SYNOPSIS section.
161		Where additional semantics apply to a function, the material is identified by use of the ML
162		margin legend.
163	MLR	Range Memory Locking
164		The functionality described is optional. The functionality described is also an extension to the
165		ISO C standard.
166		Where applicable, functions are marked with the MLR margin legend in the SYNOPSIS section.
167		Where additional semantics apply to a function, the material is identified by use of the MLR
168		margin legend.
169	MON	Monotonic Clock
170		The functionality described is optional. The functionality described is also an extension to the
171		ISO C standard.
172		Where applicable, functions are marked with the MON margin legend in the SYNOPSIS section.
173		Where additional semantics apply to a function, the material is identified by use of the MON
174		margin legend.
175	MSG	Message Passing
176		The functionality described is optional. The functionality described is also an extension to the
177		ISO C standard.
178		Where applicable, functions are marked with the MSG margin legend in the SYNOPSIS section.
179		Where additional semantics apply to a function, the material is identified by use of the MSG
180		margin legend.
181	MX	IEC 60559 Floating-Point
182		The functionality described is optional. The functionality described is also an extension to the
183		ISO C standard.
184		Where applicable, functions are marked with the MX margin legend in the SYNOPSIS section.
185		Where additional semantics apply to a function, the material is identified by use of the MX
186		margin legend.
187	OB	Obsolescent
188		The functionality described may be withdrawn in a future version of this volume of
189		IEEE Std 1003.1-200x. Strictly Conforming POSIX Applications and Strictly Conforming XSI
190		Applications shall not use obsolescent features.
191		Where applicable, the material is identified by use of the OB margin legend.
192	OF	Output Format Incompletely Specified
193		The functionality described is an XSI extension. The format of the output produced by the
194		utility is not fully specified. It is therefore not possible to post-process this output in a consistent
195		fashion. Typical problems include unknown length of strings and unspecified field delimiters.

196		Where applicable, the material is identified by use of the OF margin legend.
197	OH	Optional Header
198		In the SYNOPSIS section of some interfaces in the System Interfaces volume of
199		IEEE Std 1003.1-200x an included header is marked as in the following example:
200	OH	<code>#include <sys/types.h></code>
201		<code>#include <grp.h></code>
202		<code>struct group *getgrnam(const char *name);</code>
203		The OH margin legend indicates that the marked header is not required on XSI-conformant
204		systems.
205	PIO	Prioritized Input and Output
206		The functionality described is optional. The functionality described is also an extension to the
207		ISO C standard.
208		Where applicable, functions are marked with the PIO margin legend in the SYNOPSIS section.
209		Where additional semantics apply to a function, the material is identified by use of the PIO
210		margin legend.
211	PS	Process Scheduling
212		The functionality described is optional. The functionality described is also an extension to the
213		ISO C standard.
214		Where applicable, functions are marked with the PS margin legend in the SYNOPSIS section.
215		Where additional semantics apply to a function, the material is identified by use of the PS
216		margin legend.
217	RPI	Robust Mutex Priority Inheritance
218		The functionality described is optional. The functionality described is also an extension to the
219		ISO C standard.
220		Where applicable, functions are marked with the RPI margin legend in the SYNOPSIS section.
221		Where additional semantics apply to a function, the material is identified by use of the RPI
222		margin legend.
223	RPP	Robust Mutex Priority Protection
224		The functionality described is optional. The functionality described is also an extension to the
225		ISO C standard.
226		Where applicable, functions are marked with the RPP margin legend in the SYNOPSIS section.
227		Where additional semantics apply to a function, the material is identified by use of the RPP
228		margin legend.
229	RS	Raw Sockets
230		The functionality described is optional. The functionality described is also an extension to the
231		ISO C standard.
232		Where applicable, functions are marked with the RS margin legend in the SYNOPSIS section.
233		Where additional semantics apply to a function, the material is identified by use of the RS
234		margin legend.
235	SD	Software Development Utilities
236		The functionality described is optional.
237		Where applicable, utilities are marked with the SD margin legend in the SYNOPSIS section.
238		Where additional semantics apply to a utility, the material is identified by use of the SD margin
239		legend.

240	SHM	Shared Memory Objects
241		The functionality described is optional. The functionality described is also an extension to the
242		ISO C standard.
243		Where applicable, functions are marked with the SHM margin legend in the SYNOPSIS section.
244		Where additional semantics apply to a function, the material is identified by use of the SHM
245		margin legend.
246	SIO	Synchronized Input and Output
247		The functionality described is optional. The functionality described is also an extension to the
248		ISO C standard.
249		Where applicable, functions are marked with the SIO margin legend in the SYNOPSIS section.
250		Where additional semantics apply to a function, the material is identified by use of the SIO
251		margin legend.
252	SPN	Spawn
253		The functionality described is optional. The functionality described is also an extension to the
254		ISO C standard.
255		Where applicable, functions are marked with the SPN margin legend in the SYNOPSIS section.
256		Where additional semantics apply to a function, the material is identified by use of the SPN
257		margin legend.
258	SS	Process Sporadic Server
259		The functionality described is optional. The functionality described is also an extension to the
260		ISO C standard.
261		Where applicable, functions are marked with the SS margin legend in the SYNOPSIS section.
262		Where additional semantics apply to a function, the material is identified by use of the SS
263		margin legend.
264	TCT	Thread CPU-Time Clocks
265		The functionality described is optional. The functionality described is also an extension to the
266		ISO C standard.
267		Where applicable, functions are marked with the TCT margin legend in the SYNOPSIS section.
268		Where additional semantics apply to a function, the material is identified by use of the TCT
269		margin legend.
270	TEF	Trace Event Filter
271		The functionality described is optional. This functionality is dependent on support for the Trace
272		option. The functionality described is also an extension to the ISO C standard.
273		Where applicable, functions are marked with the TEF margin legend in the SYNOPSIS section.
274		Where additional semantics apply to a function, the material is identified by use of the TEF
275		margin legend.
276	TPI	Non-Robust Mutex Priority Inheritance
277		The functionality described is optional. The functionality described is also an extension to the
278		ISO C standard.
279		Where applicable, functions are marked with the TPI margin legend in the SYNOPSIS section.
280		Where additional semantics apply to a function, the material is identified by use of the TPI
281		margin legend.
282	TPP	Non-Robust Mutex Priority Protection
283		The functionality described is optional. The functionality described is also an extension to the
284		ISO C standard.

285 Where applicable, functions are marked with the TPP margin legend in the SYNOPSIS section.
 286 Where additional semantics apply to a function, the material is identified by use of the TPP
 287 margin legend.

288 TPS **Thread Execution Scheduling**
 289 The functionality described is optional. The functionality described is also an extension to the
 290 ISO C standard.

291 Where applicable, functions are marked with the TPS margin legend for the SYNOPSIS section.
 292 Where additional semantics apply to a function, the material is identified by use of the TPS
 293 margin legend.

294 TRC **Trace**
 295 The functionality described is optional. The functionality described is also an extension to the
 296 ISO C standard.

297 Where applicable, functions are marked with the TRC margin legend in the SYNOPSIS section.
 298 Where additional semantics apply to a function, the material is identified by use of the TRC
 299 margin legend.

300 TRI **Trace Inherit**
 301 The functionality described is optional. This functionality is dependent on support for the Trace
 302 option. The functionality described is also an extension to the ISO C standard.

303 Where applicable, functions are marked with the TRI margin legend in the SYNOPSIS section.
 304 Where additional semantics apply to a function, the material is identified by use of the TRI
 305 margin legend.

306 TRL **Trace Log**
 307 The functionality described is optional. This functionality is dependent on support for the Trace
 308 option. The functionality described is also an extension to the ISO C standard.

309 Where applicable, functions are marked with the TRL margin legend in the SYNOPSIS section.
 310 Where additional semantics apply to a function, the material is identified by use of the TRL
 311 margin legend.

312 TSA **Thread Stack Address Attribute**
 313 The functionality described is optional. The functionality described is also an extension to the
 314 ISO C standard.

315 Where applicable, functions are marked with the TSA margin legend for the SYNOPSIS section.
 316 Where additional semantics apply to a function, the material is identified by use of the TSA
 317 margin legend.

318 TSH **Thread Process-Shared Synchronization**
 319 The functionality described is optional. The functionality described is also an extension to the
 320 ISO C standard.

321 Where applicable, functions are marked with the TSH margin legend in the SYNOPSIS section.
 322 Where additional semantics apply to a function, the material is identified by use of the TSH
 323 margin legend.

324 TSP **Thread Sporadic Server**
 325 The functionality described is optional. The functionality described is also an extension to the
 326 ISO C standard.

327 Where applicable, functions are marked with the TSP margin legend in the SYNOPSIS section.
 328 Where additional semantics apply to a function, the material is identified by use of the TSP
 329 margin legend.

330	TSS	Thread Stack Size Attribute
331		The functionality described is optional. The functionality described is also an extension to the
332		ISO C standard.
333		Where applicable, functions are marked with the TSS margin legend in the SYNOPSIS section.
334		Where additional semantics apply to a function, the material is identified by use of the TSS
335		margin legend.
336	TYM	Typed Memory Objects
337		The functionality described is optional. The functionality described is also an extension to the
338		ISO C standard.
339		Where applicable, functions are marked with the TYM margin legend in the SYNOPSIS section.
340		Where additional semantics apply to a function, the material is identified by use of the TYM
341		margin legend.
342	UP	User Portability Utilities
343		The functionality described is optional.
344		Where applicable, utilities are marked with the UP margin legend in the SYNOPSIS section.
345		Where additional semantics apply to a utility, the material is identified by use of the UP margin
346		legend.
347	UU	UUCP Utilities
348		The functionality described is optional. The functionality described is also an extension to the
349		ISO C standard.
350		Where applicable, functions are marked with the UU margin legend in the SYNOPSIS section.
351		Where additional semantics apply to a function, the material is identified by use of the UU
352		margin legend.
353	XSI	X/Open System Interfaces
354		The functionality described is part of the X/Open Systems Interfaces option. Functionality
355		marked XSI is an extension to the ISO C standard. Application writers may confidently make
356		use of such extensions on all systems supporting the X/Open System Interfaces option.
357		If an entire SYNOPSIS section is shaded and marked XSI, all the functionality described in that
358		reference page is an extension. See the Base Definitions volume of IEEE Std 1003.1-200x, Section
359		3.439, XSI.
360	XSR	XSI STREAMS
361		The functionality described is optional. The functionality described is also an extension to the
362		ISO C standard.
363		Where applicable, functions are marked with the XSR margin legend in the SYNOPSIS section.
364		Where additional semantics apply to a function, the material is identified by use of the XSR
365		margin legend.

1.9 Format of Entries

The entries in [Chapter 3](#) are based on a common format as follows. The only sections relating to conformance are the SYNOPSIS, DESCRIPTION, RETURN VALUE, and ERRORS sections.

NAME

This section gives the name or names of the entry and briefly states its purpose.

SYNOPSIS

This section summarizes the use of the entry being described. If it is necessary to include a header to use this function, the names of such headers are shown, for example:

```
#include <stdio.h>
```

DESCRIPTION

This section describes the functionality of the function or header.

RETURN VALUE

This section indicates the possible return values, if any.

If the implementation can detect errors, “successful completion” means that no error has been detected during execution of the function. If the implementation does detect an error, the error is indicated.

For functions where no errors are defined, “successful completion” means that if the implementation checks for errors, no error has been detected. If the implementation can detect errors, and an error is detected, the indicated return value is returned and *errno* may be set.

ERRORS

This section gives the symbolic names of the error values returned by a function or stored into a variable accessed through the symbol *errno* if an error occurs.

“No errors are defined” means that error values returned by a function or stored into a variable accessed through the symbol *errno*, if any, depend on the implementation.

EXAMPLES

This section is informative.

This section gives examples of usage, where appropriate. In the event of conflict between an example and a normative part of this volume of IEEE Std 1003.1-200x, the normative material is to be taken as correct.

APPLICATION USAGE

This section is informative.

This section gives warnings and advice to application writers about the entry. In the event of conflict between warnings and advice and a normative part of this volume of IEEE Std 1003.1-200x, the normative material is to be taken as correct.

RATIONALE

This section is informative.

This section contains historical information concerning the contents of this volume of IEEE Std 1003.1-200x and why features were included or discarded by the standard developers.

407
408
409
410
411
412
413
414
415
416
417

FUTURE DIRECTIONS

This section is informative.

This section provides comments which should be used as a guide to current thinking; there is not necessarily a commitment to adopt these future directions.

SEE ALSO

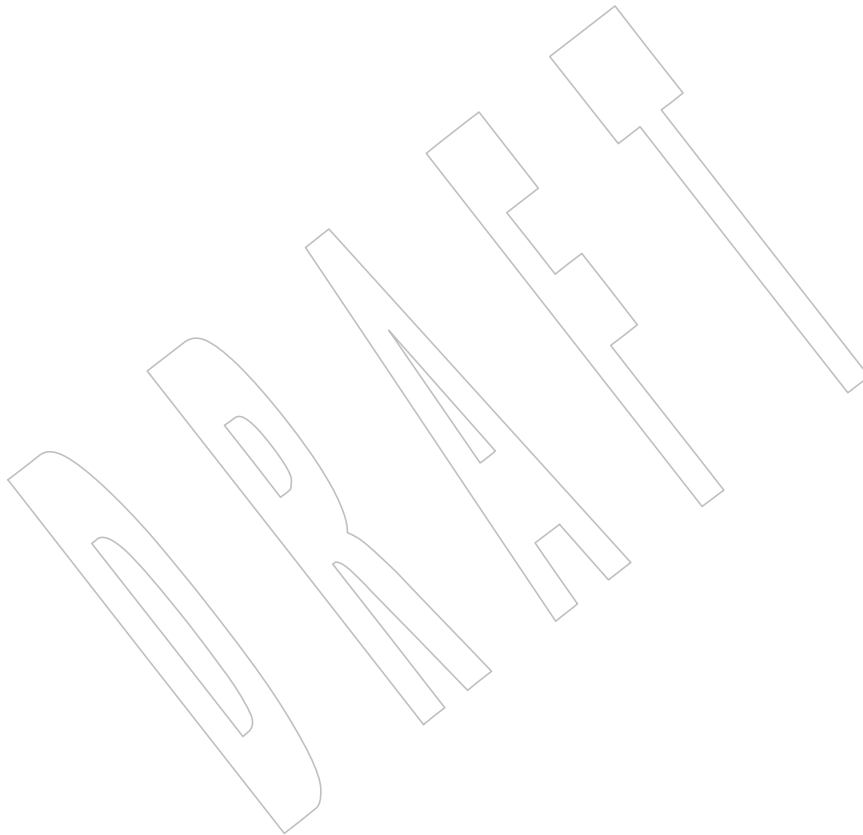
This section is informative.

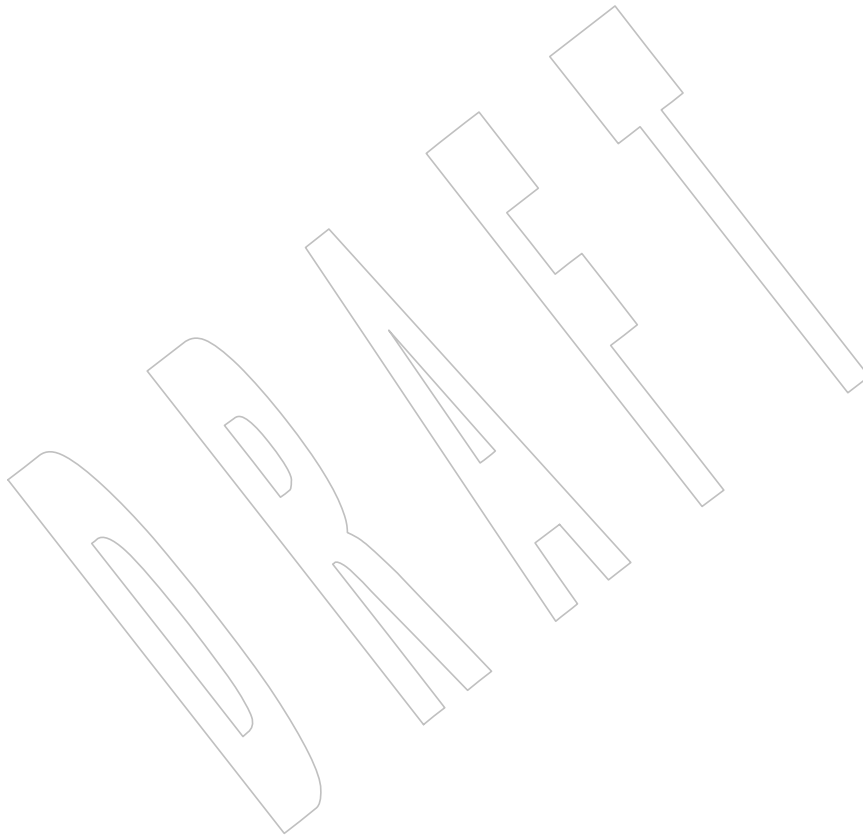
This section gives references to related information.

CHANGE HISTORY

This section is informative.

This section shows the derivation of the entry and any significant changes that have been made to it.





This chapter covers information that is relevant to all the functions specified in Chapter 3 and the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers.

2.1 Use and Implementation of Functions

Each of the following statements shall apply unless explicitly stated otherwise in the detailed descriptions that follow:

1. If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer), the behavior is undefined.
2. Any function declared in a header may also be implemented as a macro defined in the header, so a function should not be declared explicitly if its header is included. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a function even if it is also defined as a macro. The use of the C-language `#undef` construct to remove any such macro definition shall also ensure that an actual function is referred to.
3. Any invocation of a function that is implemented as a macro shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments. Likewise, those function-like macros described in the following sections may be invoked in an expression anywhere a function with a compatible return type could be called.
4. Provided that a function can be declared without reference to any type defined in a header, it is also permissible to declare the function explicitly and use it without including its associated header.
5. If a function that accepts a variable number of arguments is not declared (explicitly or by including its associated header), the behavior is undefined.

446 2.2 The Compilation Environment

447 2.2.1 POSIX.1 Symbols

448 Certain symbols in this volume of IEEE Std 1003.1-200x are defined in headers (see the Base
449 Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers). Some of those headers could
450 also define symbols other than those defined by IEEE Std 1003.1-200x, potentially conflicting
451 with symbols used by the application. Also, IEEE Std 1003.1-200x defines symbols that are not
452 permitted by other standards to appear in those headers without some control on the visibility
453 of those symbols.

454 Symbols called “feature test macros” are used to control the visibility of symbols that might be
455 included in a header. Implementations, future versions of IEEE Std 1003.1-200x, and other
456 standards may define additional feature test macros.

457 In the compilation of an application that **#defines** a feature test macro specified by
458 IEEE Std 1003.1-200x, no header defined by IEEE Std 1003.1-200x shall be included prior to the
459 definition of the feature test macro. This restriction also applies to any implementation-
460 provided header in which these feature test macros are used. If the definition of the macro does
461 not precede the **#include**, the result is undefined.

462 Feature test macros shall begin with the underscore character ('_').

463 2.2.1.1 The `_POSIX_C_SOURCE` Feature Test Macro

464 A POSIX-conforming application should ensure that the feature test macro `_POSIX_C_SOURCE`
465 is defined before inclusion of any header.

466 When an application includes a header described by IEEE Std 1003.1-200x, and when this feature
467 test macro is defined to have the value `200xxxL`:

- 468 1. All symbols required by IEEE Std 1003.1-200x to appear when the header is included shall
469 be made visible.
- 470 2. Symbols that are explicitly permitted, but not required, by IEEE Std 1003.1-200x to appear
471 in that header (including those in reserved name spaces) may be made visible.
- 472 3. Additional symbols not required or explicitly permitted by IEEE Std 1003.1-200x to be in
473 that header shall not be made visible, except when enabled by another feature test macro.

474 Identifiers in IEEE Std 1003.1-200x may only be undefined using the **#undef** directive as
475 described in [Section 2.1](#) or [Section 2.2.2](#) (on page 15). These **#undef** directives shall follow all
476 **#include** directives of any header in IEEE Std 1003.1-200x.

477 **Note:** The POSIX.1-1990 standard specified a macro called `_POSIX_SOURCE`. This has been
478 superseded by `_POSIX_C_SOURCE`.

479 2.2.1.2 The `_XOPEN_SOURCE` Feature Test Macro

480 XSI An XSI-conforming application should ensure that the feature test macro `_XOPEN_SOURCE` is
481 defined with the value 700 before inclusion of any header. This is needed to enable the
482 functionality described in [Section 2.2.1.1](#) and to ensure that the XSI option is enabled.

483 Since this volume of IEEE Std 1003.1-200x is aligned with the ISO C standard, and since all
484 functionality enabled by `_POSIX_C_SOURCE` set equal to `200xxxL` is enabled by
485 `_XOPEN_SOURCE` set equal to 700, there should be no need to define `_POSIX_C_SOURCE` if
486 `_XOPEN_SOURCE` is so defined. Therefore, if `_XOPEN_SOURCE` is set equal to 700 and
487 `_POSIX_C_SOURCE` is set equal to `200xxxL`, the behavior is the same as if only
488 `_XOPEN_SOURCE` is defined and set equal to 700. However, should `_POSIX_C_SOURCE` be set

489 to a value greater than 200xxxL, the behavior is unspecified.

490 If `_XOPEN_SOURCE` is defined with the value 700 and `_POSIX_C_SOURCE` is undefined before
 491 inclusion of any header, then the header may define the `_POSIX_C_SOURCE` macro with the
 492 value 200xxxL.

493 2.2.2 The Name Space

494 All identifiers in this volume of IEEE Std 1003.1-200x, except *environ*, are defined in at least one
 495 of the headers, as shown in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13,
 496 Headers. When `_XOPEN_SOURCE` or `_POSIX_C_SOURCE` is defined, each header defines or
 497 declares some identifiers, potentially conflicting with identifiers used by the application. The set
 498 of identifiers visible to the application consists of precisely those identifiers from the header
 499 pages of the included headers, as well as additional identifiers reserved for the implementation.
 500 In addition, some headers may make visible identifiers from other headers as indicated on the
 501 relevant header pages.

502 Implementations may also add members to a structure or union without controlling the
 503 visibility of those members with a feature test macro, as long as a user-defined macro with the
 504 same name cannot interfere with the correct interpretation of the program. The identifiers
 505 reserved for use by the implementation are described below:

- 506 1. Each identifier with external linkage described in the header section is reserved for use as
 507 an identifier with external linkage if the header is included.
- 508 2. Each macro described in the header section is reserved for any use if the header is
 509 included.
- 510 3. Each identifier with file scope described in the header section is reserved for use as an
 511 identifier with file scope in the same name space if the header is included.

512 The prefixes `posix_`, `POSIX_`, and `_POSIX_` are reserved for use by IEEE Std 1003.1-200x and
 513 other POSIX standards. Implementations may add symbols to the headers shown in the
 514 following table, provided the identifiers for those symbols either:

- 515 1. Begin with the corresponding reserved prefixes in the table, or
- 516 2. Have one of the corresponding complete names in the table, or
- 517 3. End in the string indicated as a reserved suffix in the table and do not use the reserved
 518 prefixes `posix_`, `POSIX_`, or `_POSIX_`, as long as the reserved suffix is in that part of the
 519 name considered significant by the implementation.

520 Symbols that use the reserved prefix `_POSIX_` may be made visible by implementations in any
 521 header defined by IEEE Std 1003.1-200x.

	Header	Prefix	Suffix	Complete Name
522	<stdio.h>	FILE_		
523	<arpa/inet.h>	in_, inet_		
524	<ctype.h>	to[a-z], is[a-z]		
525	<dirent.h>	d_		
526	<errno.h>	E[0-9], E[A-Z]		
527	<fcntl.h>	l_		
528	<fenv.h>	FE_[A-Z]		
529	<glob.h>	gl_		
530	<grp.h>	gr_		
531	<inttypes.h>			int[0-9a-z-]*_t, uint[0-9a-z-]*_t
532	<limits.h>		_MAX, _MIN	
533	<locale.h>	LC_[A-Z]		
534	MSG <mqueue.h>	mq_, MQ_		
535	XSI <ndbm.h>	dbm_		
536	<netdb.h>	ai_, h_, n_, p_, s_		
537	<net/if.h>	if_		
538	<netinet/in.h>	in_, ip_, s_, sin_		
539	IP6 <poll.h>	in6_, s6_, sin6_		
540	XSI <pthread.h>	pd_, ph_, ps_		
541	<pwd.h>	pthread_, PTHREAD_		
542	<regex.h>	pw_		
543	PS <sched.h>	re_, rm_		
544	<semaphore.h>	sched_, SCHED_		
545	<signal.h>	sem_, SEM_		
546	<string.h>	sa_, si_, SL_, SIG[A-Z], SIG_[A-Z], sigev_, SIGEV_, sival_, uc_		
547	XSI <stropts.h>	ss_, sv_		
548	OB XSR <stdint.h>	bi_, ic_, l_, sl_, str_		int[0-9a-z-]*_t, uint[0-9a-z-]*_t
549	<stdlib.h>			
550	<string.h>	str[a-z]		
551	XSI <sys/ipc.h>	str[a-z], mem[a-z], wcs[a-z]		key, pad, seq
552	<sys/mman.h>	ipc_		
553	XSI <sys/msg.h>	shm_, MAP_, MCL_, MS_, PROT_		msg
554	XSI <sys/resource.h>	msg		
555	XSI <sys/select.h>	rlim_, ru_		
556	XSI <sys/sem.h>	fd_, fds_, FD_		sem
557	XSI <sys/shm.h>	sem		
558	<sys/socket.h>	shm		
559	<sys/stat.h>	ss_, sa_, if_, ifc_, ifru_, infu_, ifra_, msg_, cmsg_, l_		
560	<sys/statvfs.h>	st_		
561	XSI <sys/time.h>	f_		
562	<sys/times.h>	fds_, it_, tv_, FD_		
563	XSI <sys/times.h>	tms_		

	Header	Prefix
596		
597	<dlfcn.h>	RTLD_
598	<fcntl.h>	F_, O_, S_, SEEK_
599	XSI <fmtmsg.h>	MM_
600	<fnmatch.h>	FNM_
601	XSI <ftw.h>	FTW
602	<glob.h>	GLOB_
603	<inttypes.h>	PRI[Xa-z], SCN[Xa-z]
604	<math.h>	FP_[A-Z]
605	XSI <ndbm.h>	DBM_
606	<net/if.h>	IF_
607	<netinet/in.h>	IMPLINK_, IN_, INADDR_, IP_, IPPORT_, IPPROTO_, SOCK_
608	IP6	IPV6_, IN6_
609	<netinet/tcp.h>	TCP_
610	<nl_types.h>	NL_
611	<poll.h>	POLL
612	<regex.h>	REG_
613	<signal.h>	BUS_, CLD_, FPE_, ILL_, SA_, SEGV_, SI_, SIG_[0-9a-z_],
614	XSI	SS_, SV_, TRAP_
615	OB XSR	POLL_
616	CX	SEEK_
617	OB XSR	<stdio.h> FLUSH[A-Z], I_, M_, MUXID_R[A-Z], S_, SND[A-Z], STR
618	XSI	<syslog.h> LOG_
619	XSI	<sys/ipc.h> IPC_
620	XSI	<sys/mman.h> PROT_, MAP_, MS_
621	XSI	<sys/msg.h> MSG[A-Z], MSG_[A-Z]
622	XSI	<sys/resource.h> PRIO_, RLIM_, RLIMIT_, RUSAGE_
623	XSI	<sys/sem.h> SEM_
624	XSI	<sys/shm.h> SHM[A-Z], SHM_[A-Z]
625	XSI	<sys/socket.h> AF_, CMSG_, MSG_, PF_, SCM_, SHUT_, SO
626		<sys/stat.h> S_
627		<sys/statvfs.h> ST_
628	XSI	<sys/time.h> FD_, ITIMER_
629	XSI	<sys/uio.h> IOV_
630	XSI	<sys/wait.h> BUS_, CLD_, FPE_, ILL_, POLL_, SEGV_, SI_, TRAP_
631		<termios.h> V, I, O, TC, B[0-9] (See below.)
632		<unistd.h> SEEK_
633		<wordexp.h> WRDE_

634 The following are used to reserve complete names for the <stdint.h> header:

```

635     INT[0-9A-Za-z_]*_MIN
636     INT[0-9A-Za-z_]*_MAX
637     INT[0-9A-Za-z_]*_C
638     UINT[0-9A-Za-z_]*_MIN
639     UINT[0-9A-Za-z_]*_MAX
640     UINT[0-9A-Za-z_]*_C

```

641 **Note:** The notation [0-9] indicates any digit. The notation [A-Z] indicates any uppercase letter in the
642 portable character set. The notation [0-9a-z_] indicates any digit, any lowercase letter in the
643 portable character set, or underscore.

644 XSI The following reserved names are used as exact matches for `<termios.h>`:

645	CBAUD	EXTB	VDSUSP
646	DEFECHO	FLUSHO	VLNEXT
647	ECHOCTL	LOBLK	VREPRINT
648	ECHOKE	PENDIN	VSTATUS
649	ECHOPRT	SWTCH	VWERASE
650	EXTA	VDISCARD	

651 The following identifiers are reserved regardless of the inclusion of headers:

- 652 1. With the exception of identifiers beginning with the prefix `_POSIX_`, all identifiers that
653 begin with an underscore and either an uppercase letter or another underscore are always
654 reserved for any use by the implementation.
- 655 2. All identifiers that begin with an underscore are always reserved for use as identifiers with
656 file scope in both the ordinary identifier and tag name spaces.
- 657 3. All identifiers in the table below are reserved for use as identifiers with external linkage.
658 Some of these identifiers do not appear in this volume of IEEE Std 1003.1-200x, but are
659 reserved for future use by the ISO C standard.

660	<code>_Exit</code>	<code>atof</code>	<code>catanhl</code>	<code>cimag</code>	<code>cosl</code>
661	<code>abort</code>	<code>atoi</code>	<code>catanl</code>	<code>cimagf</code>	<code>cpow</code>
662	<code>abs</code>	<code>atol</code>	<code>cbrt</code>	<code>cimagl</code>	<code>cpowf</code>
663	<code>acos</code>	<code>atoll</code>	<code>cbrtf</code>	<code>clearerr</code>	<code>cpowl</code>
664	<code>acosf</code>	<code>bsearch</code>	<code>cbrtl</code>	<code>clgamma</code>	<code>cproj</code>
665	<code>acosh</code>	<code>cabs</code>	<code>ccos</code>	<code>clgammaf</code>	<code>cprojf</code>
666	<code>acoshf</code>	<code>cabsf</code>	<code>ccosf</code>	<code>clgammal</code>	<code>cprojl</code>
667	<code>acoshl</code>	<code>cabsl</code>	<code>ccosh</code>	<code>clock</code>	<code>creal</code>
668	<code>acosl</code>	<code>cacos</code>	<code>ccoshf</code>	<code>clog</code>	<code>crealf</code>
669	<code>acosl</code>	<code>cacosf</code>	<code>ccoshl</code>	<code>clog10</code>	<code>creall</code>
670	<code>asctime</code>	<code>cacosh</code>	<code>ccosl</code>	<code>clog10f</code>	<code>csin</code>
671	<code>asin</code>	<code>cacoshf</code>	<code>ceil</code>	<code>clog10l</code>	<code>csinf</code>
672	<code>asinf</code>	<code>cacoshl</code>	<code>ceilf</code>	<code>clog1p</code>	<code>csinh</code>
673	<code>asinh</code>	<code>cacosl</code>	<code>ceilf</code>	<code>clog1pf</code>	<code>csinhf</code>
674	<code>asinhf</code>	<code>calloc</code>	<code>ceil</code>	<code>clog1pl</code>	<code>csinhl</code>
675	<code>asinh</code>	<code>carg</code>	<code>ceil</code>	<code>clog2</code>	<code>csinl</code>
676	<code>asinl</code>	<code>cargf</code>	<code>cerf</code>	<code>clog2f</code>	<code>csqrt</code>
677	<code>asinl</code>	<code>cargl</code>	<code>cerfc</code>	<code>clog2l</code>	<code>csqrtf</code>
678	<code>atan</code>	<code>casin</code>	<code>cerfcf</code>	<code>clogf</code>	<code>csqrtl</code>
679	<code>atan2</code>	<code>casinf</code>	<code>cerfcl</code>	<code>clogl</code>	<code>ctan</code>
680	<code>atan2f</code>	<code>casinh</code>	<code>cerff</code>	<code>conj</code>	<code>ctanf</code>
681	<code>atan2l</code>	<code>casinhf</code>	<code>cerfl</code>	<code>conjf</code>	<code>ctanl</code>
682	<code>atanf</code>	<code>casinhl</code>	<code>cexmp1</code>	<code>conjl</code>	<code>ctgamma</code>
683	<code>atanf</code>	<code>casinl</code>	<code>cexmp1f</code>	<code>copysign</code>	<code>ctgammaf</code>
684	<code>atanh</code>	<code>catan</code>	<code>cexmp1l</code>	<code>copysignf</code>	<code>ctgammal</code>
685	<code>atanh</code>	<code>catanf</code>	<code>cexp</code>	<code>copysignl</code>	<code>ltime</code>
686	<code>atanhf</code>	<code>catanh</code>	<code>cexp2</code>	<code>cos</code>	<code>difftime</code>
687	<code>atanhl</code>	<code>catanh</code>	<code>cexp2f</code>	<code>cosf</code>	<code>div</code>
688	<code>atanl</code>	<code>catanhf</code>	<code>cexp2l</code>	<code>cosl</code>	<code>erfcf</code>
689	<code>atanl</code>	<code>catanhf</code>	<code>cexpf</code>	<code>cosh</code>	<code>erfcl</code>
690	<code>atexit</code>	<code>catanhl</code>	<code>cexpl</code>	<code>coshl</code>	<code>erff</code>

691	erfl	fopen	localtime	putchar	swprintf
692	errno	fprintf	log	puts	swscanf
693	exit	fputc	log10	putwc	system
694	exp	fputs	log10f	putwchar	tan
695	exp2	fputwc	log10l	qsort	tanf
696	exp2f	fputws	log1p	raise	tanh
697	exp2l	fread	log1pf	rand	tanhf
698	expf	free	log1pl	realloc	tanhf
699	expl	freopen	log2	remainderf	tanl
700	expm1	frexp	log2f	remainderl	tgamma
701	expm1f	frexpf	log2l	remove	tgammaf
702	expm1l	frexpl	logb	remquo	tgammal
703	fabs	fscanf	logbf	remquof	time
704	fabsf	fseek	logbl	remquol	tmpfile
705	fabsl	fsetpos	logf	rename	tmpnam
706	fclose	ftell	logl	rewind	to[a-z]*
707	fdim	fwide	longjmp	rint	trunc
708	fdimf	fwprintf	lrint	rintf	truncf
709	fdiml	fwrite	lrintf	rintl	truncl
710	feclearexcept	fwscanf	lrintl	round	ungetc
711	fegetenv	getc	lround	roundf	ungetwc
712	fegetexceptflag	getchar	lroundf	roundl	va_end
713	fegetround	getenv	lroundl	scalbln	vfprintf
714	feholdexcept	gets	malloc	scalblnf	vfscanf
715	feof	getwc	mblen	scalblnl	vwprintf
716	feraiseexcept	getwchar	mbrlen	scalbn	vwscanf
717	ferror	gmtime	mbrtowc	scalbnf	vprintf
718	fesetenv	hypotf	mbsinit	scalbnl	vscanf
719	fesetexceptflag	hypotl	mbsrtowcs	scanf	vsprintf
720	fesetround	ilogb	mbstowcs	setbuf	vsscanf
721	fetestexcept	ilogbf	mbtowc	setjmp	vswprintf
722	feupdateenv	ilogbl	mem[a-z]*	setlocale	vswscanf
723	fflush	imaxabs	mktime	setvbuf	vwprintf
724	fgetc	imaxdiv	modf	signal	vwscanf
725	fgetpos	is[a-z]*	modff	sin	wcrtomb
726	fgets	isblank	modfl	sinf	wcs[a-z]*
727	fgetwc	iswblank	nan	sinh	wcstof
728	fgetws	labs	nanf	sinhf	wcstoimax
729	floor	ldexp	nanl	sinhl	wcstold
730	floorf	ldexpf	nearbyint	sinl	wcstoll
731	floorl	ldexpl	nearbyintf	sprintf	wcstoull
732	fma	ldiv	nearbyintl	sqrt	wcstoumax
733	fmaf	ldiv	nextafterf	sqrtf	wctob
734	fmal	lgammaf	nextafterl	sqrtl	wctomb
735	fmax	lgammal	nexttoward	srand	wctrans
736	fmaxf	llabs	nexttowardf	sscanf	wctype
737	fmaxl	llrint	nexttowardl	str[a-z]*	wcwidth
738	fmin	llrintf	perorr	strtof	wmem[a-z]*
739	fminf	llrintl	pow	strtoimax	wprintf
740	fminl	llround	powf	strtold	wscanf
741	fmod	llroundf	powl	strtoll	
742	fmodf	llroundl	printf	strtoull	
743	fmodl	localeconv	putc	strtoumax	

744 **Note:** The notation [a–z] indicates any lowercase letter in the portable character set. The notation ‘*’
745 indicates any combination of digits, letters in the portable character set, or underscore.

746 4. All functions and external identifiers defined in the Base Definitions volume of
747 IEEE Std 1003.1-200x, Chapter 13, Headers are reserved for use as identifiers with external
748 linkage.

749 5. All the identifiers defined in this volume of IEEE Std 1003.1-200x that have external linkage
750 are always reserved for use as identifiers with external linkage.

751 No other identifiers are reserved.

752 Applications shall not declare or define identifiers with the same name as an identifier reserved
753 in the same context. Since macro names are replaced whenever found, independent of scope and
754 name space, macro names matching any of the reserved identifier names shall not be defined by
755 an application if any associated header is included.

756 Except that the effect of each inclusion of `<assert.h>` depends on the definition of `NDEBUG`,
757 headers may be included in any order, and each may be included more than once in a given
758 scope, with no difference in effect from that of being included only once.

759 If used, the application shall ensure that a header is included outside of any external declaration
760 or definition, and it shall be first included before the first reference to any type or macro it
761 defines, or to any function or object it declares. However, if an identifier is declared or defined in
762 more than one header, the second and subsequent associated headers may be included after the
763 initial reference to the identifier. Prior to the inclusion of a header, the application shall not
764 define any macros with names lexically identical to symbols defined by that header.

765 2.3 Error Numbers

766 Most functions can provide an error number. The means by which each function provides its
767 error numbers is specified in its description.

768 Some functions provide the error number in a variable accessed through the symbol `errno`. The
769 symbol `errno`, defined by including the `<errno.h>` header, expands to a modifiable lvalue of type
770 `int`. It is unspecified whether `errno` is a macro or an identifier declared with external linkage. If a
771 macro definition is suppressed in order to access an actual object, or a program defines an
772 identifier with the name `errno`, the behavior is undefined.

773 The value of `errno` should only be examined when it is indicated to be valid by a function’s
774 return value. No function in this volume of IEEE Std 1003.1-200x shall set `errno` to zero. For each
775 thread of a process, the value of `errno` shall not be affected by function calls or assignments to
776 `errno` by other threads.

777 Some functions return an error number directly as the function value. These functions return a
778 value of zero to indicate success.

779 If more than one error occurs in processing a function call, any one of the possible errors may be
780 returned, as the order of detection is undefined.

781 Implementations may support additional errors not included in this list, may generate errors
782 included in this list under circumstances other than those described here, or may contain
783 extensions or limitations that prevent some errors from occurring. The ERRORS section on each
784 reference page specifies whether an error shall be returned, or whether it may be returned.
785 Implementations shall not generate a different error number from the ones described here for
786 error conditions described in this volume of IEEE Std 1003.1-200x, but may generate additional
787 errors unless explicitly disallowed for a particular function.

788 Each implementation shall document, in the conformance document, situations in which each of
 789 the optional conditions defined in IEEE Std 1003.1-200x is detected. The conformance document
 790 may also contain statements that one or more of the optional error conditions are not detected.

791 Certain threads-related functions are not allowed to return an error code of [EINTR]. Where this
 792 applies it is stated in the ERRORS section on the individual function pages.

793 The following symbolic names identify the possible error numbers, in the context of the
 794 functions specifically defined in this volume of IEEE Std 1003.1-200x; these general descriptions
 795 are more precisely defined in the ERRORS sections of the functions that return them. Only these
 796 symbolic names should be used in programs, since the actual value of the error number is
 797 unspecified. All values listed in this section shall be unique integer constant expressions with
 798 type **int** suitable for use in **#if** preprocessing directives, except as noted below. The values for all
 799 these names shall be found in the **<errno.h>** header defined in the Base Definitions volume of
 800 IEEE Std 1003.1-200x. The actual values are unspecified by this volume of IEEE Std 1003.1-200x.

801 [E2BIG]

802 Argument list too long. The sum of the number of bytes used by the new process image's
 803 argument list and environment list is greater than the system-imposed limit of {ARG_MAX}
 804 bytes.

805 or:

806 Lack of space in an output buffer.

807 or:

808 Argument is greater than the system-imposed maximum.

809 [EACCES]

810 Permission denied. An attempt was made to access a file in a way forbidden by its file
 811 access permissions.

812 [EADDRINUSE]

813 Address in use. The specified address is in use.

814 [EADDRNOTAVAIL]

815 Address not available. The specified address is not available from the local system.

816 [EAFNOSUPPORT]

817 Address family not supported. The implementation does not support the specified address
 818 family, or the specified address is not a valid address for the address family of the specified
 819 socket.

820 [EAGAIN]

821 Resource temporarily unavailable. This is a temporary condition and later calls to the same
 822 routine may complete normally.

823 [EALREADY]

824 Connection already in progress. A connection request is already in progress for the specified
 825 socket.

826 [EBADF]

827 Bad file descriptor. A file descriptor argument is out of range, refers to no open file, or a
 828 read (write) request is made to a file that is only open for writing (reading).

829 [EBADMSG]

830 OB XSR Bad message. During a *read()*, *getmsg()*, *getpmsg()*, or *ioctl()* I_RECVFD request to a
 831 STREAMS device, a message arrived at the head of the STREAM that is inappropriate for
 832 the function receiving the message.

833	<code>read()</code>	Message waiting to be read on a STREAM is not a data message.
834	<code>getmsg()</code> or <code>getpmsg()</code>	A file descriptor was received instead of a control message.
835		
836	<code>ioctl()</code>	Control or data information was received instead of a file descriptor when
837		<code>I_RECVFD</code> was specified.
838	or:	
839	Bad Message. The implementation has detected a corrupted message.	
840	[EBUSY]	
841	Resource busy. An attempt was made to make use of a system resource that is not currently	
842	available, as it is being used by another process in a manner that would have conflicted	
843	with the request being made by this process.	
844	[ECANCELED]	
845	Operation canceled. The associated asynchronous operation was canceled before	
846	completion.	
847	[ECHILD]	
848	No child process. A <code>wait()</code> or <code>waitpid()</code> function was executed by a process that had no	
849	existing or unwaited-for child process.	
850	[ECONNABORTED]	
851	Connection aborted. The connection has been aborted.	
852	[ECONNREFUSED]	
853	Connection refused. An attempt to connect to a socket was refused because there was no	
854	process listening or because the queue of connection requests was full and the underlying	
855	protocol does not support retransmissions.	
856	[ECONNRESET]	
857	Connection reset. The connection was forcibly closed by the peer.	
858	[EDEADLK]	
859	Resource deadlock would occur. An attempt was made to lock a system resource that would	
860	have resulted in a deadlock situation.	
861	[EDESTADDRREQ]	
862	Destination address required. No bind address was established.	
863	[EDOM]	
864	Domain error. An input argument is outside the defined domain of the mathematical	
865	function (defined in the ISO C standard).	
866	[EDQUOT]	
867	Reserved.	
868	[EEXIST]	
869	File exists. An existing file was mentioned in an inappropriate context; for example, as a	
870	new link name in the <code>link()</code> function.	
871	[EFAULT]	
872	Bad address. The system detected an invalid address in attempting to use an argument of a	
873	call. The reliable detection of this error cannot be guaranteed, and when not detected may	
874	result in the generation of a signal, indicating an address violation, which is sent to the	
875	process.	

- 876 [EFBIG]
877 File too large. The size of a file would exceed the maximum file size of an implementation
878 or offset maximum established in the corresponding file description.
- 879 [EHOSTUNREACH]
880 Host is unreachable. The destination host cannot be reached (probably because the host is
881 down or a remote router cannot reach it).
- 882 [EIDRM]
883 Identifier removed. Returned during XSI interprocess communication if an identifier has
884 been removed from the system.
- 885 [EILSEQ]
886 Illegal byte sequence. A wide-character code has been detected that does not correspond to
887 a valid character, or a byte sequence does not form a valid wide-character code (defined in
888 the ISO C standard).
- 889 [EINPROGRESS]
890 Operation in progress. This code is used to indicate that an asynchronous operation has not
891 yet completed.
- 892 or:
893 O_NONBLOCK is set for the socket file descriptor and the connection cannot be
894 immediately established.
- 895 [EINTR]
896 Interrupted function call. An asynchronous signal was caught by the process during the
897 execution of an interruptible function. If the signal handler performs a normal return, the
898 interrupted function call may return this condition (see the Base Definitions volume of
899 IEEE Std 1003.1-200x, <signal.h>).
- 900 [EINVAL]
901 Invalid argument. Some invalid argument was supplied; for example, specifying an
902 undefined signal in a *signal()* function or a *kill()* function.
- 903 [EIO]
904 Input/output error. Some physical input or output error has occurred. This error may be
905 reported on a subsequent operation on the same file descriptor. Any other error-causing
906 operation on the same file descriptor may cause the [EIO] error indication to be lost.
- 907 [EISCONN]
908 Socket is connected. The specified socket is already connected.
- 909 [EISDIR]
910 Is a directory. An attempt was made to open a directory with write mode specified.
- 911 [ELOOP]
912 Symbolic link loop. A loop exists in symbolic links encountered during pathname
913 resolution. This error may also be returned if more than {SYMLOOP_MAX} symbolic links
914 are encountered during pathname resolution.
- 915 [EMFILE]
916 File descriptor value too large. An attempt was made to open a file descriptor with a value
917 XSI greater than or equal to {OPEN_MAX}, or greater than or equal to the soft limit
918 RLIMIT_NOFILE for the process (if smaller than {OPEN_MAX}).
- 919 [EMLINK]
920 Too many links. An attempt was made to have the link count of a single file exceed
921 {LINK_MAX}.

922	[EMSGSIZE]
923	Message too large. A message sent on a transport provider was larger than an internal
924	message buffer or some other network limit.
925	or:
926	Inappropriate message buffer length.
927	[EMULTIHOP]
928	Reserved.
929	[ENAMETOOLONG]
930	Filename too long. The length of a pathname exceeds {PATH_MAX}, or a pathname
931	component is longer than {NAME_MAX}. This error may also occur when pathname
932	substitution, as a result of encountering a symbolic link during pathname resolution, results
933	in a pathname string the size of which exceeds {PATH_MAX}.
934	[ENETDOWN]
935	Network is down. The local network interface used to reach the destination is down.
936	[ENETRESET]
937	The connection was aborted by the network.
938	[ENETUNREACH]
939	Network unreachable. No route to the network is present.
940	[ENFILE]
941	Too many files open in system. Too many files are currently open in the system. The system
942	has reached its predefined limit for simultaneously open files and temporarily cannot
943	accept requests to open another one.
944	[ENOBUFS]
945	No buffer space available. Insufficient buffer resources were available in the system to
946	perform the socket operation.
947	OB XSR [ENODATA]
948	No message available. No message is available on the STREAM head read queue.
949	[ENODEV]
950	No such device. An attempt was made to apply an inappropriate function to a device; for
951	example, trying to read a write-only device such as a printer.
952	[ENOENT]
953	No such file or directory. A component of a specified pathname does not exist, or the
954	pathname is an empty string.
955	[ENOEXEC]
956	Executable file format error. A request is made to execute a file that, although it has the
957	appropriate permissions, is not in the format required by the implementation for executable
958	files.
959	[ENOLCK]
960	No locks available. A system-imposed limit on the number of simultaneous file and record
961	locks has been reached and no more are currently available.
962	[ENOLINK]
963	Reserved.
964	[ENOMEM]
965	Not enough space. The new process image requires more memory than is allowed by the
966	hardware or system-imposed memory management constraints.

967		[ENOMSG]
968		No message of the desired type. The message queue does not contain a message of the
969		required type during XSI interprocess communication.
970		[ENOPROTOOPT]
971		Protocol not available. The protocol option specified to <i>setsockopt()</i> is not supported by the
972		implementation.
973		[ENOSPC]
974		No space left on a device. During the <i>write()</i> function on a regular file or when extending a
975		directory, there is no free space left on the device.
976	OB XSR	[ENOSR]
977		No STREAM resources. Insufficient STREAMS memory resources are available to perform a
978		STREAMS-related function. This is a temporary condition; it may be recovered from if other
979		processes release resources.
980	OB XSR	[ENOSTR]
981		Not a STREAM. A STREAM function was attempted on a file descriptor that was not
982		associated with a STREAMS device.
983		[ENOSYS]
984		Function not implemented. An attempt was made to use a function that is not available in
985		this implementation.
986		[ENOTCONN]
987		Socket not connected. The socket is not connected.
988		[ENOTDIR]
989		Not a directory. A component of the specified pathname exists, but it is not a directory,
990		when a directory was expected.
991		[ENOTEMPTY]
992		Directory not empty. A directory other than an empty directory was supplied when an
993		empty directory was expected.
994		[ENOTSOCK]
995		Not a socket. The file descriptor does not refer to a socket.
996		[ENOTSUP]
997		Not supported. The implementation does not support this feature of the Realtime Option
998		Group.
999		[ENOTTY]
1000		Inappropriate I/O control operation. A control function has been attempted for a file or
1001		special file for which the operation is inappropriate.
1002		[ENXIO]
1003		No such device or address. Input or output on a special file refers to a device that does not
1004		exist, or makes a request beyond the capabilities of the device. It may also occur when, for
1005		example, a tape drive is not on-line.
1006		[EOPNOTSUPP]
1007		Operation not supported on socket. The type of socket (address family or protocol) does not
1008		support the requested operation. A conforming implementation may assign the same values
1009		for [EOPNOTSUPP] and [ENOTSUP].
1010		[EOVERFLOW]
1011		Value too large to be stored in data type. An operation was attempted which would
1012		generate a value that is outside the range of values that can be represented in the relevant

1013		data type or that are allowed for a given data item.
1014		[EPERM]
1015		Operation not permitted. An attempt was made to perform an operation limited to
1016		processes with appropriate privileges or to the owner of a file or other resource.
1017		[EPIPE]
1018		Broken pipe. A write was attempted on a socket, pipe, or FIFO for which there is no process
1019		to read the data.
1020		[EPROTO]
1021		Protocol error. Some protocol error occurred. This error is device-specific, but is generally
1022		not related to a hardware failure.
1023		[EPROTONOSUPPORT]
1024		Protocol not supported. The protocol is not supported by the address family, or the protocol
1025		is not supported by the implementation.
1026		[EPROTOTYPE]
1027		Protocol wrong type for socket. The socket type is not supported by the protocol.
1028		[ERANGE]
1029		Result too large or too small. The result of the function is too large (overflow) or too small
1030		(underflow) to be represented in the available space (defined in the ISO C standard).
1031		[EROFS]
1032		Read-only file system. An attempt was made to modify a file or directory on a file system
1033		that is read-only.
1034		[ESPIPE]
1035		Invalid seek. An attempt was made to access the file offset associated with a pipe or FIFO.
1036		[ESRCH]
1037		No such process. No process can be found corresponding to that specified by the given
1038		process ID.
1039		[ESTALE]
1040		Reserved.
1041	OB XSR	[ETIME]
1042		STREAM <i>ioctl()</i> timeout. The timer set for a STREAMS <i>ioctl()</i> call has expired. The cause of
1043		this error is device-specific and could indicate either a hardware or software failure, or a
1044		timeout value that is too short for the specific operation. The status of the <i>ioctl()</i> operation is
1045		unspecified.
1046		[ETIMEDOUT]
1047		Connection timed out. The connection to a remote machine has timed out. If the connection
1048		timed out during execution of the function that reported this error (as opposed to timing
1049		out prior to the function being called), it is unspecified whether the function has completed
1050		some or all of the documented behavior associated with a successful completion of the
1051		function.
1052		or:
1053		Operation timed out. The time limit associated with the operation was exceeded before the
1054		operation completed.
1055		[ETXTBSY]
1056		Text file busy. An attempt was made to execute a pure-procedure program that is currently
1057		open for writing, or an attempt has been made to open for writing a pure-procedure

1058 program that is being executed.

1059 [EWOULDBLOCK]

1060 Operation would block. An operation on a socket marked as non-blocking has encountered
1061 a situation such as no data available that otherwise would have caused the function to
1062 suspend execution.

1063 A conforming implementation may assign the same values for [EWOULDBLOCK] and
1064 [EAGAIN].

1065 [EXDEV]

1066 Improper link. A link to a file on another file system was attempted.

1067 2.3.1 Additional Error Numbers

1068 Additional implementation-defined error numbers may be defined in `<errno.h>`.

1069 2.4 Signal Concepts

1070 2.4.1 Signal Generation and Delivery

1071 A signal is said to be “generated” for (or sent to) a process or thread when the event that causes
1072 the signal first occurs. Examples of such events include detection of hardware faults, timer
1073 expiration, signals generated via the **sigevent** structure and terminal activity, as well as
1074 invocations of the `kill()` and `sigqueue()` functions. In some circumstances, the same event
1075 generates signals for multiple processes.

1076 At the time of generation, a determination shall be made whether the signal has been generated
1077 for the process or for a specific thread within the process. Signals which are generated by some
1078 action attributable to a particular thread, such as a hardware fault, shall be generated for the
1079 thread that caused the signal to be generated. Signals that are generated in association with a
1080 process ID or process group ID or an asynchronous event, such as terminal activity, shall be
1081 generated for the process.

1082 Each process has an action to be taken in response to each signal defined by the system (see
1083 [Section 2.4.3](#) (on page 30)). A signal is said to be “delivered” to a process when the appropriate
1084 action for the process and signal is taken. A signal is said to be “accepted” by a process when the
1085 signal is selected and returned by one of the `sigwait()` functions.

1086 During the time between the generation of a signal and its delivery or acceptance, the signal is
1087 said to be “pending”. Ordinarily, this interval cannot be detected by an application. However, a
1088 signal can be “blocked” from delivery to a thread. If the action associated with a blocked signal
1089 is anything other than to ignore the signal, and if that signal is generated for the thread, the
1090 signal shall remain pending until it is unblocked, it is accepted when it is selected and returned
1091 by a call to the `sigwait()` function, or the action associated with it is set to ignore the signal.
1092 Signals generated for the process shall be delivered to exactly one of those threads within the
1093 process which is in a call to a `sigwait()` function selecting that signal or has not blocked delivery
1094 of the signal. If there are no threads in a call to a `sigwait()` function selecting that signal, and if all
1095 threads within the process block delivery of the signal, the signal shall remain pending on the
1096 process until a thread calls a `sigwait()` function selecting that signal, a thread unblocks delivery
1097 of the signal, or the action associated with the signal is set to ignore the signal. If the action
1098 associated with a blocked signal is to ignore the signal and if that signal is generated for the
1099 process, it is unspecified whether the signal is discarded immediately upon generation or
1100 remains pending.

1101 Each thread has a “signal mask” that defines the set of signals currently blocked from delivery
 1102 to it. The signal mask for a thread shall be initialized from that of its parent or creating thread,
 1103 or from the corresponding thread in the parent process if the thread was created as the result of a
 1104 call to *fork()*. The *pthread_sigmask()*, *sigaction()*, *sigprocmask()*, and *sigsuspend()* functions control
 1105 the manipulation of the signal mask.

1106 The determination of which action is taken in response to a signal is made at the time the signal
 1107 is delivered, allowing for any changes since the time of generation. This determination is
 1108 independent of the means by which the signal was originally generated. If a subsequent
 1109 occurrence of a pending signal is generated, it is implementation-defined as to whether the
 1110 signal is delivered or accepted more than once in circumstances other than those in which
 1111 queuing is required. The order in which multiple, simultaneously pending signals outside the
 1112 range SIGRTMIN to SIGRTMAX are delivered to or accepted by a process is unspecified.

1113 When any stop signal (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) is generated for a process, any
 1114 pending SIGCONT signals for that process shall be discarded. Conversely, when SIGCONT is
 1115 generated for a process, all pending stop signals for that process shall be discarded. When
 1116 SIGCONT is generated for a process that is stopped, the process shall be continued, even if the
 1117 SIGCONT signal is blocked or ignored. If SIGCONT is blocked and not ignored, it shall remain
 1118 pending until it is either unblocked or a stop signal is generated for the process.

1119 An implementation shall document any condition not specified by this volume of
 1120 IEEE Std 1003.1-200x under which the implementation generates signals.

1121 2.4.2 Realtime Signal Generation and Delivery

1122 This section describes functionality to support realtime signal generation and delivery.

1123 Some signal-generating functions, such as high-resolution timer expiration, asynchronous I/O
 1124 completion, interprocess message arrival, and the *sigqueue()* function, support the specification
 1125 of an application-defined value, either explicitly as a parameter to the function or in a **sigevent**
 1126 structure parameter. The **sigevent** structure is defined in **<signal.h>** and contains at least the
 1127 following members:

Member Type	Member Name	Description
int	<i>sigev_notify</i>	Notification type.
int	<i>sigev_signo</i>	Signal number.
union sigval	<i>sigev_value</i>	Signal value.
void*(union sigval)	<i>sigev_notify_function</i>	Notification function.
(pthread_attr_t*)	<i>sigev_notify_attributes</i>	Notification attributes.

1128 The *sigev_notify* member specifies the notification mechanism to use when an asynchronous
 1129 event occurs. This volume of IEEE Std 1003.1-200x defines the following values for the
 1130 *sigev_notify* member:

1131 SIGEV_NONE No asynchronous notification shall be delivered when the event of
 1132 interest occurs.

1133 SIGEV_SIGNAL The signal specified in *sigev_signo* shall be generated for the process when
 1134 the event of interest occurs. If the implementation supports the Realtime
 1135 Signals Extension option and if the SA_SIGINFO flag is set for that signal
 1136 number, then the signal shall be queued to the process and the value
 1137 specified in *sigev_value* shall be the *si_value* component of the generated
 1138 signal. If SA_SIGINFO is not set for that signal number, it is unspecified
 1139 whether the signal is queued and what value, if any, is sent.

1146 SIGEV_THREAD A notification function shall be called to perform notification.

1147 An implementation may define additional notification mechanisms.

1148 The *sigev_signo* member specifies the signal to be generated. The *sigev_value* member is the
1149 application-defined value to be passed to the signal-catching function at the time of the signal
1150 delivery or to be returned at signal acceptance as the *si_value* member of the **siginfo_t** structure.

1151 The **signal** union is defined in **<signal.h>** and contains at least the following members:

Member Type	Member Name	Description
int	<i>sival_int</i>	Integer signal value.
void*	<i>sival_ptr</i>	Pointer signal value.

1155 The *sival_int* member shall be used when the application-defined value is of type **int**; the
1156 *sival_ptr* member shall be used when the application-defined value is a pointer.

1157 When a signal is generated by the *sigqueue()* function or any signal-generating function that
1158 supports the specification of an application-defined value, the signal shall be marked pending
1159 and, if the SA_SIGINFO flag is set for that signal, the signal shall be queued to the process along
1160 with the application-specified signal value. Multiple occurrences of signals so generated are
1161 queued in FIFO order. It is unspecified whether signals so generated are queued when the
1162 SA_SIGINFO flag is not set for that signal.

1163 Signals generated by the *kill()* function or other events that cause signals to occur, such as
1164 detection of hardware faults, *alarm()* timer expiration, or terminal activity, and for which the
1165 implementation does not support queuing, shall have no effect on signals already queued for the
1166 same signal number.

1167 When multiple unblocked signals, all in the range SIGRTMIN to SIGRTMAX, are pending, the
1168 behavior shall be as if the implementation delivers the pending unblocked signal with the
1169 lowest signal number within that range. No other ordering of signal delivery is specified.

1170 If, when a pending signal is delivered, there are additional signals queued to that signal number,
1171 the signal shall remain pending. Otherwise, the pending indication shall be reset.

1172 Multi-threaded programs can use an alternate event notification mechanism. When a
1173 notification is processed, and the *sigev_notify* member of the **sigevent** structure has the value
1174 SIGEV_THREAD, the function *sigev_notify_function* is called with parameter *sigev_value*.

1175 The function shall be executed in an environment as if it were the *start_routine* for a newly
1176 created thread with thread attributes specified by *sigev_notify_attributes*. If *sigev_notify_attributes*
1177 is NULL, the behavior shall be as if the thread were created with the *detachstate* attribute set to
1178 PTHREAD_CREATE_DETACHED. Supplying an attributes structure with a *detachstate* attribute
1179 of PTHREAD_CREATE_JOINABLE results in undefined behavior. The signal mask of this
1180 thread is implementation-defined.

1181 2.4.3 Signal Actions

1182 There are three types of action that can be associated with a signal: SIG_DFL, SIG_IGN, or a
1183 pointer to a function. Initially, all signals shall be set to SIG_DFL or SIG_IGN prior to entry of
1184 the *main()* routine (see the *exec* functions). The actions prescribed by these values are as follows.

1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222

SIG_DFL

Signal-specific default action.

The default actions for the signals defined in this volume of IEEE Std 1003.1-200x are specified under `<signal.h>`. The default actions for the realtime signals in the range SIGRTMIN to SIGRTMAX shall be to terminate the process abnormally.

If the default action is to stop the process, the execution of that process is temporarily suspended. When a process stops, a SIGCHLD signal shall be generated for its parent process, unless the parent process has set the SA_NOCLDSTOP flag. While a process is stopped, any additional signals that are sent to the process shall not be delivered until the process is continued, except SIGKILL which always terminates the receiving process. A process that is a member of an orphaned process group shall not be allowed to stop in response to the SIGTSTP, SIGTTIN, or SIGTTOU signals. In cases where delivery of one of these signals would stop such a process, the signal shall be discarded.

Setting a signal action to SIG_DFL for a signal that is pending, and whose default action is to ignore the signal (for example, SIGCHLD), shall cause the pending signal to be discarded, whether or not it is blocked. Any queued values pending shall be discarded and the resources used to queue them shall be released and returned to the system for other use.

The default action for SIGCONT is to resume execution at the point where the process was stopped, after first handling any pending unblocked signals.

XSI When a stopped process is continued, a SIGCHLD signal may be generated for its parent process, unless the parent process has set the SA_NOCLDSTOP flag.

SIG_IGN

Ignore signal.

Delivery of the signal shall have no effect on the process. The behavior of a process is undefined after it ignores a SIGFPE, SIGILL, SIGSEGV, or SIGBUS signal that was not generated by `kill()`, `sigqueue()`, or `raise()`.

The system shall not allow the action for the signals SIGKILL or SIGSTOP to be set to SIG_IGN.

Setting a signal action to SIG_IGN for a signal that is pending shall cause the pending signal to be discarded, whether or not it is blocked.

XSI If a process sets the action for the SIGCHLD signal to SIG_IGN, the behavior is unspecified, except as specified below.

If the action for the SIGCHLD signal is set to SIG_IGN, child processes of the calling processes shall not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited-for children that were transformed into zombie processes, it shall block until all of its children terminate, and `wait()`, `waitid()`, and `waitpid()` shall fail and set `errno` to [ECHILD].

Any queued values pending shall be discarded and the resources used to queue them shall be released and made available to queue other signals.

1223 **Pointer to a Function**

1224 Catch signal.

1225 On delivery of the signal, the receiving process is to execute the signal-catching function at the
 1226 specified address. After returning from the signal-catching function, the receiving process shall
 1227 resume execution at the point at which it was interrupted.

1228 If the SA_SIGINFO flag for the signal is cleared, the signal-catching function shall be entered as
 1229 a C-language function call as follows:

1230

```
void func(int signo);
```

1231 If the SA_SIGINFO flag for the signal is set, the signal-catching function shall be entered as a C-
 1232 language function call as follows:

1233

```
void func(int signo, siginfo_t *info, void *context);
```

1234 where *func* is the specified signal-catching function, *signo* is the signal number of the signal
 1235 being delivered, and *info* is a pointer to a **siginfo_t** structure defined in **<signal.h>** containing at
 1236 least the following members:

Member Type	Member Name	Description
int	<i>si_signo</i>	Signal number.
int	<i>si_code</i>	Cause of the signal.
union signal	<i>si_value</i>	Signal value.

1241 The *si_signo* member shall contain the signal number. This shall be the same as the *signo*
 1242 parameter. The *si_code* member shall contain a code identifying the cause of the signal. The
 1243 following values are defined for *si_code*:

1244 SI_USER The signal was sent by the *kill()* function. The implementation may set *si_code*
 1245 to SI_USER if the signal was sent by the *raise()* or *abort()* functions or any
 1246 similar functions provided as implementation extensions.

1247 SI_QUEUE The signal was sent by the *sigqueue()* function.

1248 SI_TIMER The signal was generated by the expiration of a timer set by *timer_settime()*.

1249 SI_ASYNCIO The signal was generated by the completion of an asynchronous I/O request.

1250 MSG SI_MSGQ The signal was generated by the arrival of a message on an empty message
 1251 queue.

1252 If the signal was not generated by one of the functions or events listed above, the *si_code* shall be
 1253 set to an implementation-defined value that is not equal to any of the values defined above.

1254 If *si_code* is one of SI_QUEUE, SI_TIMER, SI_ASYNCIO, or SI_MSGQ, then *si_value* shall
 1255 contain the application-specified signal value. Otherwise, the contents of *si_value* are undefined.

1256 The behavior of a process is undefined after it returns normally from a signal-catching function
 1257 for a SIGBUS, SIGFPE, SIGILL, or SIGSEGV signal that was not generated by *kill()*, *sigqueue()*,
 1258 or *raise()*.

1259 The system shall not allow a process to catch the signals SIGKILL and SIGSTOP.

1260 If a process establishes a signal-catching function for the SIGCHLD signal while it has a
 1261 terminated child process for which it has not waited, it is unspecified whether a SIGCHLD
 1262 signal is generated to indicate that child process.

1263 When signal-catching functions are invoked asynchronously with process execution, the
 1264 behavior of some of the functions defined by this volume of IEEE Std 1003.1-200x is unspecified

1265 if they are called from a signal-catching function.

1266 The following table defines a set of functions that shall be either reentrant or non-interruptible
1267 by signals and shall be async-signal-safe. Therefore, applications may invoke them, without
1268 restriction, from signal-catching functions:

1269	<code>_Exit()</code>	<code>faccessat()</code>	<code>link()</code>	<code>rmdir()</code>	<code>socket()</code>
1270	<code>_exit()</code>	<code>fchmod()</code>	<code>linkat()</code>	<code>select()</code>	<code>socketpair()</code>
1271	<code>abort()</code>	<code>fchmodat()</code>	<code>listen()</code>	<code>sem_post()</code>	<code>stat()</code>
1272	<code>accept()</code>	<code>fchown()</code>	<code>lseek()</code>	<code>send()</code>	<code>symlink()</code>
1273	<code>access()</code>	<code>fchownat()</code>	<code>lstat()</code>	<code>sendmsg()</code>	<code>symlinkat()</code>
1274	<code>aio_error()</code>	<code>fcntl()</code>	<code>mkdir()</code>	<code>sendto()</code>	<code>sysconf()</code>
1275	<code>aio_return()</code>	<code>fdatasync()</code>	<code>mkdirat()</code>	<code>setgid()</code>	<code>tcdrain()</code>
1276	<code>aio_suspend()</code>	<code>execve()</code>	<code>mkfifo()</code>	<code>setpgid()</code>	<code>tcflow()</code>
1277	<code>alarm()</code>	<code>fork()</code>	<code>mkfifoat()</code>	<code>setsid()</code>	<code>tcflush()</code>
1278	<code>bind()</code>	<code>fpathconf()</code>	<code>mknodat()</code>	<code>setsockopt()</code>	<code>tcgetattr()</code>
1279	<code>cfgetispeed()</code>	<code>fstat()</code>	<code>open()</code>	<code>setuid()</code>	<code>tcgetpgrp()</code>
1280	<code>cfgetospeed()</code>	<code>fstatat()</code>	<code>openat()</code>	<code>shutdown()</code>	<code>tcsendbreak()</code>
1281	<code>cfsetispeed()</code>	<code>fsync()</code>	<code>pathconf()</code>	<code>sigaction()</code>	<code>tcsetattr()</code>
1282	<code>cfsetospeed()</code>	<code>ftruncate()</code>	<code>pause()</code>	<code>sigaddset()</code>	<code>tcsetpgrp()</code>
1283	<code>chdir()</code>	<code>futimesat()</code>	<code>pipe()</code>	<code>sigdelset()</code>	<code>time()</code>
1284	<code>chmod()</code>	<code>getegid()</code>	<code>poll()</code>	<code>sigemptyset()</code>	<code>timer_getoverrun()</code>
1285	<code>chown()</code>	<code>geteuid()</code>	<code>posix_trace_event()</code>	<code>sigfillset()</code>	<code>timer_gettime()</code>
1286	<code>clock_gettime()</code>	<code>getgid()</code>	<code>pselect()</code>	<code>sigismember()</code>	<code>timer_settime()</code>
1287	<code>close()</code>	<code>getgroups()</code>	<code>raise()</code>	<code>signal()</code>	<code>times()</code>
1288	<code>connect()</code>	<code>getpeername()</code>	<code>read()</code>	<code>sigpause()</code>	<code>umask()</code>
1289	<code>creat()</code>	<code>getpgrp()</code>	<code>readlink()</code>	<code>sigpending()</code>	<code>uname()</code>
1290	<code>dup()</code>	<code>getpid()</code>	<code>readlinkat()</code>	<code>sigprocmask()</code>	<code>unlink()</code>
1291	<code>dup2()</code>	<code>getppid()</code>	<code>recv()</code>	<code>sigqueue()</code>	<code>unlinkat()</code>
1292	<code>execl()</code>	<code>getsockname()</code>	<code>recvfrom()</code>	<code>sigset()</code>	<code>utime()</code>
1293	<code>execle()</code>	<code>getsockopt()</code>	<code>recvmsg()</code>	<code>sigsuspend()</code>	<code>wait()</code>
1294	<code>execv()</code>	<code>getuid()</code>	<code>rename()</code>	<code>sleep()</code>	<code>waitpid()</code>
1295	<code>execve()</code>	<code>kill()</code>	<code>renameat()</code>	<code>socketmark()</code>	<code>write()</code>

1296 All functions not in the above table are considered to be unsafe with respect to signals. In the
1297 presence of signals, all functions defined by this volume of IEEE Std 1003.1-200x shall behave as
1298 defined when called from or interrupted by a signal-catching function, with a single exception:
1299 when a signal interrupts an unsafe function and the signal-catching function calls an unsafe
1300 function, the behavior is undefined.

1301 When a signal is delivered to a thread, if the action of that signal specifies termination, stop, or
1302 continue, the entire process shall be terminated, stopped, or continued, respectively.

1303 2.4.4 Signal Effects on Other Functions

1304 Signals affect the behavior of certain functions defined by this volume of IEEE Std 1003.1-200x if
1305 delivered to a process while it is executing such a function. If the action of the signal is to
1306 terminate the process, the process shall be terminated and the function shall not return. If the
1307 action of the signal is to stop the process, the process shall stop until continued or terminated.
1308 Generation of a SIGCONT signal for the process shall cause the process to be continued, and the
1309 original function shall continue at the point the process was stopped. If the action of the signal is
1310 to invoke a signal-catching function, the signal-catching function shall be invoked; in this case
1311 the original function is said to be “interrupted” by the signal. If the signal-catching function
1312 executes a **return** statement, the behavior of the interrupted function shall be as described
1313 individually for that function, except as noted for unsafe functions. Signals that are ignored shall

1314 not affect the behavior of any function; signals that are blocked shall not affect the behavior of
 1315 any function until they are unblocked and then delivered, except as specified for the *sigpending()*
 1316 and *sigwait()* functions.

1317 2.5 Standard I/O Streams

1318 CX A stream is associated with an external file (which may be a physical device) or memory buffer
 1319 CX by “opening” a file or buffer. This may involve “creating” a new file. Creating an existing file
 1320 causes its former contents to be discarded if necessary. If a file can support positioning requests
 1321 (such as a disk file, as opposed to a terminal), then a “file position indicator” associated with the
 1322 stream is positioned at the start (byte number 0) of the file, unless the file is opened with append
 1323 mode, in which case it is implementation-defined whether the file position indicator is initially
 1324 positioned at the beginning or end of the file. The file position indicator is maintained by
 1325 subsequent reads, writes, and positioning requests, to facilitate an orderly progression through
 1326 the file. All input takes place as if bytes were read by successive calls to *fgetc()*; all output takes
 1327 place as if bytes were written by successive calls to *fputc()*.

1328 When a stream is “unbuffered”, bytes are intended to appear from the source or at the
 1329 destination as soon as possible; otherwise, bytes may be accumulated and transmitted as a
 1330 block. When a stream is “fully buffered”, bytes are intended to be transmitted as a block when a
 1331 buffer is filled. When a stream is “line buffered”, bytes are intended to be transmitted as a block
 1332 when a newline byte is encountered. Furthermore, bytes are intended to be transmitted as a
 1333 block when a buffer is filled, when input is requested on an unbuffered stream, or when input is
 1334 requested on a line-buffered stream that requires the transmission of bytes. Support for these
 1335 characteristics is implementation-defined, and may be affected via *setbuf()* and *setvbuf()*.

1336 A file may be disassociated from a controlling stream by “closing” the file. Output streams are
 1337 flushed (any unwritten buffer contents are transmitted) before the stream is disassociated from
 1338 the file. The value of a pointer to a **FILE** object is unspecified after the associated file is closed
 1339 (including the standard streams).

1340 A file may be subsequently reopened, by the same or another program execution, and its
 1341 contents reclaimed or modified (if it can be repositioned at its start). If the *main()* function
 1342 returns to its original caller, or if the *exit()* function is called, all open files are closed (hence all
 1343 output streams are flushed) before program termination. Other paths to program termination,
 1344 such as calling *abort()*, need not close all files properly.

1345 The address of the **FILE** object used to control a stream may be significant; a copy of a **FILE**
 1346 object need not necessarily serve in place of the original.

1347 At program start-up, three streams are predefined and need not be opened explicitly: *standard*
 1348 *input* (for reading conventional input), *standard output* (for writing conventional output), and
 1349 *standard error* (for writing diagnostic output). When opened, the standard error stream is not
 1350 fully buffered; the standard input and standard output streams are fully buffered if and only if
 1351 the stream can be determined not to refer to an interactive device.

1352 CX A stream associated with a memory buffer shall have the same operations for text files that a
 1353 stream associated with an external file would have. In addition, the stream orientation shall be
 1354 determined in exactly the same fashion.

1355 Input and output operations on a stream associated with a memory buffer by a call to
 1356 *fmemopen()* shall be constrained by the implementation to take place within the bounds of the
 1357 memory buffer. In the case of a stream opened by *open_memstream()* or *open_wmemstream()*, the
 1358 memory area shall grow dynamically to accommodate write operations as necessary. For output,
 1359 data is moved from the buffer provided by *setvbuf()* to the memory stream during a flush or
 1360 close operation.

2.5.1 Interaction of File Descriptors and Standard I/O Streams

CX This section describes the interaction of file descriptors and standard I/O streams. The functionality described in this section is an extension to the ISO C standard (and the rest of this section is not further CX shaded).

An open file description may be accessed through a file descriptor, which is created using functions such as *open()* or *pipe()*, or through a stream, which is created using functions such as *fopen()* or *popen()*. Either a file descriptor or a stream is called a “handle” on the open file description to which it refers; an open file description may have several handles.

Handles can be created or destroyed by explicit user action, without affecting the underlying open file description. Some of the ways to create them include *fcntl()*, *dup()*, *fdopen()*, *fileno()*, and *fork()*. They can be destroyed by at least *fclose()*, *close()*, and the *exec* functions.

A file descriptor that is never used in an operation that could affect the file offset (for example, *read()*, *write()*, or *lseek()*) is not considered a handle for this discussion, but could give rise to one (for example, as a consequence of *fdopen()*, *dup()*, or *fork()*). This exception does not include the file descriptor underlying a stream, whether created with *fopen()* or *fdopen()*, so long as it is not used directly by the application to affect the file offset. The *read()* and *write()* functions implicitly affect the file offset; *lseek()* explicitly affects it.

The result of function calls involving any one handle (the “active handle”) is defined elsewhere in this volume of IEEE Std 1003.1-200x, but if two or more handles are used, and any one of them is a stream, the application shall ensure that their actions are coordinated as described below. If this is not done, the result is undefined.

A handle which is a stream is considered to be closed when either an *fclose()* or *freopen()* is executed on it (the result of *freopen()* is a new stream, which cannot be a handle on the same open file description as its previous value), or when the process owning that stream terminates with *exit()*, *abort()*, or due to a signal. A file descriptor is closed by *close()*, *_exit()*, or the *exec* functions when *FD_CLOEXEC* is set on that file descriptor.

For a handle to become the active handle, the application shall ensure that the actions below are performed between the last use of the handle (the current active handle) and the first use of the second handle (the future active handle). The second handle then becomes the active handle. All activity by the application affecting the file offset on the first handle shall be suspended until it again becomes the active file handle. (If a stream function has as an underlying function one that affects the file offset, the stream function shall be considered to affect the file offset.)

The handles need not be in the same process for these rules to apply.

Note that after a *fork()*, two handles exist where one existed before. The application shall ensure that, if both handles can ever be accessed, they are both in a state where the other could become the active handle first. The application shall prepare for a *fork()* exactly as if it were a change of active handle. (If the only action performed by one of the processes is one of the *exec* functions or *_exit()* (not *exit()*), the handle is never accessed in that process.)

For the first handle, the first applicable condition below applies. After the actions required below are taken, if the handle is still open, the application can close it.

- If it is a file descriptor, no action is required.
- If the only further action to be performed on any handle to this open file descriptor is to close it, no action need be taken.
- If it is a stream which is unbuffered, no action need be taken.

- 1405 • If it is a stream which is line buffered, and the last byte written to the stream was a
1406 <newline> (that is, as if a:

1407 `putc('\n')`

1408 was the most recent operation on that stream), no action need be taken.

- 1409 • If it is a stream which is open for writing or appending (but not also open for reading), the
1410 application shall either perform an *fflush()*, or the stream shall be closed.
- 1411 • If the stream is open for reading and it is at the end of the file (*feof()* is true), no action need
1412 be taken.
- 1413 • If the stream is open with a mode that allows reading and the underlying open file
1414 description refers to a device that is capable of seeking, the application shall either perform
1415 an *fflush()*, or the stream shall be closed.

1416 Otherwise, the result is undefined.

1417 For the second handle:

- 1418 • If any previous active handle has been used by a function that explicitly changed the file
1419 offset, except as required above for the first handle, the application shall perform an *lseek()*
1420 or *fseek()* (as appropriate to the type of handle) to an appropriate location.

1421 If the active handle ceases to be accessible before the requirements on the first handle, above,
1422 have been met, the state of the open file description becomes undefined. This might occur
1423 during functions such as a *fork()* or *_exit()*.

1424 The *exec* functions make inaccessible all streams that are open at the time they are called,
1425 independent of which streams or file descriptors may be available to the new process image.

1426 When these rules are followed, regardless of the sequence of handles used, implementations
1427 shall ensure that an application, even one consisting of several processes, shall yield correct
1428 results: no data shall be lost or duplicated when writing, and all data shall be written in order,
1429 except as requested by seeks. It is implementation-defined whether, and under what conditions,
1430 all input is seen exactly once.

1431 If the rules above are not followed, the result is unspecified.

1432 Each function that operates on a stream is said to have zero or more “underlying functions”.
1433 This means that the stream function shares certain traits with the underlying functions, but does
1434 not require that there be any relation between the implementations of the stream function and its
1435 underlying functions.

1436 2.5.2 Stream Orientation and Encoding Rules

1437 For conformance to the ISO/IEC 9899:1999 standard, the definition of a stream includes an
1438 “orientation”. After a stream is associated with an external file, but before any operations are
1439 performed on it, the stream is without orientation. Once a wide-character input/output function
1440 has been applied to a stream without orientation, the stream shall become “wide-oriented”.
1441 Similarly, once a byte input/output function has been applied to a stream without orientation,
1442 the stream shall become “byte-oriented”. Only a call to the *freopen()* function or the *fwide()*
1443 function can otherwise alter the orientation of a stream.

1444 A successful call to *freopen()* shall remove any orientation. The three predefined streams *standard*
1445 *input*, *standard output*, and *standard error* shall be unoriented at program start-up.

1446 Byte input/output functions cannot be applied to a wide-oriented stream, and wide-character
1447 input/output functions cannot be applied to a byte-oriented stream. The remaining stream

1448 operations shall not affect and shall not be affected by a stream's orientation, except for the
1449 following additional restriction:

- 1450 • For wide-oriented streams, after a successful call to a file-positioning function that leaves
1451 the file position indicator prior to the end-of-file, a wide-character output function can
1452 overwrite a partial character; any file contents beyond the byte(s) written are henceforth
1453 undefined.

1454 Each wide-oriented stream has an associated **mbstate_t** object that stores the current parse state
1455 of the stream. A successful call to *fgetpos()* shall store a representation of the value of this
1456 **mbstate_t** object as part of the value of the **fpos_t** object. A later successful call to *fsetpos()* using
1457 the same stored **fpos_t** value shall restore the value of the associated **mbstate_t** object as well as
1458 the position within the controlled stream.

1459 Implementations that support multiple encoding rules associate an encoding rule with the
1460 stream. The encoding rule shall be determined by the setting of the *LC_CTYPE* category in the
1461 current locale at the time when the stream becomes wide-oriented. As with the stream's
1462 orientation, the encoding rule associated with a stream cannot be changed once it has been set,
1463 except by a successful call to *freopen()* which clears the encoding rule and resets the orientation
1464 to unoriented.

1465 Although wide-oriented streams are conceptually sequences of wide characters, the external file
1466 associated with a wide-oriented stream is a sequence of (possibly multi-byte) characters
1467 generalized as follows:

- 1468 • Multi-byte encodings within files may contain embedded null bytes (unlike multi-byte
1469 encodings valid for use internal to the program).
- 1470 • A file need not begin nor end in the initial shift state.

1471 Moreover, the encodings used for characters may differ among files. Both the nature and choice
1472 of such encodings are implementation-defined.

1473 The wide-character input functions read characters from the stream and convert them to wide
1474 characters as if they were read by successive calls to the *fgetwc()* function. Each conversion shall
1475 occur as if by a call to the *mbrtowc()* function, with the conversion state described by the
1476 stream's own **mbstate_t** object, except the encoding rule associated with the stream is used
1477 instead of the encoding rule implied by the *LC_CTYPE* category of the current locale.

CX

1478 The wide-character output functions convert wide characters to (possibly multi-byte) characters
1479 and write them to the stream as if they were written by successive calls to the *fputwc()* function.
1480 Each conversion shall occur as if by a call to the *wcrtomb()* function, with the conversion state
1481 described by the stream's own **mbstate_t** object, except the encoding rule associated with the
1482 stream is used instead of the encoding rule implied by the *LC_CTYPE* category of the current
1483 locale.

CX

1484 An "encoding error" shall occur if the character sequence presented to the underlying *mbrtowc()*
1485 function does not form a valid (generalized) character, or if the code value passed to the
1486 underlying *wcrtomb()* function does not correspond to a valid (generalized) character. The wide-
1487 character input/output functions and the byte input/output functions store the value of the
1488 macro [EILSEQ] in *errno* if and only if an encoding error occurs.

2.6 STREAMS

OB XSR STREAMS functionality is provided on implementations supporting the XSI STREAMS Option Group. The functionality described in this section is dependent on support of the XSI STREAMS option (and the rest of this section is not further shaded for this option).

STREAMS provides a uniform mechanism for implementing networking services and other character-based I/O. The STREAMS function provides direct access to protocol modules. STREAMS modules are unspecified objects. Access to STREAMS modules is provided by interfaces in IEEE Std 1003.1-200x. Creation of STREAMS modules is outside the scope of IEEE Std 1003.1-200x.

A STREAM is typically a full-duplex connection between a process and an open device or pseudo-device. However, since pipes may be STREAMS-based, a STREAM can be a full-duplex connection between two processes. The STREAM itself exists entirely within the implementation and provides a general character I/O function for processes. It optionally includes one or more intermediate processing modules that are interposed between the process end of the STREAM (STREAM head) and a device driver at the end of the STREAM (STREAM end).

STREAMS I/O is based on messages. There are three types of message:

- *Data messages* containing actual data for input or output
- *Control data* containing instructions for the STREAMS modules and underlying implementation
- Other messages, which include file descriptors

The interface between the STREAM and the rest of the implementation is provided by a set of functions at the STREAM head. When a process calls *write()*, *writv()*, *putmsg()*, *putpmsg()*, or *ioctl()*, messages are sent down the STREAM, and *read()*, *readv()*, *getmsg()*, or *getpmsg()* accepts data from the STREAM and passes it to a process. Data intended for the device at the downstream end of the STREAM is packaged into messages and sent downstream, while data and signals from the device are composed into messages by the device driver and sent upstream to the STREAM head.

When a STREAMS-based device is opened, a STREAM shall be created that contains the STREAM head and the STREAM end (driver). If pipes are STREAMS-based in an implementation, when a pipe is created, two STREAMS shall be created, each containing a STREAM head. Other modules are added to the STREAM using *ioctl()*. New modules are “pushed” onto the STREAM one at a time in last-in, first-out (LIFO) style, as though the STREAM was a push-down stack.

Priority

Message types are classified according to their queuing priority and may be *normal* (non-priority), *priority*, or *high-priority* messages. A message belongs to a particular priority band that determines its ordering when placed on a queue. Normal messages have a priority band of 0 and shall always be placed at the end of the queue following all other messages in the queue. High-priority messages are always placed at the head of a queue, but shall be discarded if there is already a high-priority message in the queue. Their priority band shall be ignored; they are high-priority by virtue of their type. Priority messages have a priority band greater than 0. Priority messages are always placed after any messages of the same or higher priority. High-priority and priority messages are used to send control and data information outside the normal flow of control. By convention, high-priority messages shall not be affected by flow control. Normal and priority messages have separate flow controls.

1534 **Message Parts**

1535 A process may access STREAMS messages that contain a data part, control part, or both. The
 1536 data part is that information which is transmitted over the communication medium and the
 1537 control information is used by the local STREAMS modules. The other types of messages are
 1538 used between modules and are not accessible to processes. Messages containing only a data part
 1539 are accessible via *putmsg()*, *putpmsg()*, *getmsg()*, *getpmsg()*, *read()*, *readv()*, *write()*, or *writv()*.
 1540 Messages containing a control part with or without a data part are accessible via calls to
 1541 *putmsg()*, *putpmsg()*, *getmsg()*, or *getpmsg()*.

1542 **2.6.1 Accessing STREAMS**

1543 A process accesses STREAMS-based files using the standard functions *close()*, *ioctl()*, *getmsg()*,
 1544 *getpmsg()*, *open()*, *pipe()*, *poll()*, *putmsg()*, *putpmsg()*, *read()*, or *write()*. Refer to the applicable
 1545 function definitions for general properties and errors.

1546 Calls to *ioctl()* shall perform control functions on the STREAM associated with the file descriptor
 1547 *fdes*. The control functions may be performed by the STREAM head, a STREAMS module, or
 1548 the STREAMS driver for the STREAM.

1549 STREAMS modules and drivers can detect errors, sending an error message to the STREAM
 1550 head, thus causing subsequent functions to fail and set *errno* to the value specified in the
 1551 message. In addition, STREAMS modules and drivers can elect to fail a particular *ioctl()* request
 1552 alone by sending a negative acknowledgement message to the STREAM head. This shall cause
 1553 just the pending *ioctl()* request to fail and set *errno* to the value specified in the message.

1554 **2.7 XSI Interprocess Communication**

1555 XSI This section describes extensions to support interprocess communication. The functionality
 1556 described in this section shall be provided on implementations that support the XSI option (and
 1557 the rest of this section is not further shaded).

1558 The following message passing, semaphore, and shared memory services form an XSI
 1559 interprocess communication facility. Certain aspects of their operation are common, and are
 1560 defined as follows.

IPC Functions		
<i>msgctl()</i>	<i>semctl()</i>	<i>shmctl()</i>
<i>msgget()</i>	<i>semget()</i>	<i>shmdt()</i>
<i>msgrcv()</i>	<i>semop()</i>	<i>shmget()</i>
<i>msgsnd()</i>	<i>shmat()</i>	

1566 Another interprocess communication facility is provided by functions in the Realtime Option
 1567 Group; see [Section 2.8](#) (on page 40).

1568 **2.7.1 IPC General Description**

1569 Each individual shared memory segment, message queue, and semaphore set shall be identified
 1570 by a unique positive integer, called, respectively, a shared memory identifier, *shm_{id}*, a semaphore
 1571 identifier, *sem_{id}*, and a message queue identifier, *msg_{id}*. The identifiers shall be returned by calls
 1572 to *shmget()*, *semget()*, and *msgget()*, respectively.

1573 Associated with each identifier is a data structure which contains data related to the operations
 1574 which may be or may have been performed; see the Base Definitions volume of
 1575 IEEE Std 1003.1-200x, `<sys/shm.h>`, `<sys/sem.h>`, and `<sys/msg.h>` for their descriptions.

1576 Each of the data structures contains both ownership information and an `ipc_perm` structure (see

1577 the Base Definitions volume of IEEE Std 1003.1-200x, `<sys/ipc.h>`) which are used in conjunction
 1578 to determine whether or not read/write (read/alter for semaphores) permissions should be
 1579 granted to processes using the IPC facilities. The *mode* member of the `ipc_perm` structure acts as
 1580 a bit field which determines the permissions.

1581 The values of the bits are given below in octal notation.

Bit	Meaning
0400	Read by user.
0200	Write by user.
0040	Read by group.
0020	Write by group.
0004	Read by others.
0002	Write by others.

1589 The name of the `ipc_perm` structure is *shm_perm*, *sem_perm*, or *msg_perm*, depending on which
 1590 service is being used. In each case, read and write/alter permissions shall be granted to a
 1591 process if one or more of the following are true ("*xxx*" is replaced by *shm*, *sem*, or *msg*, as
 1592 appropriate):

- 1593 • The process has appropriate privileges.
- 1594 • The effective user ID of the process matches *xxx_perm.cuid* or *xxx_perm.uid* in the data
 1595 structure associated with the IPC identifier, and the appropriate bit of the *user* field in
 1596 *xxx_perm.mode* is set.
- 1597 • The effective user ID of the process does not match *xxx_perm.cuid* or *xxx_perm.uid* but the
 1598 effective group ID of the process matches *xxx_perm.cgid* or *xxx_perm.gid* in the data
 1599 structure associated with the IPC identifier, and the appropriate bit of the *group* field in
 1600 *xxx_perm.mode* is set.
- 1601 • The effective user ID of the process does not match *xxx_perm.cuid* or *xxx_perm.uid* and the
 1602 effective group ID of the process does not match *xxx_perm.cgid* or *xxx_perm.gid* in the data
 1603 structure associated with the IPC identifier, but the appropriate bit of the *other* field in
 1604 *xxx_perm.mode* is set.

1605 Otherwise, the permission shall be denied.

1606 2.8 Realtime

1607 This section defines functions to support the source portability of applications with realtime
 1608 requirements. The presence of some of these functions is dependent on support for
 1609 implementation options described in the text.

1610 The specific functional areas included in this section and their scope include the following. Full
 1611 definitions of these terms can be found in the Base Definitions volume of IEEE Std 1003.1-200x,
 1612 Chapter 3, Definitions.

- 1613 • Semaphores
- 1614 • Process Memory Locking
- 1615 • Memory Mapped Files and Shared Memory Objects
- 1616 • Priority Scheduling

- 1617 • Realtime Signal Extension
- 1618 • Timers
- 1619 • Interprocess Communication
- 1620 • Synchronized Input and Output
- 1621 • Asynchronous Input and Output

1622 All the realtime functions defined in this volume of IEEE Std 1003.1-200x are portable, although
1623 some of the numeric parameters used by an implementation may have hardware dependencies.

1624 2.8.1 Realtime Signals

1625 See [Section 2.4.2](#) (on page 29).

1626 2.8.2 Asynchronous I/O

1627 An asynchronous I/O control block structure **aiocb** is used in many asynchronous I/O
1628 functions. It is defined in the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>** and has
1629 at least the following members:

Member Type	Member Name	Description
int	<i>aio_fildes</i>	File descriptor.
off_t	<i>aio_offset</i>	File offset.
volatile void*	<i>aio_buf</i>	Location of buffer.
size_t	<i>aio_nbytes</i>	Length of transfer.
int	<i>aio_reqprio</i>	Request priority offset.
struct sigevent	<i>aio_sigevent</i>	Signal number and value.
int	<i>aio_lio_opcode</i>	Operation to be performed.

1638 The *aio_fildes* element is the file descriptor on which the asynchronous operation is performed.

1639 If O_APPEND is not set for the file descriptor *aio_fildes* and if *aio_fildes* is associated with a
1640 device that is capable of seeking, then the requested operation takes place at the absolute
1641 position in the file as given by *aio_offset*, as if *lseek()* were called immediately prior to the
1642 operation with an *offset* argument equal to *aio_offset* and a *whence* argument equal to SEEK_SET.
1643 If O_APPEND is set for the file descriptor, or if *aio_fildes* is associated with a device that is
1644 incapable of seeking, write operations append to the file in the same order as the calls were
1645 made, with the following exception: under implementation-defined circumstances, such as
1646 operation on a multi-processor or when requests of differing priorities are submitted at the same
1647 time, the ordering restriction may be relaxed. Since there is no way for a strictly conforming
1648 application to determine whether this relaxation applies, all strictly conforming applications
1649 which rely on ordering of output shall be written in such a way that they will operate correctly if
1650 the relaxation applies. After a successful call to enqueue an asynchronous I/O operation, the
1651 value of the file offset for the file is unspecified. The *aio_nbytes* and *aio_buf* elements are the same
1652 as the *nbyte* and *buf* arguments defined by *read()* and *write()*, respectively.

1653 If _POSIX_PRIORITIZED_IO and _POSIX_PRIORITY_SCHEDULING are defined, then
1654 asynchronous I/O is queued in priority order, with the priority of each asynchronous operation
1655 based on the current scheduling priority of the calling process. The *aio_reqprio* member can be
1656 used to lower (but not raise) the asynchronous I/O operation priority and is within the range
1657 zero through {AIO_PRIO_DELTA_MAX}, inclusive. Unless both _POSIX_PRIORITIZED_IO and
1658 _POSIX_PRIORITY_SCHEDULING are defined, the order of processing asynchronous I/O
1659 requests is unspecified. When both _POSIX_PRIORITIZED_IO and
1660 _POSIX_PRIORITY_SCHEDULING are defined, the order of processing of requests submitted
1661 by processes whose schedulers are not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC is

1662 unspecified. The priority of an asynchronous request is computed as (process scheduling
 1663 priority) minus *aio_reqprio*. The priority assigned to each asynchronous I/O request is an
 1664 indication of the desired order of execution of the request relative to other asynchronous I/O
 1665 requests for this file. If `_POSIX_PRIORITIZED_IO` is defined, requests issued with the same
 1666 priority to a character special file are processed by the underlying device in FIFO order; the
 1667 order of processing of requests of the same priority issued to files that are not character special
 1668 files is unspecified. Numerically higher priority values indicate requests of higher priority. The
 1669 value of *aio_reqprio* has no effect on process scheduling priority. When prioritized asynchronous
 1670 I/O requests to the same file are blocked waiting for a resource required for that I/O operation,
 1671 the higher-priority I/O requests shall be granted the resource before lower-priority I/O requests
 1672 are granted the resource. The relative priority of asynchronous I/O and synchronous I/O is
 1673 implementation-defined. If `_POSIX_PRIORITIZED_IO` is defined, the implementation shall
 1674 define for which files I/O prioritization is supported.

1675 The *aio_sigevent* determines how the calling process shall be notified upon I/O completion, as
 1676 specified in Section 2.4.1 (on page 28). If *aio_sigevent.sigev_notify* is `SIGEV_NONE`, then no
 1677 signal shall be posted upon I/O completion, but the error status for the operation and the return
 1678 status for the operation shall be set appropriately.

1679 The *aio_lio_opcode* field is used only by the *lio_listio()* call. The *lio_listio()* call allows multiple
 1680 asynchronous I/O operations to be submitted at a single time. The function takes as an
 1681 argument an array of pointers to **aiocb** structures. Each **aiocb** structure indicates the operation to
 1682 be performed (read or write) via the *aio_lio_opcode* field.

1683 The address of the **aiocb** structure is used as a handle for retrieving the error status and return
 1684 status of the asynchronous operation while it is in progress.

1685 The **aiocb** structure and the data buffers associated with the asynchronous I/O operation are
 1686 being used by the system for asynchronous I/O while, and only while, the error status of the
 1687 asynchronous operation is equal to `[EINPROGRESS]`. Applications shall not modify the **aiocb**
 1688 structure while the structure is being used by the system for asynchronous I/O.

1689 The return status of the asynchronous operation is the number of bytes transferred by the I/O
 1690 operation. If the error status is set to indicate an error completion, then the return status is set to
 1691 the return value that the corresponding *read()*, *write()*, or *fsync()* call would have returned.
 1692 When the error status is not equal to `[EINPROGRESS]`, the return status shall reflect the return
 1693 status of the corresponding synchronous operation.

1694 2.8.3 Memory Management

1695 2.8.3.1 Memory Locking

1696 MLR Range memory locking operations are defined in terms of pages. Implementations may restrict
 1697 the size and alignment of range lockings to be on page-size boundaries. The page size, in bytes,
 1698 is the value of the configurable system variable `{PAGESIZE}`. If an implementation has no
 1699 restrictions on size or alignment, it may specify a 1-byte page size.

1700 ML | MLR Memory locking guarantees the residence of portions of the address space. It is implementation-
 1701 defined whether locking memory guarantees fixed translation between virtual addresses (as
 1702 seen by the process) and physical addresses. Per-process memory locks are not inherited across a
 1703 *fork()*, and all memory locks owned by a process are unlocked upon *exec* or process termination.
 1704 Unmapping of an address range removes any memory locks established on that address range
 1705 by this process.

1706 2.8.3.2 *Memory Mapped Files*

1707 Range memory mapping operations are defined in terms of pages. Implementations may
 1708 restrict the size and alignment of range mappings to be on page-size boundaries. The page size,
 1709 in bytes, is the value of the configurable system variable {PAGESIZE}. If an implementation has
 1710 no restrictions on size or alignment, it may specify a 1-byte page size.

1711 Memory mapped files provide a mechanism that allows a process to access files by directly
 1712 incorporating file data into its address space. Once a file is mapped into a process address space,
 1713 the data can be manipulated as memory. If more than one process maps a file, its contents are
 1714 shared among them. If the mappings allow shared write access, then data written into the
 1715 memory object through the address space of one process appears in the address spaces of all
 1716 processes that similarly map the same portion of the memory object.

1717 SHM Shared memory objects are named regions of storage that may be independent of the file system
 1718 and can be mapped into the address space of one or more processes to allow them to share the
 1719 associated memory.

1720 SHM An *unlink()* of a file or *shm_unlink()* of a shared memory object, while causing the removal of
 1721 the name, does not unmap any mappings established for the object. Once the name has been
 1722 removed, the contents of the memory object are preserved as long as it is referenced. The
 1723 memory object remains referenced as long as a process has the memory object open or has some
 1724 area of the memory object mapped.

1725 2.8.3.3 *Memory Protection*

1726 When an object is mapped, various application accesses to the mapped region may result in
 1727 signals. In this context, SIGBUS is used to indicate an error using the mapped object, and
 1728 SIGSEGV is used to indicate a protection violation or misuse of an address:

- 1729 • A mapping may be restricted to disallow some types of access.
- 1730 • Write attempts to memory that was mapped without write access, or any access to
 1731 memory mapped PROT_NONE, shall result in a SIGSEGV signal.
- 1732 • References to unmapped addresses shall result in a SIGSEGV signal.
- 1733 • Reference to whole pages within the mapping, but beyond the current length of the object,
 1734 shall result in a SIGBUS signal.
- 1735 • The size of the object is unaffected by access beyond the end of the object (even if a
 1736 SIGBUS is not generated).

1737 2.8.3.4 *Typed Memory Objects*

1738 TYM The functionality described in this section shall be provided on implementations that support
 1739 the Typed Memory Objects option (and the rest of this section is not further shaded for this
 1740 option).

1741 Implementations may support the Typed Memory Objects option independently of support for
 1742 memory mapped files or shared memory objects. Typed memory objects are implementation-
 1743 configurable named storage pools accessible from one or more processors in a system, each via
 1744 one or more ports, such as backplane buses, LANs, I/O channels, and so on. Each valid
 1745 combination of a storage pool and a port is identified through a name that is defined at system
 1746 configuration time, in an implementation-defined manner; the name may be independent of the
 1747 file system. Using this name, a typed memory object can be opened and mapped into process
 1748 address space. For a given storage pool and port, it is necessary to support both dynamic
 1749 allocation from the pool as well as mapping at an application-supplied offset within the pool;
 1750 when dynamic allocation has been performed, subsequent deallocation must be supported.
 1751 Lastly, accessing typed memory objects from different ports requires a method for obtaining the

1752 offset and length of contiguous storage of a region of typed memory (dynamically allocated or
 1753 not); this allows typed memory to be shared among processes and/or processors while being
 1754 accessed from the desired port.

1755 **2.8.4 Process Scheduling**

1756 PS The functionality described in this section shall be provided on implementations that support
 1757 the Process Scheduling option (and the rest of this section is not further shaded for this option).

1758 **Scheduling Policies**

1759 The scheduling semantics described in this volume of IEEE Std 1003.1-200x are defined in terms
 1760 of a conceptual model that contains a set of thread lists. No implementation structures are
 1761 necessarily implied by the use of this conceptual model. It is assumed that no time elapses
 1762 during operations described using this model, and therefore no simultaneous operations are
 1763 possible. This model discusses only processor scheduling for runnable threads, but it should be
 1764 noted that greatly enhanced predictability of realtime applications results if the sequencing of
 1765 other resources takes processor scheduling policy into account.

1766 There is, conceptually, one thread list for each priority. A runnable thread will be on the thread
 1767 list for that thread's priority. Multiple scheduling policies shall be provided. Each non-empty
 1768 thread list is ordered, contains a head as one end of its order, and a tail as the other. The purpose
 1769 of a scheduling policy is to define the allowable operations on this set of lists (for example,
 1770 moving threads between and within lists).

1771 The POSIX model treats a "process" as an aggregation of system resources, including one or
 1772 more threads that may be scheduled by the operating system on the processor(s) it controls.
 1773 Although a process has its own set of scheduling attributes, these have an indirect effect (if any)
 1774 on the scheduling behavior of individual threads as described below.

1775 Each thread shall be controlled by an associated scheduling policy and priority. These
 1776 parameters may be specified by explicit application execution of the *pthread_setschedparam()*
 1777 function. Additionally, the scheduling parameters of a thread (but not its scheduling policy) may
 1778 be changed by application execution of the *pthread_setschedprio()* function.

1779 Each process shall be controlled by an associated scheduling policy and priority. These
 1780 parameters may be specified by explicit application execution of the *sched_setscheduler()* or
 1781 *sched_setparam()* functions.

1782 The effect of the process scheduling attributes on individual threads in the process is dependent
 1783 on the scheduling contention scope of the threads (see [Section 2.9.4](#) (on page 52)):

- 1784 • For threads with system scheduling contention scope, the process scheduling attributes
 1785 shall have no effect on the scheduling attributes or behavior either of the thread or an
 1786 underlying kernel scheduling entity dedicated to that thread.
- 1787 • For threads with process scheduling contention scope, the process scheduling attributes
 1788 shall have no effect on the scheduling attributes of the thread. However, any underlying
 1789 kernel scheduling entity used by these threads shall at all times behave as specified by the
 1790 scheduling attributes of the containing process, and this behavior may affect the
 1791 scheduling behavior of the process contention scope threads. For example, a process
 1792 contention scope thread with scheduling policy SCHED_FIFO and the system maximum
 1793 priority *H* (the value returned by *sched_get_priority_max*(SCHED_FIFO)) in a process with
 1794 scheduling policy SCHED_RR and system minimum priority *L* (the value returned by
 1795 *sched_get_priority_min*(SCHED_RR)) shall be subject to timeslicing and to preemption by
 1796 any thread with an effective priority higher than *L*.

1797 Associated with each policy is a priority range. Each policy definition shall specify the minimum

1798 priority range for that policy. The priority ranges for each policy may but need not overlap the
1799 priority ranges of other policies.

1800 A conforming implementation shall select the thread that is defined as being at the head of the
1801 highest priority non-empty thread list to become a running thread, regardless of its associated
1802 policy. This thread is then removed from its thread list.

1803 Four scheduling policies are specifically required. Other implementation-defined scheduling
1804 policies may be defined. The following symbols are defined in the Base Definitions volume of
1805 IEEE Std 1003.1-200x, <sched.h>:

1806 SCHED_FIFO First in, first out (FIFO) scheduling policy.

1807 SCHED_RR Round robin scheduling policy.

1808 SS SCHED_SPORADIC Sporadic server scheduling policy.

1809 SCHED_OTHER Another scheduling policy.

1810 The values of these symbols shall be distinct.

1811 SCHED_FIFO

1812 Conforming implementations shall include a scheduling policy called the FIFO scheduling
1813 policy.

1814 Threads scheduled under this policy are chosen from a thread list that is ordered by the time its
1815 threads have been on the list without being executed; generally, the head of the list is the thread
1816 that has been on the list the longest time, and the tail is the thread that has been on the list the
1817 shortest time.

1818 Under the SCHED_FIFO policy, the modification of the definitional thread lists is as follows:

- 1819 1. When a running thread becomes a preempted thread, it becomes the head of the thread
1820 list for its priority.
- 1821 2. When a blocked thread becomes a runnable thread, it becomes the tail of the thread list
1822 for its priority.
- 1823 3. When a running thread calls the *sched_setscheduler()* function, the process specified in the
1824 function call is modified to the specified policy and the priority specified by the *param*
1825 argument.
- 1826 4. When a running thread calls the *sched_setparam()* function, the priority of the process
1827 specified in the function call is modified to the priority specified by the *param* argument.
- 1828 5. When a running thread calls the *pthread_setschedparam()* function, the thread specified in
1829 the function call is modified to the specified policy and the priority specified by the *param*
1830 argument.
- 1831 6. When a running thread calls the *pthread_setschedprio()* function, the thread specified in the
1832 function call is modified to the priority specified by the *prio* argument.
- 1833 7. If a thread whose policy or priority has been modified other than by *pthread_setschedprio()*
1834 is a running thread or is runnable, it then becomes the tail of the thread list for its new
1835 priority.
- 1836 8. If a thread whose priority has been modified by *pthread_setschedprio()* is a running thread
1837 or is runnable, the effect on its position in the thread list depends on the direction of the
1838 modification, as follows:

- 1839 a. If the priority is raised, the thread becomes the tail of the thread list.
- 1840 b. If the priority is unchanged, the thread does not change position in the thread list.
- 1841 c. If the priority is lowered, the thread becomes the head of the thread list.

1842 9. When a running thread issues the *sched_yield()* function, the thread becomes the tail of
1843 the thread list for its priority.

1844 10. At no other time is the position of a thread with this scheduling policy within the thread
1845 lists affected.

1846 For this policy, valid priorities shall be within the range returned by the *sched_get_priority_max()*
1847 and *sched_get_priority_min()* functions when SCHED_FIFO is provided as the parameter.
1848 Conforming implementations shall provide a priority range of at least 32 priorities for this
1849 policy.

1850 SCHED_RR

1851 Conforming implementations shall include a scheduling policy called the “round robin”
1852 scheduling policy. This policy shall be identical to the SCHED_FIFO policy with the additional
1853 condition that when the implementation detects that a running thread has been executing as a
1854 running thread for a time period of the length returned by the *sched_rr_get_interval()* function or
1855 longer, the thread shall become the tail of its thread list and the head of that thread list shall be
1856 removed and made a running thread.

1857 The effect of this policy is to ensure that if there are multiple SCHED_RR threads at the same
1858 priority, one of them does not monopolize the processor. An application should not rely only on
1859 the use of SCHED_RR to ensure application progress among multiple threads if the application
1860 includes threads using the SCHED_FIFO policy at the same or higher priority levels or
1861 SCHED_RR threads at a higher priority level.

1862 A thread under this policy that is preempted and subsequently resumes execution as a running
1863 thread completes the unexpired portion of its round robin interval time period.

1864 For this policy, valid priorities shall be within the range returned by the *sched_get_priority_max()*
1865 and *sched_get_priority_min()* functions when SCHED_RR is provided as the parameter.
1866 Conforming implementations shall provide a priority range of at least 32 priorities for this
1867 policy.

1868 SCHED_SPORADIC

1869 SS | TSP The functionality described in this section shall be provided on implementations that support
1870 the Process Sporadic Server or Thread Sporadic Server options (and the rest of this section is not
1871 further shaded for these options).

1872 If `_POSIX_SPORADIC_SERVER` or `_POSIX_THREAD_SPORADIC_SERVER` is defined, the
1873 implementation shall include a scheduling policy identified by the value SCHED_SPORADIC.

1874 The sporadic server policy is based primarily on two time parameters: the replenishment period
1875 and the available execution capacity. The replenishment period is given by the
1876 *sched_ss_repl_period* member of the **sched_param** structure. The available execution capacity is
1877 initialized to the value given by the *sched_ss_init_budget* member of the same parameter. The
1878 sporadic server policy is identical to the SCHED_FIFO policy with some additional conditions
1879 that cause the thread’s assigned priority to be switched between the values specified by the
1880 *sched_priority* and *sched_ss_low_priority* members of the **sched_param** structure.

1881 The priority assigned to a thread using the sporadic server scheduling policy is determined in
1882 the following manner: if the available execution capacity is greater than zero and the number of

1883 pending replenishment operations is strictly less than *sched_ss_max_repl*, the thread is assigned
 1884 the priority specified by *sched_priority*; otherwise, the assigned priority shall be
 1885 *sched_ss_low_priority*. If the value of *sched_priority* is less than or equal to the value of
 1886 *sched_ss_low_priority*, the results are undefined. When active, the thread shall belong to the
 1887 thread list corresponding to its assigned priority level, according to the mentioned priority
 1888 assignment. The modification of the available execution capacity and, consequently of the
 1889 assigned priority, is done as follows:

- 1890 1. When the thread at the head of the *sched_priority* list becomes a running thread, its
 1891 execution time shall be limited to at most its available execution capacity, plus the
 1892 resolution of the execution time clock used for this scheduling policy. This resolution shall
 1893 be implementation-defined.
- 1894 2. Each time the thread is inserted at the tail of the list associated with *sched_priority*—
 1895 because as a blocked thread it became runnable with priority *sched_priority* or because a
 1896 replenishment operation was performed—the time at which this operation is done is
 1897 posted as the *activation_time*.
- 1898 3. When the running thread with assigned priority equal to *sched_priority* becomes a
 1899 preempted thread, it becomes the head of the thread list for its priority, and the execution
 1900 time consumed is subtracted from the available execution capacity. If the available
 1901 execution capacity would become negative by this operation, it shall be set to zero.
- 1902 4. When the running thread with assigned priority equal to *sched_priority* becomes a blocked
 1903 thread, the execution time consumed is subtracted from the available execution capacity,
 1904 and a replenishment operation is scheduled, as described in 6 and 7. If the available
 1905 execution capacity would become negative by this operation, it shall be set to zero.
- 1906 5. When the running thread with assigned priority equal to *sched_priority* reaches the limit
 1907 imposed on its execution time, it becomes the tail of the thread list for
 1908 *sched_ss_low_priority*, the execution time consumed is subtracted from the available
 1909 execution capacity (which becomes zero), and a replenishment operation is scheduled, as
 1910 described in 6 and 7.
- 1911 6. Each time a replenishment operation is scheduled, the amount of execution capacity to be
 1912 replenished, *replenish_amount*, is set equal to the execution time consumed by the thread
 1913 since the *activation_time*. The replenishment is scheduled to occur at *activation_time* plus
 1914 *sched_ss_repl_period*. If the scheduled time obtained is before the current time, the
 1915 replenishment operation is carried out immediately. Several replenishment operations
 1916 may be pending at the same time, each of which will be serviced at its respective
 1917 scheduled time. With the above rules, the number of replenishment operations
 1918 simultaneously pending for a given thread that is scheduled under the sporadic server
 1919 policy shall not be greater than *sched_ss_max_repl*.
- 1920 7. A replenishment operation consists of adding the corresponding *replenish_amount* to the
 1921 available execution capacity at the scheduled time. If, as a consequence of this operation,
 1922 the execution capacity would become larger than *sched_ss_initial_budget*, it shall be
 1923 rounded down to a value equal to *sched_ss_initial_budget*. Additionally, if the thread was
 1924 runnable or running, and had assigned priority equal to *sched_ss_low_priority*, then it
 1925 becomes the tail of the thread list for *sched_priority*.

1926 Execution time is defined in Base Definitions volume of IEEE Std 1003.1-200x, Section 3.155,
 1927 Execution Time.

1928 For this policy, changing the value of a CPU-time clock via *clock_settime()* shall have no effect on
 1929 its behavior.

1930 For this policy, valid priorities shall be within the range returned by the *sched_get_priority_min()*

1931 and *sched_get_priority_max()* functions when SCHED_SPORADIC is provided as the parameter.
 1932 Conforming implementations shall provide a priority range of at least 32 distinct priorities for
 1933 this policy.

1934 If the scheduling policy of the target process is either SCHED_FIFO or SCHED_RR, the
 1935 *sched_ss_low_priority*, *sched_ss_repl_period*, and *sched_ss_init* budget members of the *param*
 1936 argument shall have no effect on the scheduling behavior. If the scheduling policy of this process
 1937 is not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC, the effects of these members are
 1938 implementation-defined; this case includes the SCHED_OTHER policy.

1939 SCHED_OTHER

1940 Conforming implementations shall include one scheduling policy identified as SCHED_OTHER
 1941 (which may execute identically with either the FIFO or round robin scheduling policy). The
 1942 effect of scheduling threads with the SCHED_OTHER policy in a system in which other threads
 1943 SS are executing under SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC is implementation-
 1944 defined.

1945 This policy is defined to allow strictly conforming applications to be able to indicate in a
 1946 portable manner that they no longer need a realtime scheduling policy.

1947 For threads executing under this policy, the implementation shall use only priorities within the
 1948 range returned by the *sched_get_priority_max()* and *sched_get_priority_min()* functions when
 1949 SCHED_OTHER is provided as the parameter.

1950 2.8.5 Clocks and Timers

1951 The `<time.h>` header defines the types and manifest constants used by the timing facility.

1952 Time Value Specification Structures

1953 Many of the timing facility functions accept or return time value specifications. A time value
 1954 structure **timespec** specifies a single time value and includes at least the following members:

Member Type	Member Name	Description
time_t	<i>tv_sec</i>	Seconds.
long	<i>tv_nsec</i>	Nanoseconds.

1955 The *tv_nsec* member is only valid if greater than or equal to zero, and less than the number of
 1956 nanoseconds in a second (1 000 million). The time interval described by this structure is (*tv_sec* *
 1957 10^9 + *tv_nsec*) nanoseconds.

1961 A time value structure **itimerspec** specifies an initial timer value and a repetition interval for use
 1962 by the per-process timer functions. This structure includes at least the following members:

Member Type	Member Name	Description
struct timespec	<i>it_interval</i>	Timer period.
struct timespec	<i>it_value</i>	Timer expiration.

1966 If the value described by *it_value* is non-zero, it indicates the time to or time of the next timer
 1967 expiration (for relative and absolute timer values, respectively). If the value described by *it_value*
 1968 is zero, the timer shall be disarmed.

1969 If the value described by *it_interval* is non-zero, it specifies an interval which shall be used in
 1970 reloading the timer when it expires; that is, a periodic timer is specified. If the value described
 1971 by *it_interval* is zero, the timer is disarmed after its next expiration; that is, a one-shot timer is
 1972 specified.

1973		Timer Event Notification Control Block
1974		Per-process timers may be created that notify the process of timer expirations by queuing a
1975		realtime extended signal. The sigevent structure, defined in the Base Definitions volume of
1976		IEEE Std 1003.1-200x, <signal.h> , is used in creating such a timer. The sigevent structure
1977		contains the signal number and an application-specific data value which shall be used when
1978		notifying the calling process of timer expiration events.
1979		Manifest Constants
1980		The following constants are defined in the Base Definitions volume of IEEE Std 1003.1-200x,
1981		<time.h> :
1982		CLOCK_REALTIME The identifier for the system-wide realtime clock.
1983		TIMER_ABSTIME Flag indicating time is absolute with respect to the clock associated
1984		with a timer.
1985	MON	CLOCK_MONOTONIC The identifier for the system-wide monotonic clock, which is defined
1986		as a clock whose value cannot be set via <i>clock_settime()</i> and which
1987		cannot have backward clock jumps. The maximum possible clock
1988		jump is implementation-defined.
1989	MON	The maximum allowable resolution for CLOCK_REALTIME and CLOCK_MONOTONIC clocks
1990		and all time services based on these clocks is represented by {_POSIX_CLOCKRES_MIN} and
1991		shall be defined as 20 ms (1/50 of a second). Implementations may support smaller values of
1992		resolution for these clocks to provide finer granularity time bases. The actual resolution
1993		supported by an implementation for a specific clock is obtained using the <i>clock_getres()</i> function.
1994		If the actual resolution supported for a time service based on one of these clocks differs from the
1995		resolution supported for that clock, the implementation shall document this difference.
1996	MON	The minimum allowable maximum value for CLOCK_REALTIME and CLOCK_MONOTONIC
1997		clocks and all absolute time services based on them is the same as that defined by the ISO C
1998		standard for the time_t type. If the maximum value supported by a time service based on one of
1999		these clocks differs from the maximum value supported by that clock, the implementation shall
2000		document this difference.
2001		Execution Time Monitoring
2002	CPT	If _POSIX_CPUTIME is defined, process CPU-time clocks shall be supported in addition to the
2003		clocks described in Manifest Constants (on page 49).
2004	TCT	If _POSIX_THREAD_CPUTIME is defined, thread CPU-time clocks shall be supported.
2005	CPT TCT	CPU-time clocks measure execution or CPU time, which is defined in the Base Definitions
2006		volume of IEEE Std 1003.1-200x, Section 3.117, CPU Time (Execution Time). The mechanism
2007		used to measure execution time is described in the Base Definitions volume of
2008		IEEE Std 1003.1-200x, Section 4.9, Measurement of Execution Time.
2009	CPT	If _POSIX_CPUTIME is defined, the following constant of the type clockid_t is defined in
2010		<time.h> :
2011		CLOCK_PROCESS_CPUTIME_ID
2012		When this value of the type clockid_t is used in a <i>clock()</i> or <i>timer*()</i> function call, it is
2013		interpreted as the identifier of the CPU-time clock associated with the process making the
2014		function call.

2015 TCT If `_POSIX_THREAD_CPUTIME` is defined, the following constant of the type `clockid_t` is
 2016 defined in `<time.h>`:

2017 `CLOCK_THREAD_CPUTIME_ID`

2018 When this value of the type `clockid_t` is used in a `clock()` or `timer*()` function call, it is
 2019 interpreted as the identifier of the CPU-time clock associated with the thread making the
 2020 function call.

2021 2.9 Threads

2022 This section defines functionality to support multiple flows of control, called “threads”, within a
 2023 process. For the definition of threads, see the Base Definitions volume of IEEE Std 1003.1-200x,
 2024 Section 3.393, Thread.

2025 The specific functional areas covered by threads and their scope include:

- 2026 • Thread management: the creation, control, and termination of multiple flows of control in
 2027 the same process under the assumption of a common shared address space
- 2028 • Synchronization primitives optimized for tightly coupled operation of multiple control
 2029 flows in a common, shared address space

2030 2.9.1 Thread-Safety

2031 All functions defined by this volume of IEEE Std 1003.1-200x shall be thread-safe, except that the
 2032 following functions¹ need not be thread-safe.

2033 <i>asctime()</i>	<i>ftw()</i>	<i>getserbyport()</i>	<i>putc_unlocked()</i>
2034 <i>basename()</i>	<i>getc_unlocked()</i>	<i>getservent()</i>	<i>putchar_unlocked()</i>
2035 <i>catgets()</i>	<i>getchar_unlocked()</i>	<i>getutxent()</i>	<i>putenv()</i>
2036 <i>crypt()</i>	<i>getdate()</i>	<i>getutxid()</i>	<i>pututxline()</i>
2037 <i>ctime()</i>	<i>getenv()</i>	<i>getutxline()</i>	<i>rand()</i>
2038 <i>dbm_clearerr()</i>	<i>getgrent()</i>	<i>gmtime()</i>	<i>readdir()</i>
2039 <i>dbm_close()</i>	<i>getgrgid()</i>	<i>hcreate()</i>	<i>setenv()</i>
2040 <i>dbm_delete()</i>	<i>getgrnam()</i>	<i>hdestroy()</i>	<i>setgrent()</i>
2041 <i>dbm_error()</i>	<i>gethostent()</i>	<i>hsearch()</i>	<i>setkey()</i>
2042 <i>dbm_fetch()</i>	<i>getlogin()</i>	<i>inet_ntoa()</i>	<i>setpwent()</i>
2043 <i>dbm_firstkey()</i>	<i>getnetbyaddr()</i>	<i>l64a()</i>	<i>setutxent()</i>
2044 <i>dbm_nextkey()</i>	<i>getnetbyname()</i>	<i>lgamma()</i>	<i>strerror()</i>
2045 <i>dbm_open()</i>	<i>getnetent()</i>	<i>lgammaf()</i>	<i>strsignal()</i>
2046 <i>dbm_store()</i>	<i>getopt()</i>	<i>lgammal()</i>	<i>strtok()</i>
2047 <i>dirname()</i>	<i>getprotobyname()</i>	<i>localeconv()</i>	<i>system()</i>
2048 <i>dllerror()</i>	<i>getprotobynumber()</i>	<i>localtime()</i>	<i>ttyname()</i>
2049 <i>drand48()</i>	<i>getprotoent()</i>	<i>lrand48()</i>	<i>unsetenv()</i>
2050 <i>encrypt()</i>	<i>getpwent()</i>	<i>mrnd48()</i>	<i>wcstombs()</i>
2051 <i>endgrent()</i>	<i>getpwnam()</i>	<i>nftw()</i>	<i>wctomb()</i>
2052 <i>endpwent()</i>	<i>getpwuid()</i>	<i>nl_langinfo()</i>	
2053 <i>endutxent()</i>	<i>getserbyname()</i>	<i>ptsname()</i>	

2054 The *ctermid()* and *tmpnam()* functions need not be thread-safe if passed a NULL argument. The
 2055 *wcrtomb()* and *wcsrombs()* functions need not be thread-safe if passed a NULL *ps* argument.

2056 1. The functions in the table are not shaded to denote applicable options. Individual reference pages should be consulted.

2057 Implementations shall provide internal synchronization as necessary in order to satisfy this
2058 requirement.

2059 2.9.2 Thread IDs

2060 Although implementations may have thread IDs that are unique in a system, applications
2061 should only assume that thread IDs are usable and unique within a single process. The effect of
2062 calling any of the functions defined in this volume of IEEE Std 1003.1-200x and passing as an
2063 argument the thread ID of a thread from another process is unspecified. A conforming
2064 implementation is free to reuse a thread ID after the thread terminates if it was created with the
2065 *detachstate* attribute set to *PTHREAD_CREATE_DETACHED* or if *pthread_detach()* or
2066 *pthread_join()* has been called for that thread. If a thread is detached, its thread ID is invalid for
2067 use as an argument in a call to *pthread_detach()* or *pthread_join()*.

2068 2.9.3 Thread Mutexes

2069 A thread that has blocked shall not prevent any unblocked thread that is eligible to use the same
2070 processing resources from eventually making forward progress in its execution. Eligibility for
2071 processing resources is determined by the scheduling policy.

2072 A thread shall become the owner of a mutex, *m*, when one of the following occurs:

- 2073 • It returns successfully from *pthread_mutex_lock()* with *m* as the *mutex* argument.
- 2074 • It returns successfully from *pthread_mutex_trylock()* with *m* as the *mutex* argument.
- 2075 • It returns successfully from *pthread_mutex_timedlock()* with *m* as the *mutex* argument.
- 2076 • It returns (successfully or not) from *pthread_cond_wait()* with *m* as the *mutex* argument
2077 (except as explicitly indicated otherwise for certain errors).
- 2078 • It returns (successfully or not) from *pthread_cond_timedwait()* with *m* as the *mutex*
2079 argument (except as explicitly indicated otherwise for certain errors).

2080 The thread shall remain the owner of *m* until one of the following occurs:

- 2081 • It executes *pthread_mutex_unlock()* with *m* as the *mutex* argument
- 2082 • It blocks in a call to *pthread_cond_wait()* with *m* as the *mutex* argument.
- 2083 • It blocks in a call to *pthread_cond_timedwait()* with *m* as the *mutex* argument.

2084 The implementation shall behave as if at all times there is at most one owner of any mutex.

2085 A thread that becomes the owner of a mutex is said to have “acquired” the mutex and the mutex
2086 is said to have become “locked”; when a thread gives up ownership of a mutex it is said to have
2087 “released” the mutex and the mutex is said to have become “unlocked”.

2088 A problem can occur if a process terminates while one of its threads holds a mutex lock.
2089 Depending on the mutex type, it might be possible for another thread to unlock the mutex and
2090 recover the state of the mutex. However, it is difficult to perform this recovery reliably.

2091 Robust mutexes provide a means to enable the implementation to notify other threads in the
2092 event of a process terminating while one of its threads holds a mutex lock. The next thread that
2093 acquires the mutex is notified about the termination by the return value [EOWNERDEAD] from
2094 the locking function. The notified thread can then attempt to recover the state protected by the
2095 mutex, and if successful mark the state protected by the mutex as consistent by a call to
2096 *pthread_mutex_consistent()*. If the notified thread is unable to recover the state, it can declare the
2097 state as not recoverable by a call to *pthread_mutex_unlock()* without a prior call to
2098 *pthread_mutex_consistent()*.

2099 Whether or not the state protected by a mutex can be recovered is dependent solely on the

2100 application using robust mutexes. The robust mutex support provided in the implementation
 2101 provides notification only that a mutex owner has terminated while holding a lock, or that the
 2102 state of the mutex is not recoverable.

2103 2.9.4 Thread Scheduling

2104 TPS The functionality described in this section shall be provided on implementations that support
 2105 the Thread Execution Scheduling option (and the rest of this section is not further shaded for
 2106 this option).

2107 Thread Scheduling Attributes

2108 In support of the scheduling function, threads have attributes which are accessed through the
 2109 **pthread_attr_t** thread creation attributes object.

2110 The *contentionscope* attribute defines the scheduling contention scope of the thread to be either
 2111 PTHREAD_SCOPE_PROCESS or PTHREAD_SCOPE_SYSTEM.

2112 The *inheritsched* attribute specifies whether a newly created thread is to inherit the scheduling
 2113 attributes of the creating thread or to have its scheduling values set according to the other
 2114 scheduling attributes in the **pthread_attr_t** object.

2115 The *schedpolicy* attribute defines the scheduling policy for the thread. The *schedparam* attribute
 2116 defines the scheduling parameters for the thread. The interaction of threads having different
 2117 policies within a process is described as part of the definition of those policies.

2118 If the Thread Execution Scheduling option is defined, and the *schedpolicy* attribute specifies one
 2119 of the priority-based policies defined under this option, the *schedparam* attribute contains the
 2120 scheduling priority of the thread. A conforming implementation ensures that the priority value
 2121 in *schedparam* is in the range associated with the scheduling policy when the thread attributes
 2122 object is used to create a thread, or when the scheduling attributes of a thread are dynamically
 2123 modified. The meaning of the priority value in *schedparam* is the same as that of *priority*.

2124 TSP If `_POSIX_THREAD_SPORADIC_SERVER` is defined, the *schedparam* attribute supports four
 2125 new members that are used for the sporadic server scheduling policy. These members are
 2126 *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*, and *sched_ss_max_repl*. The
 2127 meaning of these attributes is the same as in the definitions that appear under [Section 2.8.4](#) (on
 2128 page 44).

2129 When a process is created, its single thread has a scheduling policy and associated attributes
 2130 equal to the policy and attributes of the process. The default scheduling contention scope value
 2131 is implementation-defined. The default values of other scheduling attributes are
 2132 implementation-defined.

2133 Thread Scheduling Contention Scope

2134 The scheduling contention scope of a thread defines the set of threads with which the thread
 2135 competes for use of the processing resources. The scheduling operation selects at most one
 2136 thread to execute on each processor at any point in time and the thread's scheduling attributes
 2137 (for example, *priority*), whether under process scheduling contention scope or system scheduling
 2138 contention scope, are the parameters used to determine the scheduling decision.

2139 The scheduling contention scope, in the context of scheduling a mixed scope environment,
 2140 affects threads as follows:

- 2141 • A thread created with PTHREAD_SCOPE_SYSTEM scheduling contention scope contends
 2142 for resources with all other threads in the same scheduling allocation domain relative to
 2143 their system scheduling attributes. The system scheduling attributes of a thread created
 2144 with PTHREAD_SCOPE_SYSTEM scheduling contention scope are the scheduling

2145 attributes with which the thread was created. The system scheduling attributes of a thread
 2146 created with `PTHREAD_SCOPE_PROCESS` scheduling contention scope are the
 2147 implementation-defined mapping into system attribute space of the scheduling attributes
 2148 with which the thread was created.

- 2149 • Threads created with `PTHREAD_SCOPE_PROCESS` scheduling contention scope contend
 2150 directly with other threads within their process that were created with
 2151 `PTHREAD_SCOPE_PROCESS` scheduling contention scope. The contention is resolved
 2152 based on the threads' scheduling attributes and policies. It is unspecified how such threads
 2153 are scheduled relative to threads in other processes or threads with
 2154 `PTHREAD_SCOPE_SYSTEM` scheduling contention scope.
- 2155 • Conforming implementations shall support the `PTHREAD_SCOPE_PROCESS` scheduling
 2156 contention scope, the `PTHREAD_SCOPE_SYSTEM` scheduling contention scope, or both.

2157 Scheduling Allocation Domain

2158 Implementations shall support scheduling allocation domains containing one or more
 2159 processors. It should be noted that the presence of multiple processors does not automatically
 2160 indicate a scheduling allocation domain size greater than one. Conforming implementations on
 2161 multi-processors may map all or any subset of the CPUs to one or multiple scheduling allocation
 2162 domains, and could define these scheduling allocation domains on a per-thread, per-process, or
 2163 per-system basis, depending on the types of applications intended to be supported by the
 2164 implementation. The scheduling allocation domain is independent of scheduling contention
 2165 scope, as the scheduling contention scope merely defines the set of threads with which a thread
 2166 contends for processor resources, while scheduling allocation domain defines the set of
 2167 processors for which it contends. The semantics of how this contention is resolved among
 2168 threads for processors is determined by the scheduling policies of the threads.

2169 The choice of scheduling allocation domain size and the level of application control over
 2170 scheduling allocation domains is implementation-defined. Conforming implementations may
 2171 change the size of scheduling allocation domains and the binding of threads to scheduling
 2172 allocation domains at any time.

2173 For application threads with scheduling allocation domains of size equal to one, the scheduling
 2174 rules defined for `SCHED_FIFO` and `SCHED_RR` shall be used; see [Scheduling Policies](#) (on page
 2175 44). All threads with system scheduling contention scope, regardless of the processes in which
 2176 they reside, compete for the processor according to their priorities. Threads with process
 2177 scheduling contention scope compete only with other threads with process scheduling
 2178 contention scope within their process.

2179 For application threads with scheduling allocation domains of size greater than one, the rules
 2180 TSP defined for `SCHED_FIFO`, `SCHED_RR`, and `SCHED_SPORADIC` shall be used in an
 2181 implementation-defined manner. Each thread with system scheduling contention scope
 2182 competes for the processors in its scheduling allocation domain in an implementation-defined
 2183 manner according to its priority. Threads with process scheduling contention scope are
 2184 scheduled relative to other threads within the same scheduling contention scope in the process.

2185 TSP If `_POSIX_THREAD_SPORADIC_SERVER` is defined, the rules defined for `SCHED_SPORADIC`
 2186 in [Scheduling Policies](#) shall be used in an implementation-defined manner for application
 2187 threads whose scheduling allocation domain size is greater than one.

2188 **Scheduling Documentation**

2189 If `_POSIX_PRIORITY_SCHEDULING` is defined, then any scheduling policies beyond
 2190 TSP `SCHED_OTHER`, `SCHED_FIFO`, `SCHED_RR`, and `SCHED_SPORADIC`, as well as the effects of
 2191 the scheduling policies indicated by these other values, and the attributes required in order to
 2192 support such a policy, are implementation-defined. Furthermore, the implementation shall
 2193 document the effect of all processor scheduling allocation domain values supported for these
 2194 policies.

2195 **2.9.5 Thread Cancellation**

2196 The thread cancellation mechanism allows a thread to terminate the execution of any other
 2197 thread in the process in a controlled manner. The target thread (that is, the one that is being
 2198 canceled) is allowed to hold cancellation requests pending in a number of ways and to perform
 2199 application-specific cleanup processing when the notice of cancellation is acted upon.

2200 Cancellation is controlled by the cancellation control functions. Each thread maintains its own
 2201 cancelability state. Cancellation may only occur at cancellation points or when the thread is
 2202 asynchronously cancelable.

2203 The thread cancellation mechanism described in this section depends upon programs having set
 2204 *deferred* cancelability state, which is specified as the default. Applications shall also carefully
 2205 follow static lexical scoping rules in their execution behavior. For example, use of *setjmp()*,
 2206 *return*, *goto*, and so on, to leave user-defined cancellation scopes without doing the necessary
 2207 scope pop operation results in undefined behavior.

2208 Use of asynchronous cancelability while holding resources which potentially need to be released
 2209 may result in resource loss. Similarly, cancellation scopes may only be safely manipulated
 2210 (pushed and popped) when the thread is in the *deferred* or *disabled* cancelability states.

2211 **2.9.5.1 Cancelability States**

2212 The cancelability state of a thread determines the action taken upon receipt of a cancellation
 2213 request. The thread may control cancellation in a number of ways.

2214 Each thread maintains its own cancelability state, which may be encoded in two bits:

- 2215 1. Cancelability-Enable: When cancelability is `PTHREAD_CANCEL_DISABLE` (as defined
 2216 in the Base Definitions volume of IEEE Std 1003.1-200x, `<pthread.h>`), cancellation
 2217 requests against the target thread are held pending. By default, cancelability is set to
 2218 `PTHREAD_CANCEL_ENABLE` (as defined in `<pthread.h>`).
- 2219 2. Cancelability Type: When cancelability is enabled and the cancelability type is
 2220 `PTHREAD_CANCEL_ASYNCHRONOUS` (as defined in `<pthread.h>`), new or pending
 2221 cancellation requests may be acted upon at any time. When cancelability is enabled and
 2222 the cancelability type is `PTHREAD_CANCEL_DEFERRED` (as defined in `<pthread.h>`),
 2223 cancellation requests are held pending until a cancellation point (see below) is reached. If
 2224 cancelability is disabled, the setting of the cancelability type has no immediate effect as all
 2225 cancellation requests are held pending; however, once cancelability is enabled again the
 2226 new type is in effect. The cancelability type is `PTHREAD_CANCEL_DEFERRED` in all
 2227 newly created threads including the thread in which *main()* was first invoked.

2228 2.9.5.2 Cancellation Points

2229 Cancellation points shall occur when a thread is executing the following functions:

2230	<i>accept()</i>	<i>nanosleep()</i>	<i>select()</i>
2231	<i>aio_suspend()</i>	<i>open()</i>	<i>sem_timedwait()</i>
2232	<i>clock_nanosleep()</i>	<i>openat()</i>	<i>sem_wait()</i>
2233	<i>close()</i>	<i>pause()</i>	<i>send()</i>
2234	<i>connect()</i>	<i>poll()</i>	<i>sendmsg()</i>
2235	<i>creat()</i>	<i>pread()</i>	<i>sendto()</i>
2236	<i>fcntl()†</i>	<i>pselect()</i>	<i>sigsuspend()</i>
2237	<i>fdatasync()</i>	<i>pthread_cond_timedwait()</i>	<i>sigtimedwait()</i>
2238	<i>fsync()</i>	<i>pthread_cond_wait()</i>	<i>sigwait()</i>
2239	<i>getmsg()</i>	<i>pthread_join()</i>	<i>sigwaitinfo()</i>
2240	<i>getpmsg()</i>	<i>pthread_testcancel()</i>	<i>sleep()</i>
2241	<i>lockf()</i>	<i>putmsg()</i>	<i>system()</i>
2242	<i>mq_receive()</i>	<i>putpmsg()</i>	<i>tcdrain()</i>
2243	<i>mq_send()</i>	<i>pwrite()</i>	<i>wait()</i>
2244	<i>mq_timedreceive()</i>	<i>read()</i>	<i>waitid()</i>
2245	<i>mq_timedsend()</i>	<i>readv()</i>	<i>waitpid()</i>
2246	<i>msgrcv()</i>	<i>recv()</i>	<i>write()</i>
2247	<i>msgsnd()</i>	<i>recvfrom()</i>	<i>writen()</i>
2248	<i>msync()</i>	<i>recvmsg()</i>	

2249 † When the *cmd* argument is F_SETLKW.

2250 A cancellation point may also occur when a thread is executing the following functions:

2251	<i>access()</i>	<i>fread()</i>	<i>getservent()</i>
2252	<i>asctime()</i>	<i>freopen()</i>	<i>getutxent()</i>
2253	<i>asctime_r()</i>	<i>fscanf()</i>	<i>getutxid()</i>
2254	<i>catclose()</i>	<i>fseek()</i>	<i>getutxline()</i>
2255	<i>catgets()</i>	<i>fseeko()</i>	<i>getwc()</i>
2256	<i>catopen()</i>	<i>fsetpos()</i>	<i>getwchar()</i>
2257	<i>chmod()</i>	<i>fstat()</i>	<i>glob()</i>
2258	<i>chown()</i>	<i>fstatat()</i>	<i>iconv_close()</i>
2259	<i>closedir()</i>	<i>ftell()</i>	<i>iconv_open()</i>
2260	<i>closelog()</i>	<i>ftello()</i>	<i>ioctl()</i>
2261	<i>ctermid()</i>	<i>ftw()</i>	<i>link()</i>
2262	<i>ctime()</i>	<i>futimesat()</i>	<i>linkat()</i>
2263	<i>ctime_r()</i>	<i>fwprintf()</i>	<i>lio_listio()</i>
2264	<i>dbm_close()</i>	<i>fwrite()</i>	<i>localtime()</i>
2265	<i>dbm_delete()</i>	<i>fwscanf()</i>	<i>localtime_r()</i>
2266	<i>dbm_fetch()</i>	<i>getaddrinfo()</i>	<i>lseek()</i>
2267	<i>dbm_nextkey()</i>	<i>getc()</i>	<i>lstat()</i>
2268	<i>dbm_open()</i>	<i>getc_unlocked()</i>	<i>mkdir()</i>
2269	<i>dbm_store()</i>	<i>getchar()</i>	<i>mkdirat()</i>
2270	<i>dlclose()</i>	<i>getchar_unlocked()</i>	<i>mkdtemp()</i>
2271	<i>dlopen()</i>	<i>getcwd()</i>	<i>mkfifo()</i>
2272	<i>dprintf()</i>	<i>getdate()</i>	<i>mkfifoat()</i>
2273	<i>endgrent()</i>	<i>getdelim()</i>	<i>mknod()</i>
2274	<i>endhostent()</i>	<i>getgrent()</i>	<i>mknodat()</i>
2275	<i>endnetent()</i>	<i>getgrgid()</i>	<i>mkstemp()</i>
2276	<i>endprotoent()</i>	<i>getgrgid_r()</i>	<i>mktime()</i>
2277	<i>endpwent()</i>	<i>getgrnam()</i>	<i>nftw()</i>
2278	<i>endservent()</i>	<i>getgrnam_r()</i>	<i>opendir()</i>
2279	<i>endutxent()</i>	<i>gethostent()</i>	<i>openlog()</i>
2280	<i>faccessat()</i>	<i>gethostid()</i>	<i>pathconf()</i>
2281	<i>fchmod()</i>	<i>gethostname()</i>	<i>pclose()</i>
2282	<i>fchmodat()</i>	<i>getline()</i>	<i>perror()</i>
2283	<i>fchown()</i>	<i>getlogin()</i>	<i>popen()</i>
2284	<i>fchownat()</i>	<i>getlogin_r()</i>	<i>posix_fadvise()</i>
2285	<i>fclose()</i>	<i>getnameinfo()</i>	<i>posix_fallocate()</i>
2286	<i>fcntl()†</i>	<i>getnetbyaddr()</i>	<i>posix_madvise()</i>
2287	<i>fflush()</i>	<i>getnetbyname()</i>	<i>posix_openpt()</i>
2288	<i>fgetc()</i>	<i>getnetent()</i>	<i>posix_spawn()</i>
2289	<i>fgetpos()</i>	<i>getopt()††</i>	<i>posix_spawnnp()</i>
2290	<i>fgets()</i>	<i>getprotobyname()</i>	<i>posix_trace_clear()</i>
2291	<i>fgetwc()</i>	<i>getprotobyname_r()</i>	<i>posix_trace_close()</i>
2292	<i>fgetws()</i>	<i>getprotoent()</i>	<i>posix_trace_create()</i>
2293	<i>fntmsg()</i>	<i>getpwent()</i>	<i>posix_trace_create_withlog()</i>
2294	<i>fopen()</i>	<i>getpwnam()</i>	<i>posix_trace_eventtypelist_getnext_id()</i>
2295	<i>fpathconf()</i>	<i>getpwnam_r()</i>	<i>posix_trace_eventtypelist_rewind()</i>
2296	<i>fprintf()</i>	<i>getpwuid()</i>	<i>posix_trace_flush()</i>
2297	<i>fputc()</i>	<i>getpwuid_r()</i>	<i>posix_trace_get_attr()</i>
2298	<i>fputs()</i>	<i>gets()</i>	<i>posix_trace_get_filter()</i>
2299	<i>fputwc()</i>	<i>getserobyname()</i>	<i>posix_trace_get_status()</i>
2300	<i>fputws()</i>	<i>getserobyport()</i>	<i>posix_trace_getnext_event()</i>

2301	<i>posix_trace_open()</i>	<i>readlink()</i>	<i>symlink()</i>
2302	<i>posix_trace_rewind()</i>	<i>readlinkat()</i>	<i>symlinkat()</i>
2303	<i>posix_trace_set_filter()</i>	<i>remove()</i>	<i>sync()</i>
2304	<i>posix_trace_shutdown()</i>	<i>rename()</i>	<i>syslog()</i>
2305	<i>posix_trace_timedgetnext_event()</i>	<i>renameat()</i>	<i>tmpfile()</i>
2306	<i>posix_typed_mem_open()</i>	<i>rewind()</i>	<i>tmpnam()</i>
2307	<i>printf()</i>	<i>rewinddir()</i>	<i>ttyname()</i>
2308	<i>psiginfo()</i>	<i>scandir()</i>	<i>ttyname_r()</i>
2309	<i>psignal()</i>	<i>scanf()</i>	<i>tzset()</i>
2310	<i>pthread_rwlock_rdlock()</i>	<i>seekdir()</i>	<i>ungetc()</i>
2311	<i>pthread_rwlock_timedrdlock()</i>	<i>semop()</i>	<i>ungetwc()</i>
2312	<i>pthread_rwlock_timedwrlock()</i>	<i>setgrent()</i>	<i>unlink()</i>
2313	<i>pthread_rwlock_wrlock()</i>	<i>sethostent()</i>	<i>unlinkat()</i>
2314	<i>putc()</i>	<i>setnetent()</i>	<i>utime()</i>
2315	<i>putc_unlocked()</i>	<i>setprotoent()</i>	<i>utimes()</i>
2316	<i>putchar()</i>	<i>setpwent()</i>	<i>vfprintf()</i>
2317	<i>putchar_unlocked()</i>	<i>setservent()</i>	<i>vfwprintf()</i>
2318	<i>puts()</i>	<i>setutxent()</i>	<i>vprintf()</i>
2319	<i>pututxline()</i>	<i>sigpause()</i>	<i>vwprintf()</i>
2320	<i>putwc()</i>	<i>stat()</i>	<i>wcsftime()</i>
2321	<i>putwchar()</i>	<i>strerror()</i>	<i>wordexp()</i>
2322	<i>readdir()</i>	<i>strerror_r()</i>	<i>wprintf()</i>
2323	<i>readdir_r()</i>	<i>strftime()</i>	<i>wscanf()</i>

2324 An implementation shall not introduce cancellation points into any other functions specified in
2325 this volume of IEEE Std 1003.1-200x.

2326 The side effects of acting upon a cancellation request while suspended during a call of a function
2327 are the same as the side effects that may be seen in a single-threaded program when a call to a
2328 function is interrupted by a signal and the given function returns [EINTR]. Any such side
2329 effects occur before any cancellation cleanup handlers are called.

2330 Whenever a thread has cancelability enabled and a cancellation request has been made with that
2331 thread as the target, and the thread then calls any function that is a cancellation point (such as
2332 *pthread_testcancel()* or *read()*), the cancellation request shall be acted upon before the function
2333 returns. If a thread has cancelability enabled and a cancellation request is made with the thread
2334 as a target while the thread is suspended at a cancellation point, the thread shall be awakened
2335 and the cancellation request shall be acted upon. However, if the thread is suspended at a
2336 cancellation point and the event for which it is waiting occurs before the cancellation request is
2337 acted upon, it is unspecified whether the cancellation request is acted upon or whether the
2338 cancellation request remains pending and the thread resumes normal execution.

2339 2.9.5.3 Thread Cancellation Cleanup Handlers

2340 Each thread maintains a list of cancellation cleanup handlers. The programmer uses the
2341 *pthread_cleanup_push()* and *pthread_cleanup_pop()* functions to place routines on and remove
2342 routines from this list.

2343 When a cancellation request is acted upon, or when a thread calls *pthread_exit()*, the thread first
2344 disables cancellation by setting its cancelability state to `PTHREAD_CANCEL_DISABLE` and its
2345 cancelability type to `PTHREAD_CANCEL_DEFERRED`. The cancelability state shall remain set
2346 to `PTHREAD_CANCEL_DISABLE` until the thread has terminated. The behavior is undefined if

2347 † For any value of the *cmd* argument.

2348 †† If *opterr* is non-zero.

2349 a cancellation cleanup handler or thread-specific data destructor routine changes the
2350 cancelability state to `PTHREAD_CANCEL_ENABLE`.

2351 The routines in the thread's list of cancellation cleanup handlers are invoked one by one in LIFO
2352 sequence; that is, the last routine pushed onto the list (Last In) is the first to be invoked (First
2353 Out). When the cancellation cleanup handler for a scope is invoked, the storage for that scope
2354 remains valid. If the last cancellation cleanup handler returns, thread-specific data destructors (if
2355 any) associated with thread-specific data keys for which the thread has non-NULL values will
2356 be run, in unspecified order, as described for `pthread_key_create()`.

2357 After all cancellation cleanup handlers and thread-specific data destructors have returned,
2358 thread execution is terminated. If the thread has terminated because of a call to `pthread_exit()`,
2359 the `value_ptr` argument is made available to any threads joining with the target. If the thread has
2360 terminated by acting on a cancellation request, a status of `PTHREAD_CANCELED` is made
2361 available to any threads joining with the target. The symbolic constant `PTHREAD_CANCELED`
2362 expands to a constant expression of type `(void *)` whose value matches no pointer to an object in
2363 memory nor the value `NULL`.

2364 A side effect of acting upon a cancellation request while in a condition variable wait is that the
2365 mutex is re-acquired before calling the first cancellation cleanup handler. In addition, the thread
2366 is no longer considered to be waiting for the condition and the thread shall not have consumed
2367 any pending condition signals on the condition.

2368 A cancellation cleanup handler cannot exit via `longjmp()` or `siglongjmp()`.

2369 2.9.5.4 Async-Cancel Safety

2370 The `pthread_cancel()`, `pthread_setcancelstate()`, and `pthread_setcanceltype()` functions are defined to
2371 be async-cancel safe.

2372 No other functions in this volume of IEEE Std 1003.1-200x are required to be async-cancel-safe.

2373 2.9.6 Thread Read-Write Locks

2374 Multiple readers, single writer (read-write) locks allow many threads to have simultaneous
2375 read-only access to data while allowing only one thread to have exclusive write access at any
2376 given time. They are typically used to protect data that is read more frequently than it is
2377 changed.

2378 One or more readers acquire read access to the resource by performing a read lock operation on
2379 the associated read-write lock. A writer acquires exclusive write access by performing a write
2380 lock operation. Basically, all readers exclude any writers and a writer excludes all readers and
2381 any other writers.

2382 A thread that has blocked on a read-write lock (for example, has not yet returned from a
2383 `pthread_rwlock_rdlock()` or `pthread_rwlock_wrlock()` call) shall not prevent any unblocked thread
2384 that is eligible to use the same processing resources from eventually making forward progress in
2385 its execution. Eligibility for processing resources shall be determined by the scheduling policy.

2386 Read-write locks can be used to synchronize threads in the current process and other processes if
2387 they are allocated in memory that is writable and shared among the cooperating processes and
2388 have been initialized for this behavior.

2389 2.9.7 Thread Interactions with Regular File Operations

2390 All of the following functions shall be atomic with respect to each other in the effects specified in
2391 IEEE Std 1003.1-200x when they operate on regular files or symbolic links:

2392	<i>chmod()</i>	<i>fchownat()</i>	<i>linkat()</i>	<i>readlink()</i>	<i>unlinkat()</i>
2393	<i>chown()</i>	<i>fcntl()</i>	<i>lseek()</i>	<i>readv()</i>	<i>utime()</i>
2394	<i>close()</i>	<i>fstat()</i>	<i>lstat()</i>	<i>rename()</i>	<i>utimes()</i>
2395	<i>creat()</i>	<i>fstatat()</i>	<i>open()</i>	<i>renameat()</i>	<i>write()</i>
2396	<i>dup2()</i>	<i>ftruncate()</i>	<i>openat()</i>	<i>stat()</i>	<i>writev()</i>
2397	<i>fchmod()</i>	<i>futimesat()</i>	<i>pread()</i>	<i>symlink()</i>	
2398	<i>fchmodat()</i>	<i>lchown()</i>	<i>read()</i>	<i>truncate()</i>	
2399	<i>fchown()</i>	<i>link()</i>	<i>pwrite()</i>	<i>unlink()</i>	

2400 If two threads each call one of these functions, each call shall either see all of the specified effects
2401 of the other call, or none of them.

2402 2.9.8 Use of Application-Managed Thread Stacks

2403 An “application-managed thread stack” is a region of memory allocated by the application—for
2404 example, memory returned by the *malloc()* or *mmap()* functions—and designated as a stack
2405 through the act of passing the address and size of the stack, respectively, as the *stackaddr* and
2406 *stacksize* arguments to *pthread_attr_setstack()*. Application-managed stacks allow the application
2407 to precisely control the placement and size of a stack.

2408 The application grants to the implementation permanent ownership of and control over the
2409 application-managed stack when the attributes object in which the *stack* or *stackaddr* attribute has
2410 been set is used, either by presenting that attribute’s object as the *attr* argument in a call to
2411 *pthread_create()* that completes successfully, or by storing a pointer to the attributes object in the
2412 *sigev_notify_attributes* member of a **struct sigevent** and passing that **struct sigevent** to a function
2413 accepting such argument that completes successfully. The application may thereafter utilize the
2414 memory within the stack only within the normal context of stack usage within or properly
2415 synchronized with a thread that has been scheduled by the implementation with stack pointer
2416 value(s) that are within the range of that stack. In particular, the region of memory cannot be
2417 freed, nor can it be later specified as the stack for another thread.

2418 When specifying an attributes object with an application-managed stack through the
2419 *sigev_notify_attributes* member of a **struct sigevent**, the results are undefined if the requested
2420 signal is generated multiple times (as for a repeating timer).

2421 Until an attributes object in which the *stack* or *stackaddr* attribute has been set is used, the
2422 application retains ownership of and control over the memory allocated to the stack. It may free
2423 or reuse the memory as long as it either deletes the attributes object, or before using the
2424 attributes object replaces the stack by making an additional call to *pthread_attr_setstack()*, that
2425 was used originally to designate the stack. There is no mechanism to retract the reference to an
2426 application-managed stack by an existing attributes object.

2427 Once an attributes object with an application-managed stack has been used, that attributes object
2428 cannot be used again by a subsequent call to *pthread_create()* or any function accepting a **struct**
2429 **sigevent** with *sigev_notify_attributes* containing a pointer to the attributes object, without
2430 designating an unused application-managed stack by making an additional call to
2431 *pthread_attr_setstack()*.

2432 2.10 Sockets

2433 A socket is an endpoint for communication using the facilities described in this section. A socket
 2434 is created with a specific socket type, described in [Section 2.10.6](#) (on page 61), and is associated
 2435 with a specific protocol, detailed in [Section 2.10.3](#) (on page 60). A socket is accessed via a file
 2436 descriptor obtained when the socket is created.

2437 2.10.1 Address Families

2438 All network protocols are associated with a specific address family. An address family provides
 2439 basic services to the protocol implementation to allow it to function within a specific network
 2440 environment. These services may include packet fragmentation and reassembly, routing,
 2441 addressing, and basic transport. An address family is normally comprised of a number of
 2442 protocols, one per socket type. Each protocol is characterized by an abstract socket type. It is not
 2443 required that an address family support all socket types. An address family may contain
 2444 multiple protocols supporting the same socket abstraction.

2445 [Section 2.10.17](#) (on page 67), [Section 2.10.19](#) (on page 68), and [Section 2.10.20](#) (on page 68),
 2446 respectively, describe the use of sockets for local UNIX connections, for Internet protocols based
 2447 on IPv4, and for Internet protocols based on IPv6.

2448 2.10.2 Addressing

2449 An address family defines the format of a socket address. All network addresses are described
 2450 using a general structure, called a `sockaddr`, as defined in the Base Definitions volume of
 2451 IEEE Std 1003.1-200x, `<sys/socket.h>`. However, each address family imposes finer and more
 2452 specific structure, generally defining a structure with fields specific to the address family. The
 2453 field `sa_family` in the `sockaddr` structure contains the address family identifier, specifying the
 2454 format of the `sa_data` area. The size of the `sa_data` area is unspecified.

2455 2.10.3 Protocols

2456 A protocol supports one of the socket abstractions detailed in [Section 2.10.6](#) (on page 61).
 2457 Selecting a protocol involves specifying the address family, socket type, and protocol number to
 2458 the `socket()` function. Certain semantics of the basic socket abstractions are protocol-specific. All
 2459 protocols are expected to support the basic model for their particular socket type, but may, in
 2460 addition, provide non-standard facilities or extensions to a mechanism.

2461 2.10.4 Routing

2462 Sockets provides packet routing facilities. A routing information database is maintained, which
 2463 is used in selecting the appropriate network interface when transmitting packets.

2464 2.10.5 Interfaces

2465 Each network interface in a system corresponds to a path through which messages can be sent
 2466 and received. A network interface usually has a hardware device associated with it, though
 2467 certain interfaces such as the loopback interface, do not.

2.10.6 Socket Types

2468
2469
2470 RS A socket is created with a specific type, which defines the communication semantics and which
2471 allows the selection of an appropriate communication protocol. Four types are defined:
2472 `SOCK_RAW`, `SOCK_STREAM`, `SOCK_SEQPACKET`, and `SOCK_DGRAM`. Implementations
may specify additional socket types.

2473
2474 The `SOCK_STREAM` socket type provides reliable, sequenced, full-duplex octet streams
2475 between the socket and a peer to which the socket is connected. A socket of type
2476 `SOCK_STREAM` must be in a connected state before any data may be sent or received. Record
2477 boundaries are not maintained; data sent on a stream socket using output operations of one size
2478 may be received using input operations of smaller or larger sizes without loss of data. Data may
2479 be buffered; successful return from an output function does not imply that the data has been
2480 delivered to the peer or even transmitted from the local system. If data cannot be successfully
2481 transmitted within a given time then the connection is considered broken, and subsequent
2482 operations shall fail. A `SIGPIPE` signal is raised if a thread sends on a broken stream (one that is
no longer connected). Support for an out-of-band data transmission facility is protocol-specific.

2483
2484 The `SOCK_SEQPACKET` socket type is similar to the `SOCK_STREAM` type, and is also
2485 connection-oriented. The only difference between these types is that record boundaries are
2486 maintained using the `SOCK_SEQPACKET` type. A record can be sent using one or more output
2487 operations and received using one or more input operations, but a single operation never
2488 transfers parts of more than one record. Record boundaries are visible to the receiver via the
2489 `MSG_EOR` flag in the received message flags returned by the `recvmsg()` function. It is protocol-
specific whether a maximum record size is imposed.

2490
2491 The `SOCK_DGRAM` socket type supports connectionless data transfer which is not necessarily
2492 acknowledged or reliable. Datagrams may be sent to the address specified (possibly multicast or
2493 broadcast) in each output operation, and incoming datagrams may be received from multiple
2494 sources. The source address of each datagram is available when receiving the datagram. An
2495 application may also pre-specify a peer address, in which case calls to output functions that do
2496 not specify a peer address shall send to the pre-specified peer. If a peer has been specified, only
2497 datagrams from that peer shall be received. A datagram must be sent in a single output
2498 operation, and must be received in a single input operation. The maximum size of a datagram is
2499 protocol-specific; with some protocols, the limit is implementation-defined. Output datagrams
2500 may be buffered within the system; thus, a successful return from an output function does not
2501 guarantee that a datagram is actually sent or received. However, implementations should
2502 attempt to detect any errors possible before the return of an output function, reporting any error
by an unsuccessful return value.

2503 RS The `SOCK_RAW` socket type is similar to the `SOCK_DGRAM` type. It differs in that it is
2504 normally used with communication providers that underlie those used for the other socket
2505 types. For this reason, the creation of a socket with type `SOCK_RAW` shall require appropriate
2506 privilege. The format of datagrams sent and received with this socket type generally include
2507 specific protocol headers, and the formats are protocol-specific and implementation-defined.

2508 **2.10.7 Socket I/O Mode**

2509 The I/O mode of a socket is described by the `O_NONBLOCK` file status flag which pertains to
 2510 the open file description for the socket. This flag is initially off when a socket is created, but may
 2511 be set and cleared by the use of the `F_SETFL` command of the `fcntl()` function.

2512 When the `O_NONBLOCK` flag is set, functions that would normally block until they are
 2513 complete shall either return immediately with an error, or shall complete asynchronously to the
 2514 execution of the calling process. Data transfer operations (the `read()`, `write()`, `send()`, and `recv()`
 2515 functions) shall complete immediately, transfer only as much as is available, and then return
 2516 without blocking, or return an error indicating that no transfer could be made without blocking.
 2517 The `connect()` function initiates a connection and shall return without blocking when
 2518 `O_NONBLOCK` is set; it shall return the error `[EINPROGRESS]` to indicate that the connection
 2519 was initiated successfully, but that it has not yet completed.

2520 **2.10.8 Socket Owner**

2521 The owner of a socket is unset when a socket is created. The owner may be set to a process ID or
 2522 process group ID using the `F_SETOWN` command of the `fcntl()` function.

2523 **2.10.9 Socket Queue Limits**

2524 The transmit and receive queue sizes for a socket are set when the socket is created. The default
 2525 sizes used are both protocol-specific and implementation-defined. The sizes may be changed
 2526 using the `setsockopt()` function.

2527 **2.10.10 Pending Error**

2528 Errors may occur asynchronously, and be reported to the socket in response to input from the
 2529 network protocol. The socket stores the pending error to be reported to a user of the socket at the
 2530 next opportunity. The error is returned in response to a subsequent `send()`, `recv()`, or `getsockopt()`
 2531 operation on the socket, and the pending error is then cleared.

2532 **2.10.11 Socket Receive Queue**

2533 A socket has a receive queue that buffers data when it is received by the system until it is
 2534 removed by a receive call. Depending on the type of the socket and the communication provider,
 2535 the receive queue may also contain ancillary data such as the addressing and other protocol data
 2536 associated with the normal data in the queue, and may contain out-of-band or expedited data.
 2537 The limit on the queue size includes any normal, out-of-band data, datagram source addresses,
 2538 and ancillary data in the queue. The description in this section applies to all sockets, even
 2539 though some elements cannot be present in some instances.

2540 The contents of a receive buffer are logically structured as a series of data segments with
 2541 associated ancillary data and other information. A data segment may contain normal data or
 2542 out-of-band data, but never both. A data segment may complete a record if the protocol
 2543 supports records (always true for types `SOCK_SEQPACKET` and `SOCK_DGRAM`). A record
 2544 may be stored as more than one segment; the complete record might never be present in the
 2545 receive buffer at one time, as a portion might already have been returned to the application, and
 2546 another portion might not yet have been received from the communications provider. A data
 2547 segment may contain ancillary protocol data, which is logically associated with the segment.
 2548 Ancillary data is received as if it were queued along with the first normal data octet in the
 2549 segment (if any). A segment may contain ancillary data only, with no normal or out-of-band
 2550 data. For the purposes of this section, a datagram is considered to be a data segment that
 2551 terminates a record, and that includes a source address as a special type of ancillary data. Data
 2552 segments are placed into the queue as data is delivered to the socket by the protocol. Normal
 2553 data segments are placed at the end of the queue as they are delivered. If a new segment

2554 contains the same type of data as the preceding segment and includes no ancillary data, and if
 2555 the preceding segment does not terminate a record, the segments are logically merged into a
 2556 single segment.

2557 The receive queue is logically terminated if an end-of-file indication has been received or a
 2558 connection has been terminated. A segment shall be considered to be terminated if another
 2559 segment follows it in the queue, if the segment completes a record, or if an end-of-file or other
 2560 connection termination has been reported. The last segment in the receive queue shall also be
 2561 considered to be terminated while the socket has a pending error to be reported.

2562 A receive operation shall never return data or ancillary data from more than one segment.

2563 **2.10.12 Socket Out-of-Band Data State**

2564 The handling of received out-of-band data is protocol-specific. Out-of-band data may be placed
 2565 in the socket receive queue, either at the end of the queue or before all normal data in the queue.
 2566 In this case, out-of-band data is returned to an application program by a normal receive call.
 2567 Out-of-band data may also be queued separately rather than being placed in the socket receive
 2568 queue, in which case it shall be returned only in response to a receive call that requests out-of-
 2569 band data. It is protocol-specific whether an out-of-band data mark is placed in the receive
 2570 queue to demarcate data preceding the out-of-band data and following the out-of-band data. An
 2571 out-of-band data mark is logically an empty data segment that cannot be merged with other
 2572 segments in the queue. An out-of-band data mark is never returned in response to an input
 2573 operation. The *socketmark()* function can be used to test whether an out-of-band data mark is the
 2574 first element in the queue. If an out-of-band data mark is the first element in the queue when an
 2575 input function is called without the MSG_PEEK option, the mark is removed from the queue
 2576 and the following data (if any) is processed as if the mark had not been present.

2577 **2.10.13 Connection Indication Queue**

2578 Sockets that are used to accept incoming connections maintain a queue of outstanding
 2579 connection indications. This queue is a list of connections that are awaiting acceptance by the
 2580 application; see *listen()*.

2581 **2.10.14 Signals**

2582 One category of event at the socket interface is the generation of signals. These signals report
 2583 protocol events or process errors relating to the state of the socket. The generation or delivery of
 2584 a signal does not change the state of the socket, although the generation of the signal may have
 2585 been caused by a state change.

2586 The SIGPIPE signal shall be sent to a thread that attempts to send data on a socket that is no
 2587 longer able to send. In addition, the send operation fails with the error [EPIPE].

2588 If a socket has an owner, the SIGURG signal is sent to the owner of the socket when it is notified
 2589 of expedited or out-of-band data. The socket state at this time is protocol-dependent, and the
 2590 status of the socket is specified in [Section 2.10.17](#) (on page 67), [Section 2.10.19](#) (on page 68), and
 2591 [Section 2.10.20](#) (on page 68). Depending on the protocol, the expedited data may or may not
 2592 have arrived at the time of signal generation.

2593 2.10.15 Asynchronous Errors

2594 If any of the following conditions occur asynchronously for a socket, the corresponding value
2595 listed below shall become the pending error for the socket:

2596 [ECONNABORTED]

2597 The connection was aborted locally.

2598 [ECONNREFUSED]

2599 For a connection-mode socket attempting a non-blocking connection, the attempt to connect
2600 was forcefully rejected. For a connectionless-mode socket, an attempt to deliver a datagram
2601 was forcefully rejected.

2602 [ECONNRESET]

2603 The peer has aborted the connection.

2604 [EHOSTDOWN]

2605 The destination host has been determined to be down or disconnected.

2606 [EHOSTUNREACH]

2607 The destination host is not reachable.

2608 [EMSGSIZE]

2609 For a connectionless-mode socket, the size of a previously sent datagram prevented
2610 delivery.

2611 [ENETDOWN]

2612 The local network connection is not operational.

2613 [ENETRESET]

2614 The connection was aborted by the network.

2615 [ENETUNREACH]

2616 The destination network is not reachable.

2617 2.10.16 Use of Options

2618 There are a number of socket options which either specialize the behavior of a socket or provide
2619 useful information. These options may be set at different protocol levels and are always present
2620 at the uppermost "socket" level.

2621 Socket options are manipulated by two functions, *getsockopt()* and *setsockopt()*. These functions
2622 allow an application program to customize the behavior and characteristics of a socket to
2623 provide the desired effect.

2624 All of the options have default values. The type and meaning of these values is defined by the
2625 protocol level to which they apply. Instead of using the default values, an application program
2626 may choose to customize one or more of the options. However, in the bulk of cases, the default
2627 values are sufficient for the application.

2628 Some of the options are used to enable or disable certain behavior within the protocol modules
2629 (for example, turn on debugging) while others may be used to set protocol-specific information
2630 (for example, IP time-to-live on all the application's outgoing packets). As each of the options is
2631 introduced, its effect on the underlying protocol modules is described.

2632 [Table 2-1](#) shows the value for the socket level.

2633

Table 2-1 Value of Level for Socket Options

2634

2635

Name	Description
SOL_SOCKET	Options are intended for the sockets level.

2636

2637

2638

2639

2640

Table 2-2 lists those options present at the socket level; that is, when the *level* parameter of the *getsockopt()* or *setsockopt()* function is SOL_SOCKET, the types of the option value parameters associated with each option, and a brief synopsis of the meaning of the option value parameter. Unless otherwise noted, each may be examined with *getsockopt()* and set with *setsockopt()* on all types of socket.

2641

Table 2-2 Socket-Level Options

2642

2643

2644

2645

2646

2647

2648

2649

2650

2651

2652

2653

2654

2655

2656

2657

2658

2659

2660

2661

2662

2663

2664

2665

2666

2667

2668

2669

Option	Parameter Type	Parameter Meaning
SO_BROADCAST	int	Non-zero requests permission to transmit broadcast datagrams (SOCK_DGRAM sockets only).
SO_DEBUG	int	Non-zero requests debugging in underlying protocol modules.
SO_DONTROUTE	int	Non-zero requests bypass of normal routing; route based on destination address only.
SO_ERROR	int	Requests and clears pending error information on the socket (<i>getsockopt()</i> only).
SO_KEEPALIVE	int	Non-zero requests periodic transmission of keepalive messages (protocol-specific).
SO_LINGER	struct linger	Specify actions to be taken for queued, unsent data on <i>close()</i> : linger on/off and linger time in seconds.
SO_OOINLINE	int	Non-zero requests that out-of-band data be placed into normal data input queue as received.
SO_RCVBUF	int	Size of receive buffer (in bytes).
SO_RCVLOWAT	int	Minimum amount of data to return to application for input operations (in bytes).
SO_RCVTIMEO	struct timeval	Timeout value for a socket receive operation.
SO_REUSEADDR	int	Non-zero requests reuse of local addresses in <i>bind()</i> (protocol-specific).
SO_SNDBUF	int	Size of send buffer (in bytes).
SO_SNDLOWAT	int	Minimum amount of data to send for output operations (in bytes).
SO_SNDTIMEO	struct timeval	Timeout value for a socket send operation.
SO_TYPE	int	Identify socket type (<i>getsockopt()</i> only).

2670

2671

2672

The SO_BROADCAST option requests permission to send broadcast datagrams on the socket. Support for SO_BROADCAST is protocol-specific. The default for SO_BROADCAST is that the ability to send broadcast datagrams on a socket is disabled.

2673

2674

2675

2676

The SO_DEBUG option enables debugging in the underlying protocol modules. This can be useful for tracing the behavior of the underlying protocol modules during normal system operation. The semantics of the debug reports are implementation-defined. The default value for SO_DEBUG is for debugging to be turned off.

2677 The SO_DONTROUTE option requests that outgoing messages bypass the standard routing
 2678 facilities. The destination must be on a directly-connected network, and messages are directed to
 2679 the appropriate network interface according to the destination address. It is protocol-specific
 2680 whether this option has any effect and how the outgoing network interface is chosen. Support
 2681 for this option with each protocol is implementation-defined.

2682 The SO_ERROR option is used only on *getsockopt()*. When this option is specified, *getsockopt()*
 2683 shall return any pending error on the socket and clear the error status. It shall return a value of 0
 2684 if there is no pending error. SO_ERROR may be used to check for asynchronous errors on
 2685 connected connectionless-mode sockets or for other types of asynchronous errors. SO_ERROR
 2686 has no default value.

2687 The SO_KEEPALIVE option enables the periodic transmission of messages on a connected
 2688 socket. The behavior of this option is protocol-specific. The default value for SO_KEEPALIVE is
 2689 zero, specifying that this capability is turned off.

2690 The SO_LINGER option controls the action of the interface when unsent messages are queued
 2691 on a socket and a *close()* is performed. The details of this option are protocol-specific. The
 2692 default value for SO_LINGER is zero, or off, for the *l_onoff* element of the option value and zero
 2693 seconds for the linger time specified by the *l_linger* element.

2694 The SO_OOBINLINE option is valid only on protocols that support out-of-band data. The
 2695 SO_OOBINLINE option requests that out-of-band data be placed in the normal data input
 2696 queue as received; it is then accessible using the *read()* or *recv()* functions without the
 2697 MSG_OOB flag set. The default for SO_OOBINLINE is off; that is, for out-of-band data not to be
 2698 placed in the normal data input queue.

2699 The SO_RCVBUF option requests that the buffer space allocated for receive operations on this
 2700 socket be set to the value, in bytes, of the option value. Applications may wish to increase buffer
 2701 size for high volume connections, or may decrease buffer size to limit the possible backlog of
 2702 incoming data. The default value for the SO_RCVBUF option value is implementation-defined,
 2703 and may vary by protocol.

2704 The SO_RCVLOWAT option sets the minimum number of bytes to process for socket input
 2705 operations. In general, receive calls block until any (non-zero) amount of data is received, then
 2706 return the smaller of the amount available or the amount requested. The default value for
 2707 SO_RCVLOWAT is 1, and does not affect the general case. If SO_RCVLOWAT is set to a larger
 2708 value, blocking receive calls normally wait until they have received the smaller of the low water
 2709 mark value or the requested amount. Receive calls may still return less than the low water mark
 2710 if an error occurs, a signal is caught, or the type of data next in the receive queue is different
 2711 from that returned (for example, out-of-band data). As mentioned previously, the default value
 2712 for SO_RCVLOWAT is 1 byte. It is implementation-defined whether the SO_RCVLOWAT option
 2713 can be set.

2714 The SO_RCVTIMEO option is an option to set a timeout value for input operations. It accepts a
 2715 **timeval** structure with the number of seconds and microseconds specifying the limit on how
 2716 long to wait for an input operation to complete. If a receive operation has blocked for this much
 2717 time without receiving additional data, it shall return with a partial count or *errno* shall be set to
 2718 [EWOULDBLOCK] if no data were received. The default for this option is the value zero, which
 2719 indicates that a receive operation will not time out. It is implementation-defined whether the
 2720 SO_RCVTIMEO option can be set.

2721 The SO_REUSEADDR option indicates that the rules used in validating addresses supplied in a
 2722 *bind()* should allow reuse of local addresses. Operation of this option is protocol-specific. The
 2723 default value for SO_REUSEADDR is off; that is, reuse of local addresses is not permitted.

2724 The SO_SNDBUF option requests that the buffer space allocated for send operations on this

2725 socket be set to the value, in bytes, of the option value. The default value for the SO_SNDBUF
2726 option value is implementation-defined, and may vary by protocol.

2727 The SO_SNDLOWAT option sets the minimum number of bytes to process for socket output
2728 operations. Most output operations process all of the data supplied by the call, delivering data to
2729 the protocol for transmission and blocking as necessary for flow control. Non-blocking output
2730 operations process as much data as permitted subject to flow control without blocking, but
2731 process no data if flow control does not allow the smaller of the send low water mark value or
2732 the entire request to be processed. A *select()* operation testing the ability to write to a socket shall
2733 return true only if the send low water mark could be processed. The default value for
2734 SO_SNDLOWAT is implementation-defined and protocol-specific. It is implementation-defined
2735 whether the SO_SNDLOWAT option can be set.

2736 The SO_SNDTIMEO option is an option to set a timeout value for the amount of time that an
2737 output function shall block because flow control prevents data from being sent. As noted in
2738 Table 2-2 (on page 65), the option value is a **timeval** structure with the number of seconds and
2739 microseconds specifying the limit on how long to wait for an output operation to complete. If a
2740 send operation has blocked for this much time, it shall return with a partial count or *errno* set to
2741 [EWOULDBLOCK] if no data were sent. The default for this option is the value zero, which
2742 indicates that a send operation will not time out. It is implementation-defined whether the
2743 SO_SNDTIMEO option can be set.

2744 The SO_TYPE option is used only on *getsockopt()*. When this option is specified, *getsockopt()*
2745 shall return the type of the socket (for example, SOCK_STREAM). This option is useful to
2746 servers that inherit sockets on start-up. SO_TYPE has no default value.

2747 2.10.17 Use of Sockets for Local UNIX Connections

2748 Support for UNIX domain sockets is mandatory.

2749 UNIX domain sockets provide process-to-process communication in a single system.

2750 2.10.17.1 Headers

2751 The symbolic constant AF_UNIX defined in the `<sys/socket.h>` header is used to identify the
2752 UNIX domain address family. The `<sys/un.h>` header contains other definitions used in
2753 connection with UNIX domain sockets. See the Base Definitions volume of IEEE Std 1003.1-200x,
2754 Chapter 13, Headers.

2755 The **sockaddr_storage** structure defined in `<sys/socket.h>` shall be large enough to
2756 accommodate a **sockaddr_un** structure (see the `<sys/un.h>` header defined in the Base
2757 Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers) and shall be aligned at an
2758 appropriate boundary so that pointers to it can be cast as pointers to **sockaddr_un** structures
2759 and used to access the fields of those structures without alignment problems. When a
2760 **sockaddr_storage** structure is cast as a **sockaddr_un** structure, the *ss_family* field maps onto the
2761 *sun_family* field.

2762 2.10.18 Use of Sockets over Internet Protocols

2763 When a socket is created in the Internet family with a protocol value of zero, the implementation
2764 shall use the protocol listed below for the type of socket created.

2765	SOCK_STREAM	IPPROTO_TCP.
2766	SOCK_DGRAM	IPPROTO_UDP.
2767	RS SOCK_RAW	IPPROTO_RAW.

2768 SOCK_SEQPACKET Unspecified.

2769 RS A raw interface to IP is available by creating an Internet socket of type SOCK_RAW. The default
 2770 protocol for type SOCK_RAW shall be identified in the IP header with the value
 2771 IPPROTO_RAW. Applications should not use the default protocol when creating a socket with
 2772 type SOCK_RAW, but should identify a specific protocol by value. The ICMP control protocol is
 2773 accessible from a raw socket by specifying a value of IPPROTO_ICMP for protocol.

2774 2.10.19 Use of Sockets over Internet Protocols Based on IPv4

2775 Support for sockets over Internet protocols based on IPv4 is mandatory.

2776 2.10.19.1 Headers

2777 The symbolic constant AF_INET defined in the `<sys/socket.h>` header is used to identify the
 2778 IPv4 Internet address family. The `<netinet/in.h>` header contains other definitions used in
 2779 connection with IPv4 Internet sockets. See the Base Definitions volume of IEEE Std 1003.1-200x,
 2780 Chapter 13, Headers.

2781 The `sockaddr_storage` structure defined in `<sys/socket.h>` shall be large enough to
 2782 accommodate a `sockaddr_in` structure (see the `<netinet/in.h>` header defined in the Base
 2783 Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers) and shall be aligned at an
 2784 appropriate boundary so that pointers to it can be cast as pointers to `sockaddr_in` structures and
 2785 used to access the fields of those structures without alignment problems. When a
 2786 `sockaddr_storage` structure is cast as a `sockaddr_in` structure, the `ss_family` field maps onto the
 2787 `sin_family` field.

2788 2.10.20 Use of Sockets over Internet Protocols Based on IPv6

2789 IP6 This section describes extensions to support sockets over Internet protocols based on IPv6. The
 2790 functionality described in this section shall be provided on implementations that support the
 2791 IPV6 option (and the rest of this section is not further shaded for this option).

2792 To enable smooth transition from IPv4 to IPv6, the features defined in this section may, in certain
 2793 circumstances, also be used in connection with IPv4; see [Section 2.10.20.2](#) (on page 69).

2794 2.10.20.1 Addressing

2795 IPv6 overcomes the addressing limitations of previous versions by using 128-bit addresses
 2796 instead of 32-bit addresses. The IPv6 address architecture is described in RFC 2373.

2797 There are three kinds of IPv6 address:

2798 Unicast

2799 Identifies a single interface.

2800 A unicast address can be global, link-local (designed for use on a single link), or site-local
 2801 (designed for systems not connected to the Internet). Link-local and site-local addresses
 2802 need not be globally unique.

2803 Anycast

2804 Identifies a set of interfaces such that a packet sent to the address can be delivered to any
 2805 member of the set.

2806 An anycast address is similar to a unicast address; the nodes to which an anycast address is
 2807 assigned must be explicitly configured to know that it is an anycast address.

2808 Multicast

2809 Identifies a set of interfaces such that a packet sent to the address should be delivered to
 2810 every member of the set.

2811 An application can send multicast datagrams by simply specifying an IPv6 multicast
 2812 address in the *address* argument of *sendto()*. To receive multicast datagrams, an application
 2813 must join the multicast group (using *setsockopt()* with `IPV6_JOIN_GROUP`) and must bind
 2814 to the socket the UDP port on which datagrams will be received. Some applications should
 2815 also bind the multicast group address to the socket, to prevent other datagrams destined to
 2816 that port from being delivered to the socket.

2817 A multicast address can be global, node-local, link-local, site-local, or organization-local.

2818 The following special IPv6 addresses are defined:

2819 Unspecified

2820 An address that is not assigned to any interface and is used to indicate the absence of an
 2821 address.

2822 Loopback

2823 A unicast address that is not assigned to any interface and can be used by a node to send
 2824 packets to itself.

2825 Two sets of IPv6 addresses are defined to correspond to IPv4 addresses:

2826 IPv4-compatible addresses

2827 These are assigned to nodes that support IPv6 and can be used when traffic is “tunneled”
 2828 through IPv4.

2829 IPv4-mapped addresses

2830 These are used to represent IPv4 addresses in IPv6 address format; see [Section 2.10.20.2](#) (on
 2831 page 69).

2832 Note that the unspecified address and the loopback address must not be treated as
 2833 IPv4-compatible addresses.

2834 2.10.20.2 Compatibility with IPv4

2835 The API provides the ability for IPv6 applications to interoperate with applications using IPv4,
 2836 by using IPv4-mapped IPv6 addresses. These addresses can be generated automatically by the
 2837 *getaddrinfo()* function when the specified host has only IPv4 addresses.

2838 Applications can use `AF_INET6` sockets to open TCP connections to IPv4 nodes, or send UDP
 2839 packets to IPv4 nodes, by simply encoding the destination’s IPv4 address as an IPv4-mapped
 2840 IPv6 address, and passing that address, within a `sockaddr_in6` structure, in the *connect()*,
 2841 *sendto()*, or *sendmsg()* function. When applications use `AF_INET6` sockets to accept TCP
 2842 connections from IPv4 nodes, or receive UDP packets from IPv4 nodes, the system shall return
 2843 the peer’s address to the application in the *accept()*, *recvfrom()*, *recvmsg()*, or *getpeername()*
 2844 function using a `sockaddr_in6` structure encoded this way. If a node has an IPv4 address, then
 2845 the implementation shall allow applications to communicate using that address via an
 2846 `AF_INET6` socket. In such a case, the address will be represented at the API by the
 2847 corresponding IPv4-mapped IPv6 address. Also, the implementation may allow an `AF_INET6`
 2848 socket bound to `in6addr_any` to receive inbound connections and packets destined to one of the
 2849 node’s IPv4 addresses.

2850 An application can use `AF_INET6` sockets to bind to a node’s IPv4 address by specifying the
 2851 address as an IPv4-mapped IPv6 address in a `sockaddr_in6` structure in the *bind()* function. For
 2852 an `AF_INET6` socket bound to a node’s IPv4 address, the system shall return the address in the
 2853 *getsockname()* function as an IPv4-mapped IPv6 address in a `sockaddr_in6` structure.

2854 2.10.20.3 Interface Identification

2855 Each local interface is assigned a unique positive integer as a numeric index. Indexes start at 1;
 2856 zero is not used. There may be gaps so that there is no current interface for a particular positive
 2857 index. Each interface also has a unique implementation-defined name.

2858 2.10.20.4 Options

2859 The following options apply at the IPPROTO_IPV6 level:

2860 IPV6_JOIN_GROUP

2861 When set via *setsockopt()*, it joins the application to a multicast group on an interface
 2862 (identified by its index) and addressed by a given multicast address, enabling packets sent
 2863 to that address to be read via the socket. If the interface index is specified as zero, the
 2864 system selects the interface (for example, by looking up the address in a routing table and
 2865 using the resulting interface).

2866 An attempt to read this option using *getsockopt()* shall result in an [EOPNOTSUPP] error.

2867 The parameter type of this option is a pointer to an **ipv6_mreq** structure.

2868 IPV6_LEAVE_GROUP

2869 When set via *setsockopt()*, it removes the application from the multicast group on an
 2870 interface (identified by its index) and addressed by a given multicast address.

2871 An attempt to read this option using *getsockopt()* shall result in an [EOPNOTSUPP] error.

2872 The parameter type of this option is a pointer to an **ipv6_mreq** structure.

2873 IPV6_MULTICAST_HOPS

2874 The value of this option is the hop limit for outgoing multicast IPv6 packets sent via the
 2875 socket. Its possible values are the same as those of IPV6_UNICAST_HOPS. If the
 2876 IPV6_MULTICAST_HOPS option is not set, a value of 1 is assumed. This option can be set
 2877 via *setsockopt()* and read via *getsockopt()*.

2878 The parameter type of this option is a pointer to an **int**. (Default value: 1)

2879 IPV6_MULTICAST_IF

2880 The index of the interface to be used for outgoing multicast packets. It can be set via
 2881 *setsockopt()* and read via *getsockopt()*. If the interface index is specified as zero, the system
 2882 selects the interface (for example, by looking up the address in a routing table and using the
 2883 resulting interface).

2884 The parameter type of this option is a pointer to an **unsigned int**. (Default value: 0)

2885 IPV6_MULTICAST_LOOP

2886 This option controls whether outgoing multicast packets should be delivered back to the
 2887 local application when the sending interface is itself a member of the destination multicast
 2888 group. If it is set to 1 they are delivered. If it is set to 0 they are not. Other values result in an
 2889 [EINVAL] error. This option can be set via *setsockopt()* and read via *getsockopt()*.

2890 The parameter type of this option is a pointer to an **unsigned int** which is used as a Boolean
 2891 value. (Default value: 1)

2892 IPV6_UNICAST_HOPS

2893 The value of this option is the hop limit for outgoing unicast IPv6 packets sent via the
 2894 socket. If the option is not set, or is set to -1, the system selects a default value. Attempts to
 2895 set a value less than -1 or greater than 255 shall result in an [EINVAL] error. This option can
 2896 be set via *setsockopt()* and read via *getsockopt()*.

2897 The parameter type of this option is a pointer to an **int**. (Default value: Unspecified)

2898 **IPV6_V6ONLY**
 2899 This socket option restricts AF_INET6 sockets to IPv6 communications only. AF_INET6
 2900 sockets may be used for both IPv4 and IPv6 communications. Some applications may want
 2901 to restrict their use of an AF_INET6 socket to IPv6 communications only. For these
 2902 applications, the IPV6_V6ONLY socket option is defined. When this option is turned on, the
 2903 socket can be used to send and receive IPv6 packets only. This is an IPPROTO_IPV6-level
 2904 option.
 2905 The parameter type of this option is a pointer to an **int** which is used as a Boolean value.
 2906 (Default value: 0)
 2907 An [EOPNOTSUPP] error shall result if IPV6_JOIN_GROUP or IPV6_LEAVE_GROUP is used
 2908 with *getsockopt()*.

2909 2.10.20.5 Headers

2910 The symbolic constant AF_INET6 is defined in the **<sys/socket.h>** header to identify the IPv6
 2911 Internet address family. See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13,
 2912 Headers.

2913 The **sockaddr_storage** structure defined in **<sys/socket.h>** shall be large enough to
 2914 accommodate a **sockaddr_in6** structure (see the **<netinet/in.h>** header defined in the Base
 2915 Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers) and shall be aligned at an
 2916 appropriate boundary so that pointers to it can be cast as pointers to **sockaddr_in6** structures
 2917 and used to access the fields of those structures without alignment problems. When a
 2918 **sockaddr_storage** structure is cast as a **sockaddr_in6** structure, the *ss_family* field maps onto the
 2919 *sin6_family* field.

2920 The **<netinet/in.h>**, **<arpa/inet.h>**, and **<netdb.h>** headers contain other definitions used in
 2921 connection with IPv6 Internet sockets; see the Base Definitions volume of IEEE Std 1003.1-200x,
 2922 Chapter 13, Headers.

2923 2.11 Tracing

2924 OB TRC This section describes extensions to support tracing of user applications. The functionality
 2925 described in this section is dependent on support of the Trace option (and the rest of this section
 2926 is not further shaded for this option).

2927 The tracing facilities defined in IEEE Std 1003.1-200x allow a process to select a set of trace event
 2928 types, to activate a trace stream of the selected trace events as they occur in the flow of
 2929 execution, and to retrieve the recorded trace events.

2930 The tracing operation relies on three logically different components: the traced process, the
 2931 controller process, and the analyzer process. During the execution of the traced process, when a
 2932 trace point is reached, a trace event is recorded into the trace streams created for that process in
 2933 which the associated trace event type identifier is not being filtered out. The controller process
 2934 controls the operation of recording the trace events into the trace stream. It shall be able to:

- 2935 • Initialize the attributes of a trace stream
- 2936 • Create the trace stream (for a specified traced process) using those attributes
- 2937 • Start and stop tracing for the trace stream
- 2938 • Filter the type of trace events to be recorded, if the Trace Event Filter option is supported
- 2939 • Shut a trace stream down

2940 These operations can be done for an active trace stream. The analyzer process retrieves the

2941 traced events either at runtime, when the trace stream has not yet been shut down, but is still
 2942 recording trace events; or after opening a trace log that had been previously recorded and shut
 2943 down. These three logically different operations can be performed by the same process, or can
 2944 be distributed into different processes.

2945 A trace stream identifier can be created by a call to *posix_trace_create()*,
 2946 *posix_trace_create_withlog()*, or *posix_trace_open()*. The *posix_trace_create()* and
 2947 *posix_trace_create_withlog()* functions should be used by a controller process. The
 2948 *posix_trace_open()* should be used by an analyzer process.

2949 The tracing functions can serve different purposes. One purpose is debugging the possibly pre-
 2950 instrumented code, while another is post-mortem fault analysis. These two potential uses differ
 2951 in that the first requires pre-filtering capabilities to avoid overwhelming the trace stream and
 2952 permits focusing on expected information; while the second needs comprehensive trace
 2953 capabilities in order to be able to record all types of information.

2954 The events to be traced belong to two classes:

- 2955 1. User trace events (generated by the application instrumentation)
- 2956 2. System trace events (generated by the operating system)

2957 The trace interface defines several system trace event types associated with control of and
 2958 operation of the trace stream. This small set of system trace events includes the minimum
 2959 required to interpret correctly the trace event information present in the stream. Other desirable
 2960 system trace events for some particular application profile may be implemented and are
 2961 encouraged; for example, process and thread scheduling, signal occurrence, and so on.

2962 Each traced process shall have a mapping of the trace event names to trace event type identifiers
 2963 that have been defined for that process. Each active trace stream shall have a mapping that
 2964 incorporates all the trace event type identifiers predefined by the trace system plus all the
 2965 mappings of trace event names to trace event type identifiers of the processes that are being
 2966 traced into that trace stream. These mappings are defined from the instrumented application by
 2967 calling the *posix_trace_eventid_open()* function and from the controller process by calling the
 2968 *posix_trace_trid_eventid_open()* function. For a pre-recorded trace stream, the list of trace event
 2969 types is obtained from the pre-recorded trace log.

2970 The *st_ctime* and *st_mtime* fields of a file associated with an active trace stream shall be marked
 2971 for update every time any of the tracing operations modifies that file.

2972 The *st_atime* field of a file associated with a trace stream shall be marked for update every time
 2973 any of the tracing operations causes data to be read from that file.

2974 Results are undefined if the application performs any operation on a file descriptor associated
 2975 with an active or pre-recorded trace stream until *posix_trace_shutdown()* or *posix_trace_close()* is
 2976 called for that trace stream. Results are also undefined if the analyzer process and the traced
 2977 process do not share the same programming environment (see *c99*, Programming Environments
 2978 in the Shell and Utilities volume of IEEE Std 1003.1-200x).

2979 The main purpose of this option is to define a complete set of functions and concepts that allow
 2980 a conforming application to be traced from creation to termination, whatever its realtime
 2981 constraints and properties.

2982 **2.11.1 Tracing Data Definitions**2983 **2.11.1.1 Structures**

2984 The **<trace.h>** header shall define the *posix_trace_status_info* and *posix_trace_event_info* structures
 2985 described below. Implementations may add extensions to these structures.

2986 **posix_trace_status_info Structure**

2987 To facilitate control of a trace stream, information about the current state of an active trace
 2988 stream can be obtained dynamically. This structure is returned by a call to the
 2989 *posix_trace_get_status()* function.

2990 The **posix_trace_status_info** structure defined in **<trace.h>** shall contain at least the following
 2991 members:

Member Type	Member Name	Description
int	<i>posix_stream_status</i>	The operating mode of the trace stream.
int	<i>posix_stream_full_status</i>	The full status of the trace stream.
int	<i>posix_stream_overrun_status</i>	Indicates whether trace events were lost in the trace stream.

2997 If the Trace Log option is supported in addition to the Trace option, the **posix_trace_status_info**
 2998 structure defined in **<trace.h>** shall contain at least the following additional members:

Member Type	Member Name	Description
int	<i>posix_stream_flush_status</i>	Indicates whether a flush is in progress.
int	<i>posix_stream_flush_error</i>	Indicates whether any error occurred during the last flush operation.
int	<i>posix_log_overrun_status</i>	Indicates whether trace events were lost in the trace log.
int	<i>posix_log_full_status</i>	The full status of the trace log.

3006 The *posix_stream_status* member indicates the operating mode of the trace stream and shall have
 3007 one of the following values defined by manifest constants in the **<trace.h>** header:

3008 POSIX_TRACE_RUNNING

3009 Tracing is in progress; that is, the trace stream is accepting trace events.

3010 POSIX_TRACE_SUSPENDED

3011 The trace stream is not accepting trace events. The tracing operation has not yet started or
 3012 has stopped, either following a *posix_trace_stop()* function call or because the trace resources
 3013 are exhausted.

3014 The *posix_stream_full_status* member indicates the full status of the trace stream, and it shall have
 3015 one of the following values defined by manifest constants in the **<trace.h>** header:

3016 POSIX_TRACE_FULL

3017 The space in the trace stream for trace events is exhausted.

3018 POSIX_TRACE_NOT_FULL

3019 There is still space available in the trace stream.

3020 The combination of the *posix_stream_status* and *posix_stream_full_status* members also indicates
 3021 the actual status of the stream. The status shall be interpreted as follows:

3022 POSIX_TRACE_RUNNING and POSIX_TRACE_NOT_FULL
 3023 This status combination indicates that tracing is in progress, and there is space available for
 3024 recording more trace events.

3025 POSIX_TRACE_RUNNING and POSIX_TRACE_FULL
 3026 This status combination indicates that tracing is in progress and that the trace stream is full
 3027 of trace events. This status combination cannot occur unless the *stream-full-policy* is set to
 3028 POSIX_TRACE_LOOP. The trace stream contains trace events recorded during a moving
 3029 time window of prior trace events, and some older trace events may have been overwritten
 3030 and thus lost.

3031 POSIX_TRACE_SUSPENDED and POSIX_TRACE_NOT_FULL
 3032 This status combination indicates that tracing has not yet been started, has been stopped by
 3033 the *posix_trace_stop()* function, or has been cleared by the *posix_trace_clear()* function.

3034 POSIX_TRACE_SUSPENDED and POSIX_TRACE_FULL
 3035 This status combination indicates that tracing has been stopped by the implementation
 3036 because the *stream-full-policy* attribute was POSIX_TRACE_UNTIL_FULL and trace
 3037 resources were exhausted, or that the trace stream was stopped by the function
 3038 *posix_trace_stop()* at a time when trace resources were exhausted.

3039 The *posix_stream_overrun_status* member indicates whether trace events were lost in the trace
 3040 stream, and shall have one of the following values defined by manifest constants in the
 3041 **<trace.h>** header:

3042 POSIX_TRACE_OVERRUN
 3043 At least one trace event was lost and thus was not recorded in the trace stream.

3044 POSIX_TRACE_NO_OVERRUN
 3045 No trace events were lost.

3046 When the corresponding trace stream is created, the *posix_stream_overrun_status* member shall be
 3047 set to POSIX_TRACE_NO_OVERRUN.

3048 Whenever an overrun occurs, the *posix_stream_overrun_status* member shall be set to
 3049 POSIX_TRACE_OVERRUN.

3050 An overrun occurs when:

- 3051 • The policy is POSIX_TRACE_LOOP and a recorded trace event is overwritten.
- 3052 • The policy is POSIX_TRACE_UNTIL_FULL and the trace stream is full when a trace event
 3053 is generated.
- 3054 • If the Trace Log option is supported, the policy is POSIX_TRACE_FLUSH and at least one
 3055 trace event is lost while flushing the trace stream to the trace log.

3056 The *posix_stream_overrun_status* member is reset to zero after its value is read.

3057 If the Trace Log option is supported in addition to the Trace option, the *posix_stream_flush_status*,
 3058 *posix_stream_flush_error*, *posix_log_overrun_status*, and *posix_log_full_status* members are defined
 3059 as follows; otherwise, they are undefined.

3060 The *posix_stream_flush_status* member indicates whether a flush operation is being performed
 3061 and shall have one of the following values defined by manifest constants in the header
 3062 **<trace.h>**:

3063 POSIX_TRACE_FLUSHING
 3064 The trace stream is currently being flushed to the trace log.

3065 POSIX_TRACE_NOT_FLUSHING

3066 No flush operation is in progress.

3067 The *posix_stream_flush_status* member shall be set to POSIX_TRACE_FLUSHING if a flush
3068 operation is in progress either due to a call to the *posix_trace_flush()* function (explicit or caused
3069 by a trace stream shutdown operation) or because the trace stream has become full with the
3070 *stream-full-policy* attribute set to POSIX_TRACE_FLUSH. The *posix_stream_flush_status* member
3071 shall be set to POSIX_TRACE_NOT_FLUSHING if no flush operation is in progress.

3072 The *posix_stream_flush_error* member shall be set to zero if no error occurred during flushing. If
3073 an error occurred during a previous flushing operation, the *posix_stream_flush_error* member
3074 shall be set to the value of the first error that occurred. If more than one error occurs while
3075 flushing, error values after the first shall be discarded. The *posix_stream_flush_error* member is
3076 reset to zero after its value is read.

3077 The *posix_log_outrun_status* member indicates whether trace events were lost in the trace log,
3078 and shall have one of the following values defined by manifest constants in the **<trace.h>**
3079 header:

3080 POSIX_TRACE_OVERRUN

3081 At least one trace event was lost.

3082 POSIX_TRACE_NO_OVERRUN

3083 No trace events were lost.

3084 When the corresponding trace stream is created, the *posix_log_outrun_status* member shall be set
3085 to POSIX_TRACE_NO_OVERRUN. Whenever an overrun occurs, this status shall be set to
3086 POSIX_TRACE_OVERRUN. The *posix_log_outrun_status* member is reset to zero after its value
3087 is read.

3088 The *posix_log_full_status* member indicates the full status of the trace log, and it shall have one of
3089 the following values defined by manifest constants in the **<trace.h>** header:

3090 POSIX_TRACE_FULL

3091 The space in the trace log is exhausted.

3092 POSIX_TRACE_NOT_FULL

3093 There is still space available in the trace log.

3094 The *posix_log_full_status* member is only meaningful if the *log-full-policy* attribute is either
3095 POSIX_TRACE_UNTIL_FULL or POSIX_TRACE_LOOP.

3096 For an active trace stream without log, that is created by the *posix_trace_create()* function, the
3097 *posix_log_outrun_status* member shall be set to POSIX_TRACE_NO_OVERRUN and the
3098 *posix_log_full_status* member shall be set to POSIX_TRACE_NOT_FULL.

3099 **posix_trace_event_info** Structure

3100 The trace event structure **posix_trace_event_info** contains the information for one recorded
3101 trace event. This structure is returned by the set of functions *posix_trace_getnext_event()*,
3102 *posix_trace_timedgetnext_event()*, and *posix_trace_trygetnext_event()*.

3103 The **posix_trace_event_info** structure defined in **<trace.h>** shall contain at least the following
3104 members:

Member Type	Member Name	Description
trace_event_id_t	<i>posix_event_id</i>	Trace event type identification.
pid_t	<i>posix_pid</i>	Process ID of the process that generated the trace event.
void *	<i>posix_prog_address</i>	Address at which the trace point was invoked.
int	<i>posix_truncation_status</i>	Status about the truncation of the data associated with this trace event.
struct timespec	<i>posix_timestamp</i>	Time at which the trace event was generated.

In addition, the **posix_trace_event_info** structure defined in `<trace.h>` shall contain the following additional member:

Member Type	Member Name	Description
pthread_t	<i>posix_thread_id</i>	Thread ID of the thread that generated the trace event.

The *posix_event_id* member represents the identification of the trace event type and its value is not directly defined by the user. This identification is returned by a call to one of the following functions: *posix_trace_trid_eventid_open()*, *posix_trace_eventtypelist_getnext_id()*, or *posix_trace_eventid_open()*. The name of the trace event type can be obtained by calling *posix_trace_eventid_get_name()*.

The *posix_pid* is the process identifier of the traced process which generated the trace event. If the *posix_event_id* member is one of the implementation-defined system trace events and that trace event is not associated with any process, the *posix_pid* member shall be set to zero.

For a user trace event, the *posix_prog_address* member is the process mapped address of the point at which the associated call to the *posix_trace_event()* function was made. For a system trace event, if the trace event is caused by a system service explicitly called by the application, the *posix_prog_address* member shall be the address of the process at the point where the call to that system service was made.

The *posix_truncation_status* member defines whether the data associated with a trace event has been truncated at the time the trace event was generated, or at the time the trace event was read from the trace stream, or (if the Trace Log option is supported) from the trace log (see the *event* argument from the *posix_trace_getnext_event()* function). The *posix_truncation_status* member shall have one of the following values defined by manifest constants in the `<trace.h>` header:

POSIX_TRACE_NOT_TRUNCATED

All the traced data is available.

POSIX_TRACE_TRUNCATED_RECORD

Data was truncated at the time the trace event was generated.

POSIX_TRACE_TRUNCATED_READ

Data was truncated at the time the trace event was read from a trace stream or a trace log because the reader's buffer was too small. This truncation status overrides the POSIX_TRACE_TRUNCATED_RECORD status.

The *posix_timestamp* member shall be the time at which the trace event was generated. The clock used is implementation-defined, but the resolution of this clock can be retrieved by a call to the *posix_trace_attr_getclockres()* function.

The *posix_thread_id* member is the identifier of the thread that generated the trace event. If the *posix_event_id* member is one of the implementation-defined system trace events and that trace event is not associated with any thread, the *posix_thread_id* member shall be set to zero.

3151 2.11.1.2 Trace Stream Attributes

3152 Trace streams have attributes that compose the **posix_trace_attr_t** trace stream attributes object.
 3153 This object shall contain at least the following attributes:

- 3154 • The *generation-version* attribute identifies the origin and version of the trace system.
- 3155 • The *trace-name* attribute is a character string defined by the trace controller, and that
 3156 identifies the trace stream.
- 3157 • The *creation-time* attribute represents the time of the creation of the trace stream.
- 3158 • The *clock-resolution* attribute defines the clock resolution of the clock used to generate
 3159 timestamps.
- 3160 • The *stream-min-size* attribute defines the minimum size in bytes of the trace stream strictly
 3161 reserved for the trace events.
- 3162 • The *stream-full-policy* attribute defines the policy followed when the trace stream is full; its
 3163 value is `POSIX_TRACE_LOOP`, `POSIX_TRACE_UNTIL_FULL`, or `POSIX_TRACE_FLUSH`.
- 3164 • The *max-data-size* attribute defines the maximum record size in bytes of a trace event.

3165 In addition, if the Trace option and the Trace Inherit option are both supported, the
 3166 **posix_trace_attr_t** trace stream creation attributes object shall contain at least the following
 3167 attributes:

- 3168 • The *inheritance* attribute specifies whether a newly created trace stream will inherit tracing
 3169 in its parent's process trace stream. It is either `POSIX_TRACE_INHERITED` or
 3170 `POSIX_TRACE_CLOSE_FOR_CHILD`.

3171 In addition, if the Trace option and the Trace Log option are both supported, the
 3172 **posix_trace_attr_t** trace stream creation attributes object shall contain at least the following
 3173 attribute:

- 3174 • If the file type corresponding to the trace log supports the `POSIX_TRACE_LOOP` or the
 3175 `POSIX_TRACE_UNTIL_FULL` policies, the *log-max-size* attribute defines the maximum
 3176 size in bytes of the trace log associated with an active trace stream. Other stream data—for
 3177 example, trace attribute values—shall not be included in this size.
- 3178 • The *log-full-policy* attribute defines the policy of a trace log associated with an active trace
 3179 stream to be `POSIX_TRACE_LOOP`, `POSIX_TRACE_UNTIL_FULL`, or
 3180 `POSIX_TRACE_APPEND`.

3181 **2.11.2 Trace Event Type Definitions**

3182 2.11.2.1 System Trace Event Type Definitions

3183 The following system trace event types, defined in the **<trace.h>** header, track the invocation of
 3184 the trace operations:

- 3185 • `POSIX_TRACE_START` shall be associated with a trace start operation.
- 3186 • `POSIX_TRACE_STOP` shall be associated with a trace stop operation.
- 3187 • If the Trace Event Filter option is supported, `POSIX_TRACE_FILTER` shall be associated
 3188 with a trace event type filter change operation.

3189 The following system trace event types, defined in the **<trace.h>** header, report operational trace
 3190 events:

- 3191 • POSIX_TRACE_OVERFLOW shall mark the beginning of a trace overflow condition.
- 3192 • POSIX_TRACE_RESUME shall mark the end of a trace overflow condition.
- 3193 • If the Trace Log option is supported, POSIX_TRACE_FLUSH_START shall mark the
- 3194 beginning of a flush operation.
- 3195 • If the Trace Log option is supported, POSIX_TRACE_FLUSH_STOP shall mark the end of
- 3196 a flush operation.
- 3197 • If an implementation-defined trace error condition is reported, it shall be marked
- 3198 POSIX_TRACE_ERROR.

3199 The interpretation of a trace stream or a trace log by a trace analyzer process relies on the
 3200 information recorded for each trace event, and also on system trace events that indicate the
 3201 invocation of trace control operations and trace system operational trace events.

3202 The POSIX_TRACE_START and POSIX_TRACE_STOP trace events specify the time windows
 3203 during which the trace stream is running.

- 3204 • The POSIX_TRACE_STOP trace event with an associated data that is equal to zero
- 3205 indicates a call of the function *posix_trace_stop()*.

- 3206 • The POSIX_TRACE_STOP trace event with an associated data that is different from zero
- 3207 indicates an automatic stop of the trace stream (see *posix_trace_attr_getstreamfullpolicy()*).

3208 The POSIX_TRACE_FILTER trace event indicates that a trace event type filter value changed
 3209 while the trace stream was running.

3210 The POSIX_TRACE_ERROR serves to inform the analyzer process that an implementation-
 3211 defined internal error of the trace system occurred.

3212 The POSIX_TRACE_OVERFLOW trace event shall be reported with a timestamp equal to the
 3213 timestamp of the first trace event overwritten. This is an indication that some generated trace
 3214 events have been lost.

3215 The POSIX_TRACE_RESUME trace event shall be reported with a timestamp equal to the
 3216 timestamp of the first valid trace event reported after the overflow condition ends and shall be
 3217 reported before this first valid trace event. This is an indication that the trace system is reliably
 3218 recording trace events after an overflow condition.

3219 Each of these trace event types shall be defined by a constant trace event name and a
 3220 **trace_event_id_t** constant; trace event data is associated with some of these trace events.

3221 If the Trace option is supported and the Trace Event Filter option and the Trace Log option are
 3222 not supported, the following predefined system trace events in [Table 2-3](#) shall be defined:

3223

Table 2-3 Trace Option: System Trace Events

3224

3225

3226

3227

3228

3229

3230

3231

3232

Event Name	Constant	Associated Data
		Data Type
posix_trace_error	POSIX_TRACE_ERROR	error
		int
posix_trace_start	POSIX_TRACE_START	None.
posix_trace_stop	POSIX_TRACE_STOP	auto
		int
posix_trace_overflow	POSIX_TRACE_OVERFLOW	None.
posix_trace_resume	POSIX_TRACE_RESUME	None.

3233

3234

3235

If the Trace option and the Trace Event Filter option are both supported, and if the Trace Log option is not supported, the following predefined system trace events in [Table 2-4](#) shall be defined:

3236

Table 2-4 Trace and Trace Event Filter Options: System Trace Events

3237

3238

3239

3240

3241

3242

3243

3244

3245

3246

3247

3248

3249

Event Name	Constant	Associated Data
		Data Type
posix_trace_error	POSIX_TRACE_ERROR	error
		int
posix_trace_start	POSIX_TRACE_START	event_filter
		trace_event_set_t
posix_trace_stop	POSIX_TRACE_STOP	auto
		int
posix_trace_filter	POSIX_TRACE_FILTER	old_event_filter
		new_event_filter
		trace_event_set_t
posix_trace_overflow	POSIX_TRACE_OVERFLOW	None.
posix_trace_resume	POSIX_TRACE_RESUME	None.

3250

3251

3252

If the Trace option and the Trace Log option are both supported, and if the Trace Event Filter option is not supported, the following predefined system trace events in [Table 2-5](#) shall be defined:

Table 2-5 Trace and Trace Log Options: System Trace Events

Event Name	Constant	Associated Data
		Data Type
posix_trace_error	POSIX_TRACE_ERROR	error int
posix_trace_start	POSIX_TRACE_START	None.
posix_trace_stop	POSIX_TRACE_STOP	auto int
posix_trace_overflow	POSIX_TRACE_OVERFLOW	None.
posix_trace_resume	POSIX_TRACE_RESUME	None.
posix_trace_flush_start	POSIX_TRACE_FLUSH_START	None.
posix_trace_flush_stop	POSIX_TRACE_FLUSH_STOP	None.

If the Trace option, the Trace Event Filter option, and the Trace Log option are all supported, the following predefined system trace events in [Table 2-6](#) shall be defined:

Table 2-6 Trace, Trace Log, and Trace Event Filter Options: System Trace Events

Event Name	Constant	Associated Data
		Data Type
posix_trace_error	POSIX_TRACE_ERROR	error int
posix_trace_start	POSIX_TRACE_START	event_filter trace_event_set_t
posix_trace_stop	POSIX_TRACE_STOP	auto int
posix_trace_filter	POSIX_TRACE_FILTER	old_event_filter new_event_filter trace_event_set_t
posix_trace_overflow	POSIX_TRACE_OVERFLOW	None.
posix_trace_resume	POSIX_TRACE_RESUME	None.
posix_trace_flush_start	POSIX_TRACE_FLUSH_START	None.
posix_trace_flush_stop	POSIX_TRACE_FLUSH_STOP	None.

2.11.2.2 User Trace Event Type Definitions

The user trace event `POSIX_TRACE_UNNAMED_USEREVENT` is defined in the `<trace.h>` header. If the limit of per-process user trace event names represented by `{TRACE_USER_EVENT_MAX}` has already been reached, this predefined user event shall be returned when the application tries to register more events than allowed. The data associated with this trace event is application-defined.

The following predefined user trace event in [Table 2-7](#) shall be defined:

3290

Table 2-7 Trace Option: User Trace Event

3291

3292

Event Name	Constant
posix_trace_unnamed_userevent	POSIX_TRACE_UNNAMED_USEREVENT

3293

2.11.3 Trace Functions

3294

3295

3296

3297

3298

3299

The trace interface is built and structured to improve portability through use of trace data of opaque type. The object-oriented approach for the manipulation of trace attributes and trace event type identifiers requires definition of many constructor and selector functions which operate on these opaque types. Also, the trace interface must support several different tracing roles. To facilitate reading the trace interface, the trace functions are grouped into small functional sets supporting the three different roles:

3300

3301

- A trace controller process requires functions to set up and customize all the resources needed to run a trace stream, including:

3302

- Attribute initialization and destruction (*posix_trace_attr_init()*)

3303

- Identification information manipulation (*posix_trace_attr_getgenversion()*)

3304

- Trace system behavior modification (*posix_trace_attr_getinherited()*)

3305

- Trace stream and trace log size set (*posix_trace_attr_getmaxusereventsize()*)

3306

- Trace stream creation, flush, and shutdown (*posix_trace_create()*)

3307

- Trace stream and trace log clear (*posix_trace_clear()*)

3308

- Trace event type identifier manipulation (*posix_trace_trid_eventid_open()*)

3309

- Trace event type identifier list exploration (*posix_trace_eventtypelist_getnext_id()*)

3310

- Trace event type set manipulation (*posix_trace_eventset_empty()*)

3311

- Trace event type filter set (*posix_trace_set_filter()*)

3312

- Trace stream start and stop (*posix_trace_start()*)

3313

- Trace stream information and status read (*posix_trace_get_attr()*)

3314

- A traced process requires functions to instrument trace points:

3315

- Trace event type identifiers definition and trace points insertion (*posix_trace_event()*)

3316

3317

- A trace analyzer process requires functions to retrieve information from a trace stream and trace log:

3318

- Identification information read (*posix_trace_attr_getgenversion()*)

3319

- Trace system behavior information read (*posix_trace_attr_getinherited()*)

3320

- Trace stream and trace log size get (*posix_trace_attr_getmaxusereventsize()*)

3321

- Trace event type identifier manipulation (*posix_trace_trid_eventid_open()*)

3322

- Trace event type identifier list exploration (*posix_trace_eventtypelist_getnext_id()*)

3323

- Trace log open, rewind, and close (*posix_trace_open()*)

3324

- Trace stream information and status read (*posix_trace_get_attr()*)

3325 — Trace event read (*posix_trace_getnext_event()*)

3326 2.12 Data Types

3327 2.12.1 Defined Types

3328 All of the data types used by various functions are defined by the implementation. The
 3329 following table describes some of these types. Other types referenced in the description of a
 3330 function, not mentioned here, can be found in the appropriate header for that function.

3331	Defined Type	Description
3332	cc_t	Type used for terminal special characters.
3333	clock_t	Integer or real-floating type used for processor times, as defined in the ISO C standard.
3334		
3335	clockid_t	Used for clock ID type in some timer functions.
3336	dev_t	Arithmetic type used for device numbers.
3337	DIR	Type representing a directory stream.
3338	div_t	Structure type returned by the <i>div()</i> function.
3339	FILE	Structure containing information about a file.
3340	glob_t	Structure type used in pathname pattern matching.
3341	fpos_t	Type containing all information needed to specify uniquely every position within a file.
3342		
3343	gid_t	Integer type used for group IDs.
3344	iconv_t	Type used for conversion descriptors.
3345	id_t	Integer type used as a general identifier; can be used to contain at least the largest of a pid_t , uid_t , or gid_t .
3346		
3347	ino_t	Unsigned integer type used for file serial numbers.
3348	key_t	Arithmetic type used for XSI interprocess communication.
3349	ldiv_t	Structure type returned by the <i>ldiv()</i> function.
3350	mode_t	Integer type used for file attributes.
3351	mqd_t	Used for message queue descriptors.
3352	nfds_t	Integer type used for the number of file descriptors.
3353	nlink_t	Integer type used for link counts.
3354	off_t	Signed integer type used for file sizes.
3355	pid_t	Signed integer type used for process and process group IDs.
3356	pthread_attr_t	Used to identify a thread attribute object.
3357	pthread_cond_t	Used for condition variables.
3358	pthread_condattr_t	Used to identify a condition attribute object.
3359	pthread_key_t	Used for thread-specific data keys.
3360	pthread_mutex_t	Used for mutexes.
3361	pthread_mutexattr_t	Used to identify a mutex attribute object.
3362	pthread_once_t	Used for dynamic package initialization.
3363	pthread_rwlock_t	Used for read-write locks.
3364	pthread_rwlockattr_t	Used for read-write lock attributes.
3365	pthread_t	Used to identify a thread.
3366	ptrdiff_t	Signed integer type of the result of subtracting two pointers.
3367	regex_t	Structure type used in regular expression matching.
3368	regmatch_t	Structure type used in regular expression matching.
3369	rlim_t	Unsigned integer type used for limit values, to which objects of

Defined Type	Description
3370	
3371	type int and off_t can be cast without loss of value.
3372	sem_t Type used in performing semaphore operations.
3373	sig_atomic_t Integer type of an object that can be accessed as an atomic
3374	entity, even in the presence of asynchronous interrupts.
3375	sigset_t Integer or structure type of an object used to represent sets
3376	of signals.
3377	size_t Unsigned integer type used for size of objects.
3378	speed_t Type used for terminal baud rates.
3379	ssize_t Signed integer type used for a count of bytes or an error
3380	indication.
3381	suseconds_t Signed integer type used for time in microseconds.
3382	tflag_t Type used for terminal modes.
3383	time_t Integer or real-floating type used for time in seconds, as defined in
3384	the ISO C standard.
3385	timer_t Used for timer ID returned by the <i>timer_create()</i> function.
3386	uid_t Integer type used for user IDs.
3387	va_list Type used for traversing variable argument lists.
3388	wchar_t Integer type whose range of values can represent distinct codes for
3389	all members of the largest extended character set specified by the
3390	supported locales.
3391	wctype_t Scalar type which represents a character class descriptor.
3392	wint_t Integer type capable of storing any valid value of wchar_t or
3393	WEOF.
3394	wordexp_t Structure type used in word expansion.

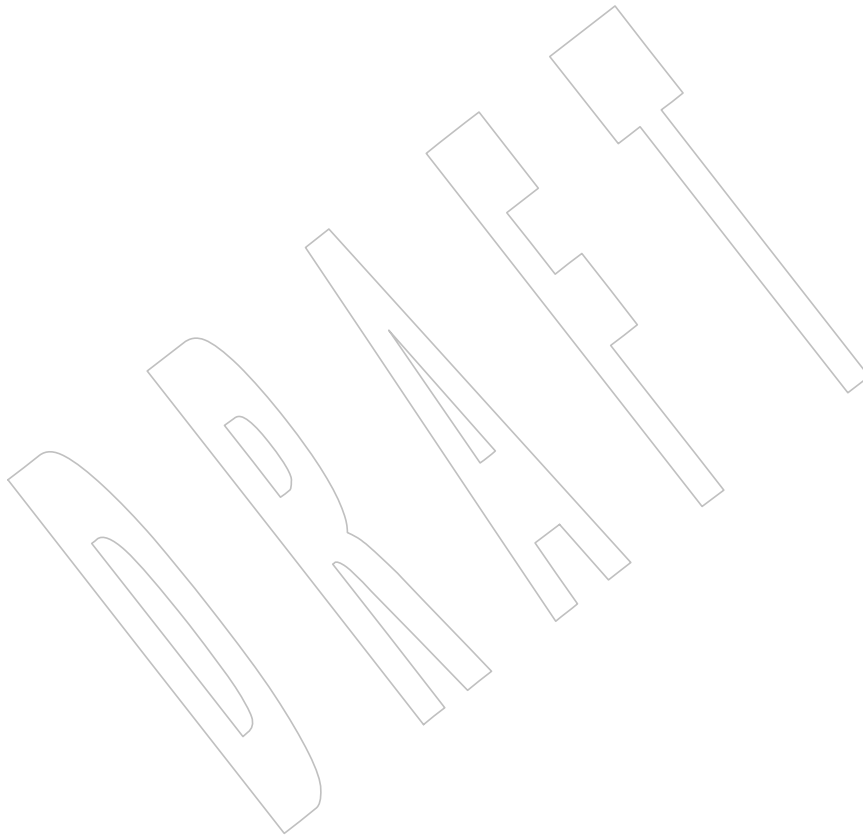
2.12.2 The char Type

The type **char** is defined as a single byte; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 3, Definitions (Byte and Character).

2.12.3 Pointer Types

Conforming implementations shall support conversion of pointers of any type to **void *** and back without loss of information.


Note: The ISO C standard does not require this, but it is required for POSIX conformance.



3402

Chapter 3

3403

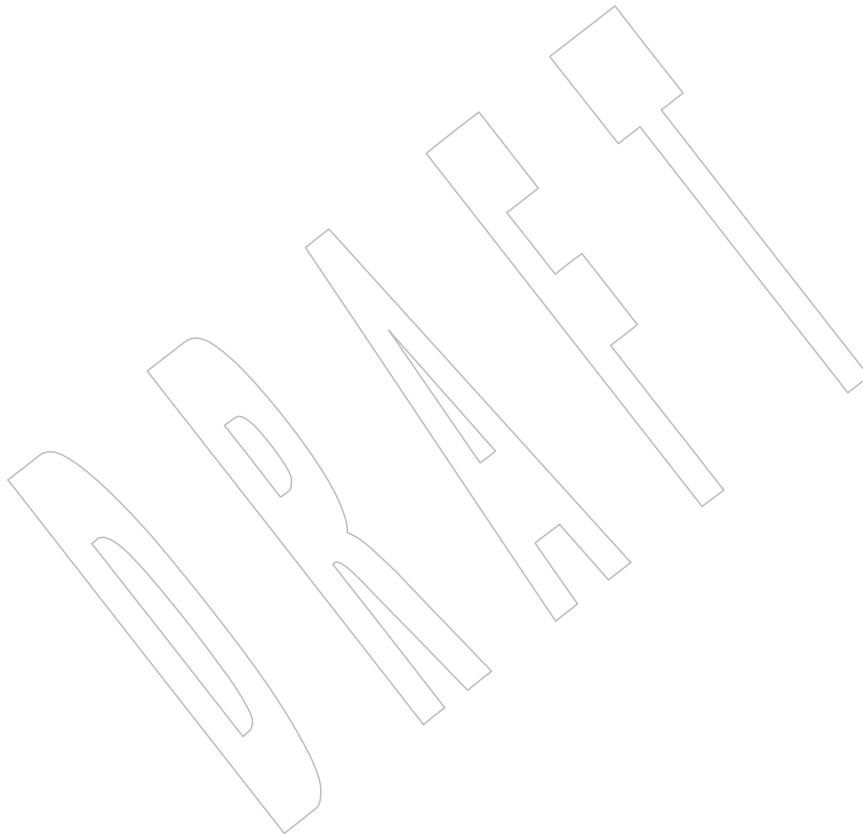


System Interfaces

3404

This chapter describes the functions, macros, and external variables to support applications portability at the C-language source level.

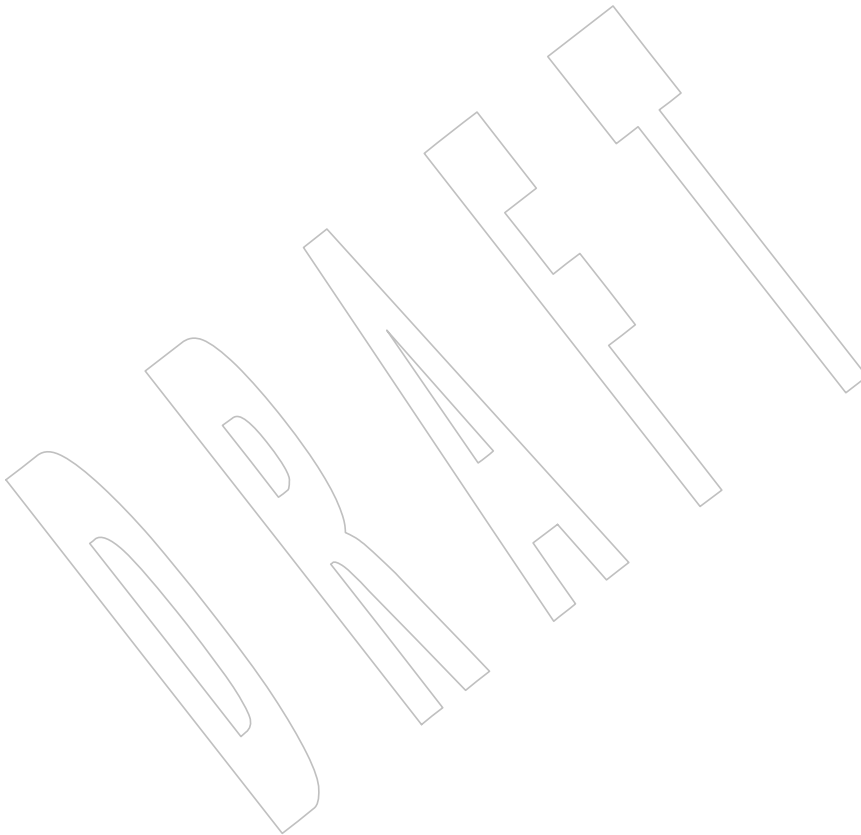
3405



3406 **NAME**
3407 FD_CLR — macros for synchronous I/O multiplexing

3408 **SYNOPSIS**
3409 #include <sys/time.h>
3410 FD_CLR(int *fd*, fd_set **fdset*);
3411 FD_ISSET(int *fd*, fd_set **fdset*);
3412 FD_SET(int *fd*, fd_set **fdset*);
3413 FD_ZERO(fd_set **fdset*);

3414 **DESCRIPTION**
3415 Refer to *pselect()*.



3416 **NAME**3417 `_Exit, _exit` — terminate a process3418 **SYNOPSIS**3419 `#include <stdlib.h>`3420 `void _Exit(int status);`3421 `#include <unistd.h>`3422 `void _exit(int status);`3423 **DESCRIPTION**3424 CX For `_Exit()`: The functionality described on this reference page is aligned with the ISO C
3425 standard. Any conflict between the requirements described here and the ISO C standard is
3426 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.3427 CX The value of *status* may be 0, EXIT_SUCCESS, EXIT_FAILURE, or any other value, though only
3428 the least significant 8 bits (that is, *status* & 0377) shall be available to a waiting parent process.3429 CX The `_Exit()` and `_exit()` functions shall be functionally equivalent.3430 CX The `_Exit()` and `_exit()` functions shall not call functions registered with `atexit()` nor any
3431 CX registered signal handlers. Open streams shall not be flushed. Whether open streams are
3432 closed (without flushing) is implementation-defined. Finally, the calling process is terminated
3433 with the consequences described below.3434 **Consequences of Process Termination**

3435 CX These functions shall terminate the calling process with the following consequences:

3436 **Note:** These consequences are all extensions to the ISO C standard and are not further CX shaded.
3437 However, functionality relating to the XSI option is shaded.3438 • All of the file descriptors, directory streams, conversion descriptors, and message catalog
3439 descriptors open in the calling process shall be closed.3440 XSI • If the parent process of the calling process is executing a `wait()` or `waitpid()`, and has
3441 neither set its SA_NOCLDWAIT flag nor set SIGCHLD to SIG_IGN, it shall be notified of
3442 termination of the calling process and the low-order eight bits (that is, bits 0377) of *status*
3443 shall be made available to it. If the parent is not waiting, the child's status shall be made
3444 available to it when the parent subsequently executes `wait()` or `waitpid()`.3445 The semantics of the `waitid()` function shall be equivalent to `wait()`.3446 XSI • If the parent process of the calling process is not executing a `wait()` or `waitpid()`, and has
3447 neither set its SA_NOCLDWAIT flag nor set SIGCHLD to SIG_IGN, the calling process
3448 shall be transformed into a *zombie process*. A *zombie process* is an inactive process and it
3449 shall be deleted at some later time when its parent process executes `wait()` or `waitpid()`.3450 XSI The semantics of the `waitid()` function shall be equivalent to `wait()`.3451 • Termination of a process does not directly terminate its children. The sending of a SIGHUP
3452 signal as described below indirectly terminates children in some circumstances.

3453 • Either:

3454 If the implementation supports the SIGCHLD signal, a SIGCHLD shall be sent to the
3455 parent process.

3456 Or:

_Exit()*System Interfaces*

- 3457 XSI If the parent process has set its SA_NOCLDWAIT flag, or set SIGCHLD to SIG_IGN, the
 3458 status shall be discarded, and the lifetime of the calling process shall end immediately. If
 3459 SA_NOCLDWAIT is set, it is implementation-defined whether a SIGCHLD signal is sent to
 3460 the parent process.
- The parent process ID of all of the existing child processes and zombie processes of the
 3461 calling process shall be set to the process ID of an implementation-defined system process.
 3462 That is, these processes shall be inherited by a special system process.
 3463
- 3464 XSI • Each attached shared-memory segment is detached and the value of *shm_nattch* (see
 3465 *shmget()*) in the data structure associated with its shared memory ID shall be decremented
 3466 by 1.
- 3467 XSI • For each semaphore for which the calling process has set a *semadj* value (see *semop()*), that
 3468 value shall be added to the *semval* of the specified semaphore.
- If the process is a controlling process, the SIGHUP signal shall be sent to each process in
 3469 the foreground process group of the controlling terminal belonging to the calling process.
 3470
 - If the process is a controlling process, the controlling terminal associated with the session
 3471 shall be disassociated from the session, allowing it to be acquired by a new controlling
 3472 process.
 3473
 - If the exit of the process causes a process group to become orphaned, and if any member of
 3474 the newly-orphaned process group is stopped, then a SIGHUP signal followed by a
 3475 SIGCONT signal shall be sent to each process in the newly-orphaned process group.
 3476
 - All open named semaphores in the calling process shall be closed as if by appropriate calls
 3477 to *sem_close()*.
 3478
- 3479 ML • Any memory locks established by the process via calls to *mlockall()* or *mlock()* shall be
 3480 removed. If locked pages in the address space of the calling process are also mapped into
 3481 the address spaces of other processes and are locked by those processes, the locks
 3482 established by the other processes shall be unaffected by the call by this process to *_Exit()*
 3483 or *_exit()*.
- Memory mappings that were created in the process shall be unmapped before the process
 3484 is destroyed.
 3485
- 3486 TYM • Any blocks of typed memory that were mapped in the calling process shall be unmapped,
 3487 as if *munmap()* was implicitly called to unmap them.
- 3488 MSG • All open message queue descriptors in the calling process shall be closed as if by
 3489 appropriate calls to *mq_close()*.
- Any outstanding cancelable asynchronous I/O operations may be canceled. Those
 3490 asynchronous I/O operations that are not canceled shall complete as if the *_Exit()* or
 3491 *_exit()* operation had not yet occurred, but any associated signal notifications shall be
 3492 suppressed. The *_Exit()* or *_exit()* operation may block awaiting such I/O completion.
 3493 Whether any I/O is canceled, and which I/O may be canceled upon *_Exit()* or *_exit()*, is
 3494 implementation-defined.
 3495
 - Threads terminated by a call to *_Exit()* or *_exit()* shall not invoke their cancellation
 3496 cleanup handlers or per-thread data destructors.
 3497
- 3498 OB TRC • If the calling process is a trace controller process, any trace streams that were created by
 3499 the calling process shall be shut down as described by the *posix_trace_shutdown()* function,
 3500 and mapping of trace event names to trace event type identifiers of any process built for
 3501 these trace streams may be deallocated.

3502 **RETURN VALUE**

3503 These functions do not return.

3504 **ERRORS**

3505 No errors are defined.

3506 **EXAMPLES**

3507 None.

3508 **APPLICATION USAGE**3509 Normally applications should use *exit()* rather than *_Exit()* or *_exit()*.3510 **RATIONALE**3511 **Process Termination**

3512 Early proposals drew a distinction between normal and abnormal process termination.
 3513 Abnormal termination was caused only by certain signals and resulted in implementation-
 3514 defined “actions”, as discussed below. Subsequent proposals distinguished three types of
 3515 termination: *normal termination* (as in the current specification), *simple abnormal termination*, and
 3516 *abnormal termination with actions*. Again the distinction between the two types of abnormal
 3517 termination was that they were caused by different signals and that implementation-defined
 3518 actions would result in the latter case. Given that these actions were completely implementation-
 3519 defined, the early proposals were only saying when the actions could occur and how their
 3520 occurrence could be detected, but not what they were. This was of little or no use to conforming
 3521 applications, and thus the distinction is not made in this volume of IEEE Std 1003.1-200x.

3522 The implementation-defined actions usually include, in most historical implementations, the
 3523 creation of a file named **core** in the current working directory of the process. This file contains an
 3524 image of the memory of the process, together with descriptive information about the process,
 3525 perhaps sufficient to reconstruct the state of the process at the receipt of the signal.

3526 There is a potential security problem in creating a **core** file if the process was set-user-ID and the
 3527 current user is not the owner of the program, if the process was set-group-ID and none of the
 3528 user’s groups match the group of the program, or if the user does not have permission to write
 3529 in the current directory. In this situation, an implementation either should not create a **core** file
 3530 or should make it unreadable by the user.

3531 Despite the silence of this volume of IEEE Std 1003.1-200x on this feature, applications are
 3532 advised not to create files named **core** because of potential conflicts in many implementations.
 3533 Some implementations use a name other than **core** for the file; for example, by appending the
 3534 process ID to the filename.

3535 **Terminating a Process**

3536 It is important that the consequences of process termination as described occur regardless of
 3537 whether the process called *_exit()* (perhaps indirectly through *exit()*) or instead was terminated
 3538 due to a signal or for some other reason. Note that in the specific case of *exit()* this means that
 3539 the *status* argument to *exit()* is treated in the same way as the *status* argument to *_exit()*.

3540 A language other than C may have other termination primitives than the C-language *exit()*
 3541 function, and programs written in such a language should use its native termination primitives,
 3542 but those should have as part of their function the behavior of *_exit()* as described.
 3543 Implementations in languages other than C are outside the scope of this version of this volume
 3544 of IEEE Std 1003.1-200x, however.

3545 As required by the ISO C standard, using **return** from *main()* has the same behavior (other than
 3546 with respect to language scope issues) as calling *exit()* with the returned value. Reaching the end
 3547 of the *main()* function has the same behavior as calling *exit(0)*.

3548 A value of zero (or `EXIT_SUCCESS`, which is required to be zero) for the argument *status*
3549 conventionally indicates successful termination. This corresponds to the specification for `exit()`
3550 in the ISO C standard. The convention is followed by utilities such as *make* and various shells,
3551 which interpret a zero status from a child process as success. For this reason, applications should
3552 not call `exit(0)` or `_exit(0)` when they terminate unsuccessfully; for example, in signal-catching
3553 functions.

3554 Historically, the implementation-defined process that inherits children whose parents have
3555 terminated without waiting on them is called *init* and has a process ID of 1.

3556 The sending of a `SIGHUP` to the foreground process group when a controlling process
3557 terminates corresponds to somewhat different historical implementations. In System V, the
3558 kernel sends a `SIGHUP` on termination of (essentially) a controlling process. In 4.2 BSD, the
3559 kernel does not send `SIGHUP` in a case like this, but the termination of a controlling process is
3560 usually noticed by a system daemon, which arranges to send a `SIGHUP` to the foreground
3561 process group with the `vhangup()` function. However, in 4.2 BSD, due to the behavior of the
3562 shells that support job control, the controlling process is usually a shell with no other processes
3563 in its process group. Thus, a change to make `_exit()` behave this way in such systems should not
3564 cause problems with existing applications.

3565 The termination of a process may cause a process group to become orphaned in either of two
3566 ways. The connection of a process group to its parent(s) outside of the group depends on both
3567 the parents and their children. Thus, a process group may be orphaned by the termination of the
3568 last connecting parent process outside of the group or by the termination of the last direct
3569 descendant of the parent process(es). In either case, if the termination of a process causes a
3570 process group to become orphaned, processes within the group are disconnected from their job
3571 control shell, which no longer has any information on the existence of the process group.
3572 Stopped processes within the group would languish forever. In order to avoid this problem,
3573 newly orphaned process groups that contain stopped processes are sent a `SIGHUP` signal and a
3574 `SIGCONT` signal to indicate that they have been disconnected from their session. The `SIGHUP`
3575 signal causes the process group members to terminate unless they are catching or ignoring
3576 `SIGHUP`. Under most circumstances, all of the members of the process group are stopped if any
3577 of them are stopped.

3578 The action of sending a `SIGHUP` and a `SIGCONT` signal to members of a newly orphaned
3579 process group is similar to the action of 4.2 BSD, which sends `SIGHUP` and `SIGCONT` to each
3580 stopped child of an exiting process. If such children exit in response to the `SIGHUP`, any
3581 additional descendants receive similar treatment at that time. In this volume of
3582 IEEE Std 1003.1-200x, the signals are sent to the entire process group at the same time. Also, in
3583 this volume of IEEE Std 1003.1-200x, but not in 4.2 BSD, stopped processes may be orphaned,
3584 but may be members of a process group that is not orphaned; therefore, the action taken at
3585 `_exit()` must consider processes other than child processes.

3586 It is possible for a process group to be orphaned by a call to `setpgid()` or `setsid()`, as well as by
3587 process termination. This volume of IEEE Std 1003.1-200x does not require sending `SIGHUP` and
3588 `SIGCONT` in those cases, because, unlike process termination, those cases are not caused
3589 accidentally by applications that are unaware of job control. An implementation can choose to
3590 send `SIGHUP` and `SIGCONT` in those cases as an extension; such an extension must be
3591 documented as required in **<signal.h>**.

3592 The ISO/IEC 9899:1999 standard adds the `_Exit()` function that results in immediate program
3593 termination without triggering signals or `atexit()`-registered functions. In IEEE Std 1003.1-200x,
3594 this is equivalent to the `_exit()` function.

3595
3596
3597
3598
3599
3600
3601
3602
3603
3604
3605
3606
3607
3608
3609
3610
3611
3612
3613
3614
3615
3616
3617
3618
3619
3620
3621
3622
3623
3624
3625

FUTURE DIRECTIONS

None.

SEE ALSO

atexit(), *exit()*, *mlock()*, *mlockall()*, *mq_close()*, *munmap()*, *posix_trace_shutdown()*, *sem_close()*, *semop()*, *setpgid()*, *setsid()*, *shmget()*, *wait()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdlib.h>`, `<unistd.h>`

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Interactions with the SA_NOCLDWAIT flag and SIGCHLD signal are further clarified.

The values of *status* from *exit()* are better described.

Issue 6

Extensions beyond the ISO C standard are marked.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for typed memory.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The *_Exit()* function is included.
- The DESCRIPTION is updated.

The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.

References to the *wait3()* function are removed.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/16 is applied, correcting grammar in the DESCRIPTION.

Issue 7

Austin Group Interpretation 1003.1-2001 #031 is applied, separating these functions from the *exit()* function.

Austin Group Interpretation 1003.1-2001 #085 is applied, clarifying the text regarding flushing of streams and closing of temporary files.

Functionality relating to the Asynchronous Input and Output, Memory Mapped Files, and Semaphores options is moved to the Base.

3626 **NAME**3627 `_longjmp`, `_setjmp` — non-local goto3628 **SYNOPSIS**

```
3629 OB XSI #include <setjmp.h>
3630 void _longjmp(jmp_buf env, int val);
3631 int _setjmp(jmp_buf env);
```

3632 **DESCRIPTION**

3633 The `_longjmp()` and `_setjmp()` functions shall be equivalent to `longjmp()` and `setjmp()`,
 3634 respectively, with the additional restriction that `_longjmp()` and `_setjmp()` shall not manipulate
 3635 the signal mask.

3636 If `_longjmp()` is called even though `env` was never initialized by a call to `_setjmp()`, or when the
 3637 last such call was in a function that has since returned, the results are undefined.

3638 **RETURN VALUE**3639 Refer to `longjmp()` and `setjmp()`.3640 **ERRORS**

3641 No errors are defined.

3642 **EXAMPLES**

3643 None.

3644 **APPLICATION USAGE**

3645 If `_longjmp()` is executed and the environment in which `_setjmp()` was executed no longer exists,
 3646 errors can occur. The conditions under which the environment of the `_setjmp()` no longer exists
 3647 include exiting the function that contains the `_setjmp()` call, and exiting an inner block with
 3648 temporary storage. This condition might not be detectable, in which case the `_longjmp()` occurs
 3649 and, if the environment no longer exists, the contents of the temporary storage of an inner block
 3650 are unpredictable. This condition might also cause unexpected process termination. If the
 3651 function has returned, the results are undefined.

3652 Passing `longjmp()` a pointer to a buffer not created by `setjmp()`, passing `_longjmp()` a pointer to a
 3653 buffer not created by `_setjmp()`, passing `siglongjmp()` a pointer to a buffer not created by
 3654 `sigsetjmp()`, or passing any of these three functions a buffer that has been modified by the user
 3655 can cause all the problems listed above, and more.

3656 The `_longjmp()` and `_setjmp()` functions are included to support programs written to historical
 3657 system interfaces. New applications should use `siglongjmp()` and `sigsetjmp()` respectively.

3658 **RATIONALE**

3659 None.

3660 **FUTURE DIRECTIONS**3661 The `_longjmp()` and `_setjmp()` functions may be removed in a future version.3662 **SEE ALSO**

3663 `longjmp()`, `setjmp()`, `siglongjmp()`, `sigsetjmp()`, the Base Definitions volume of
 3664 IEEE Std 1003.1-200x, `<setjmp.h>`

3665 **CHANGE HISTORY**

3666 First released in Issue 4, Version 2.

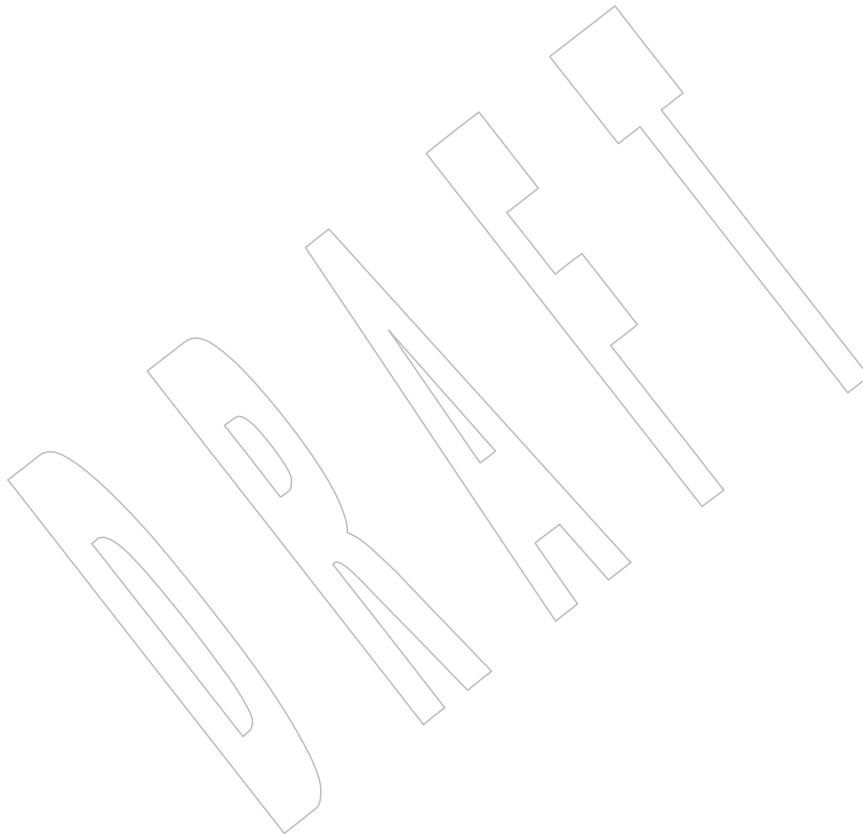
3667

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 7

The *_longjmp()* and *_setjmp()* functions are marked obsolescent.



3671 **NAME**
 3672 _toupper — transliterate uppercase characters to lowercase

3673 **SYNOPSIS**

3674 OB XSI #include <ctype.h>
 3675 int _tolower(int c);

3676 **DESCRIPTION**

3677 The *_tolower()* macro shall be equivalent to *tolower(c)* except that the application shall ensure
 3678 that the argument *c* is an uppercase letter.

3679 **RETURN VALUE**

3680 Upon successful completion, *_tolower()* shall return the lowercase letter corresponding to the
 3681 argument passed.

3682 **ERRORS**

3683 No errors are defined.

3684 **EXAMPLES**

3685 None.

3686 **APPLICATION USAGE**

3687 Applications should use the *tolower()* function instead of the obsolescent *_tolower()* function.

3688 **RATIONALE**

3689 None.

3690 **FUTURE DIRECTIONS**

3691 The *_tolower()* function may be removed in a future version.

3692 **SEE ALSO**

3693 *tolower()*, *isupper()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale,
 3694 <ctype.h>

3695 **CHANGE HISTORY**

3696 First released in Issue 1. Derived from Issue 1 of the SVID.

3697 **Issue 6**

3698 The normative text is updated to avoid use of the term “must” for application requirements.

3699 **Issue 7**

3700 The *_tolower()* function is marked obsolescent.

3701 **NAME**

3702 _toupper — transliterate lowercase characters to uppercase

3703 **SYNOPSIS**3704 OB XSI #include <ctype.h>
3705 int _toupper(int c);3706 **DESCRIPTION**3707 The *_toupper()* macro shall be equivalent to *toupper()* except that the application shall ensure
3708 that the argument *c* is a lowercase letter.3709 **RETURN VALUE**3710 Upon successful completion, *_toupper()* shall return the uppercase letter corresponding to the
3711 argument passed.3712 **ERRORS**

3713 No errors are defined.

3714 **EXAMPLES**

3715 None.

3716 **APPLICATION USAGE**3717 Applications should use the *toupper()* function instead of the obsolescent *_toupper()* function.3718 **RATIONALE**

3719 None.

3720 **FUTURE DIRECTIONS**3721 The *_toupper()* function may be removed in a future version.3722 **SEE ALSO**3723 *islower()*, *toupper()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale,
3724 <ctype.h>3725 **CHANGE HISTORY**

3726 First released in Issue 1. Derived from Issue 1 of the SVID.

3727 **Issue 6**

3728 The normative text is updated to avoid use of the term “must” for application requirements.

3729 **Issue 7**3730 The *_toupper()* function is marked obsolescent.

3731 **NAME**

3732 a64l, l64a — convert between a 32-bit integer and a radix-64 ASCII string

3733 **SYNOPSIS**

```
3734 XSI #include <stdlib.h>
3735 long a64l(const char *s);
3736 char *l64a(long value);
```

3737 **DESCRIPTION**

3738 These functions maintain numbers stored in radix-64 ASCII characters. This is a notation by
 3739 which 32-bit integers can be represented by up to six characters; each character represents a digit
 3740 in radix-64 notation. If the type **long** contains more than 32 bits, only the low-order 32 bits shall
 3741 be used for these operations.

3742 The characters used to represent digits are '.' (dot) for 0, '/' for 1, '0' through '9' for [2,11],
 3743 'A' through 'Z' for [12,37], and 'a' through 'z' for [38,63].

3744 The *a64l()* function shall take a pointer to a radix-64 representation, in which the first digit is the
 3745 least significant, and return the corresponding **long** value. If the string pointed to by *s* contains
 3746 more than six characters, *a64l()* shall use the first six. If the first six characters of the string
 3747 contain a null terminator, *a64l()* shall use only characters preceding the null terminator. The
 3748 *a64l()* function shall scan the character string from left to right with the least significant digit on
 3749 the left, decoding each character as a 6-bit radix-64 number. If the type **long** contains more than
 3750 32 bits, the resulting value is sign-extended. The behavior of *a64l()* is unspecified if *s* is a null
 3751 pointer or the string pointed to by *s* was not generated by a previous call to *l64a()*.

3752 The *l64a()* function shall take a **long** argument and return a pointer to the corresponding
 3753 radix-64 representation. The behavior of *l64a()* is unspecified if *value* is negative.

3754 The value returned by *l64a()* may be a pointer into a static buffer. Subsequent calls to *l64a()* may
 3755 overwrite the buffer.

3756 The *l64a()* function need not be thread-safe. A function that is not required to be thread-safe is
 3757 not required to be reentrant.

3758 **RETURN VALUE**

3759 Upon successful completion, *a64l()* shall return the **long** value resulting from conversion of the
 3760 input string. If a string pointed to by *s* is an empty string, *a64l()* shall return 0L.

3761 The *l64a()* function shall return a pointer to the radix-64 representation. If *value* is 0L, *l64a()* shall
 3762 return a pointer to an empty string.

3763 **ERRORS**

3764 No errors are defined.

3765 **EXAMPLES**

3766 None.

3767 **APPLICATION USAGE**3768 If the type **long** contains more than 32 bits, the result of *a64l(l64a(x))* is *x* in the low-order 32 bits.3769 **RATIONALE**3770 This is not the same encoding as used by either encoding variant of the *uuencode* utility.

3771
3772
3773
3774
3775
3776
3777
3778
3779
3780
3781
3782

FUTURE DIRECTIONS

None.

SEE ALSO

strtoul(), the Base Definitions volume of IEEE Std 1003.1-200x, `<stdlib.h>`, the Shell and Utilities volume of IEEE Std 1003.1-200x, *uuencode*

CHANGE HISTORY

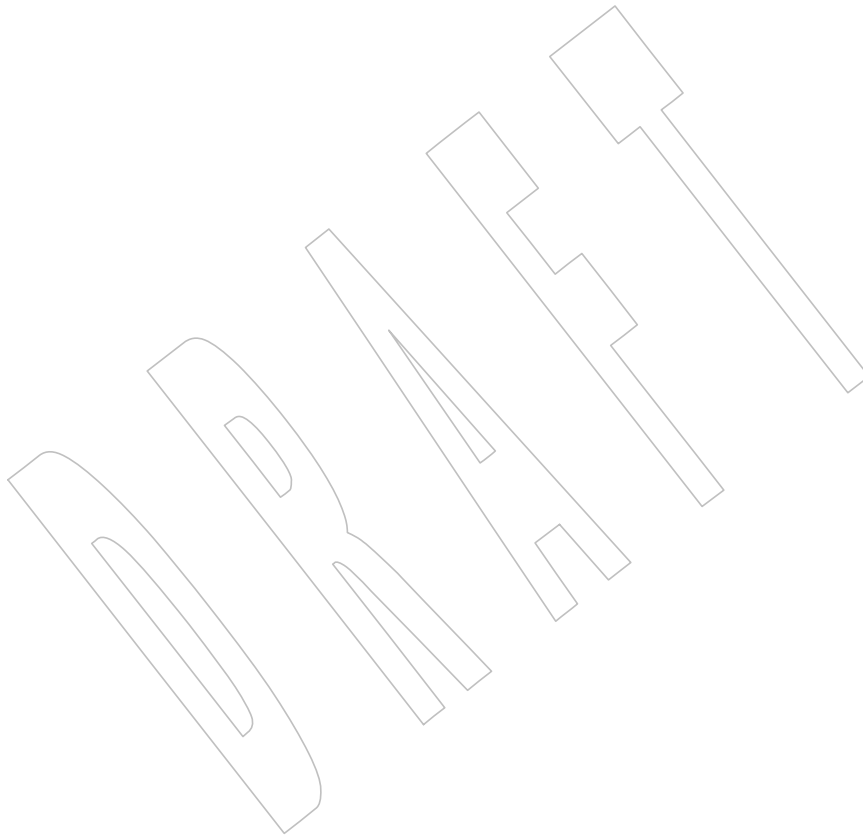
First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

A note indicating that the *l64a()* function need not be reentrant is added to the DESCRIPTION.



3783 **NAME**

3784 abort — generate an abnormal process abort

3785 **SYNOPSIS**

3786 #include <stdlib.h>

3787 void abort(void);

3788 **DESCRIPTION**

3789 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 3790 conflict between the requirements described here and the ISO C standard is unintentional. This
 3791 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

3792 The *abort()* function shall cause abnormal process termination to occur, unless the signal
 3793 SIGABRT is being caught and the signal handler does not return.

3794 CX The abnormal termination processing shall include the default actions defined for SIGABRT and
 3795 may include an attempt to effect *fclose()* on all open streams.

3796 The SIGABRT signal shall be sent to the calling process as if by means of *raise()* with the
 3797 argument SIGABRT.

3798 CX The status made available to *wait()* or *waitpid()* by *abort()* shall be that of a process terminated
 3799 by the SIGABRT signal. The *abort()* function shall override blocking or ignoring the SIGABRT
 3800 signal.

3801 **RETURN VALUE**3802 The *abort()* function shall not return.3803 **ERRORS**

3804 No errors are defined.

3805 **EXAMPLES**

3806 None.

3807 **APPLICATION USAGE**

3808 Catching the signal is intended to provide the application writer with a portable means to abort
 3809 processing, free from possible interference from any implementation-supplied functions.

3810 **RATIONALE**

3811 The ISO/IEC 9899:1999 standard requires the *abort()* function to be async-signal-safe. Since
 3812 IEEE Std 1003.1-200x defers to the ISO C standard, this required a change to the DESCRIPTION
 3813 from “shall include the effect of *fclose()*” to “may include an attempt to effect *fclose()*.”

3814 The revised wording permits some backwards-compatibility and avoids a potential deadlock
 3815 situation.

3816 The Open Group Base Resolution bwg2002-003 is applied, removing the following XSI shaded
 3817 paragraph from the DESCRIPTION:

3818 “On XSI-conformant systems, in addition the abnormal termination processing shall include the
 3819 effect of *fclose()* on message catalog descriptors.”

3820 There were several reasons to remove this paragraph:

- 3821 • No special processing of open message catalogs needs to be performed prior to abnormal
 3822 process termination.
- 3823 • The main reason to specifically mention that *abort()* includes the effect of *fclose()* on open
 3824 streams is to flush output queued on the stream. Message catalogs in this context are read-
 3825 only and, therefore, do not need to be flushed.

- The effect of *fclose()* on a message catalog descriptor is unspecified. Message catalog descriptors are allowed, but not required to be implemented using a file descriptor, but there is no mention in IEEE Std 1003.1-200x of a message catalog descriptor using a standard I/O stream FILE object as would be expected by *fclose()*.

FUTURE DIRECTIONS

None.

SEE ALSO

exit(), *kill()*, *raise()*, *signal()*, *wait()*, *waitpid()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdlib.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are marked.

Changes are made to the DESCRIPTION for alignment with the ISO/IEC 9899:1999 standard.

The Open Group Base Resolution bwg2002-003 is applied.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/10 is applied, changing the DESCRIPTION of abnormal termination processing and adding to the RATIONALE section.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/9 is applied, changing “implementation-defined functions” to “implementation-supplied functions” in the APPLICATION USAGE section.

3846 **NAME**
 3847 `abs` — return an integer absolute value

3848 **SYNOPSIS**
 3849 `#include <stdlib.h>`
 3850 `int abs(int i);`

3851 **DESCRIPTION**
 3852 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 3853 conflict between the requirements described here and the ISO C standard is unintentional. This
 3854 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

3855 The `abs()` function shall compute the absolute value of its integer operand, *i*. If the result cannot
 3856 be represented, the behavior is undefined.

3857 **RETURN VALUE**
 3858 The `abs()` function shall return the absolute value of its integer operand.

3859 **ERRORS**
 3860 No errors are defined.

3861 **EXAMPLES**
 3862 None.

3863 **APPLICATION USAGE**
 3864 In two's-complement representation, the absolute value of the negative integer with largest
 3865 magnitude {INT_MIN} might not be representable.

3866 **RATIONALE**
 3867 None.

3868 **FUTURE DIRECTIONS**
 3869 None.

3870 **SEE ALSO**
 3871 *fabs()*, *labs()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdlib.h>`

3872 **CHANGE HISTORY**
 3873 First released in Issue 1. Derived from Issue 1 of the SVID.

3874 **Issue 6**
 3875 Extensions beyond the ISO C standard are marked.

3876 **NAME**
 3877 accept — accept a new connection on a socket

3878 **SYNOPSIS**
 3879 #include <sys/socket.h>
 3880 int accept(int *socket*, struct sockaddr *restrict *address*,
 3881 socklen_t *restrict *address_len*);

3882 **DESCRIPTION**
 3883 The *accept()* function shall extract the first connection on the queue of pending connections,
 3884 create a new socket with the same socket type protocol and address family as the specified
 3885 socket, and allocate a new file descriptor for that socket.

3886 The *accept()* function takes the following arguments:

3887	<i>socket</i>	Specifies a socket that was created with <i>socket()</i> , has been bound to an address with <i>bind()</i> , and has issued a successful call to <i>listen()</i> .
3888		
3889	<i>address</i>	Either a null pointer, or a pointer to a sockaddr structure where the address of the connecting socket shall be returned.
3890		
3891	<i>address_len</i>	Points to a socklen_t structure which on input specifies the length of the supplied sockaddr structure, and on output specifies the length of the stored address.
3892		
3893		

3894 If *address* is not a null pointer, the address of the peer for the accepted connection shall be stored
 3895 in the **sockaddr** structure pointed to by *address*, and the length of this address shall be stored in
 3896 the object pointed to by *address_len*.

3897 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
 3898 the stored address shall be truncated.

3899 If the protocol permits connections by unbound clients, and the peer is not bound, then the
 3900 value stored in the object pointed to by *address* is unspecified.

3901 If the listen queue is empty of connection requests and O_NONBLOCK is not set on the file
 3902 descriptor for the socket, *accept()* shall block until a connection is present. If the *listen()* queue is
 3903 empty of connection requests and O_NONBLOCK is set on the file descriptor for the socket,
 3904 *accept()* shall fail and set *errno* to [EAGAIN] or [EWOULDBLOCK].

3905 The accepted socket cannot itself accept more connections. The original socket remains open and
 3906 can accept more connections.

3907 **RETURN VALUE**
 3908 Upon successful completion, *accept()* shall return the non-negative file descriptor of the accepted
 3909 socket. Otherwise, -1 shall be returned and *errno* set to indicate the error.

3910 **ERRORS**
 3911 The *accept()* function shall fail if:
 3912 [EAGAIN] or [EWOULDBLOCK]
 3913 O_NONBLOCK is set for the socket file descriptor and no connections are
 3914 present to be accepted.
 3915 [EBADF] The *socket* argument is not a valid file descriptor.
 3916 [ECONNABORTED]
 3917 A connection has been aborted.

accept()

3918	[EINTR]	The <i>accept()</i> function was interrupted by a signal that was caught before a valid connection arrived.
3919		
3920	[EINVAL]	The <i>socket</i> is not accepting connections.
3921	[EMFILE]	All file descriptors available to the process are currently open.
3922	[ENFILE]	The maximum number of file descriptors in the system are already open.
3923	[ENOBUFS]	No buffer space is available.
3924	[ENOMEM]	There was insufficient memory available to complete the operation.
3925	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
3926	[EOPNOTSUPP]	The socket type of the specified socket does not support accepting connections.
3927		
3928		The <i>accept()</i> function may fail if:
3929	OB XSR [EPROTO]	A protocol error has occurred; for example, the STREAMS protocol stack has not been initialized.
3930		

EXAMPLES

None.

APPLICATION USAGE

When a connection is available, *select()* indicates that the file descriptor for the socket is ready for reading.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

bind(), *connect()*, *listen()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-200x, <sys/socket.h>

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The **restrict** keyword is added to the *accept()* prototype for alignment with the ISO/IEC 9899:1999 standard.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

Austin Group Interpretation 1003.1-2001 #044 is applied, changing the “may fail” [ENOBUFS] and [ENOMEM] errors to become “shall fail” errors.

Functionality relating to XSI STREAMS is marked obsolescent.

3952 **NAME**

3953 access, faccessat — determine accessibility of a file relative to directory file descriptor

3954 **SYNOPSIS**

3955 #include <unistd.h>

3956 int access(const char *path, int amode);

3957 int faccessat(int fd, const char *path, int amode, int flag);

3958 **DESCRIPTION**3959 The *access()* function shall check the file named by the pathname pointed to by the *path*
3960 argument for accessibility according to the bit pattern contained in *amode*, using the real user ID
3961 in place of the effective user ID and the real group ID in place of the effective group ID.3962 The value of *amode* is either the bitwise-inclusive OR of the access permissions to be checked
3963 (R_OK, W_OK, X_OK) or the existence test (F_OK).3964 If any access permissions are checked, each shall be checked individually, as described in the
3965 Base Definitions volume of IEEE Std 1003.1-200x, Section 4.4, File Access Permissions, except
3966 that where that description refers to execute permission for a process with appropriate
3967 privileges, an implementation may indicate success for X_OK even if execute permission is not
3968 granted to any user.3969 The *faccessat()* function shall be equivalent to the *access()* function, except in the case where *path*
3970 specifies a relative path. In this case the file whose accessibility is to be determined shall be
3971 located relative to the directory associated with the file descriptor *fd* instead of the current
3972 working directory. It is unspecified whether directory searches are permitted based on whether
3973 the file was opened with search permission or on the current permissions of the directory
3974 underlying the file descriptor.3975 If *faccessat()* is passed the special value AT_FDCWD in the *fd* parameter, the current working
3976 directory is used and the behavior shall be identical to a call to *access()*.3977 Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined
3978 in <fcntl.h>:3979 AT_EACCESS The checks for accessibility are performed using the effective user and group
3980 IDs instead of the real user and group ID as required in a call to *access()*.3981 **RETURN VALUE**3982 Upon successful completion, these functions shall return 0. Otherwise, these functions shall
3983 return -1 and set *errno* to indicate the error.3984 **ERRORS**

3985 These functions shall fail if:

3986 [EACCES] Permission bits of the file mode do not permit the requested access, or search
3987 permission is denied on a component of the path prefix.3988 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
3989 argument.

3990 [ENAMETOOLONG]

3991 The length of the *path* argument exceeds {PATH_MAX} or a pathname
3992 component is longer than {NAME_MAX}.3993 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

- 3994 [ENOTDIR] A component of the path prefix is not a directory.
- 3995 [EROFS] Write access is requested for a file on a read-only file system.
- 3996 The *faccessat()* function shall fail if:
- 3997 [EBADF] The *path* argument does not specify an absolute path and the *fd* argument is
3998 neither AT_FDCWD nor a valid file descriptor.
- 3999 These functions may fail if:
- 4000 [EINVAL] The value of the *amode* argument is invalid.
- 4001 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
4002 resolution of the *path* argument.
- 4003 [ENAMETOOLONG]
4004 As a result of encountering a symbolic link in resolution of the *path* argument,
4005 the length of the substituted pathname string exceeded {PATH_MAX}.
- 4006 [ETXTBSY] Write access is requested for a pure procedure (shared text) file that is being
4007 executed.
- 4008 The *faccessat()* function may fail if:
- 4009 [EINVAL] The value of the *flag* argument is not valid.
- 4010 [ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a
4011 file descriptor associated with a directory.

4012 EXAMPLES

4013 Testing for the Existence of a File

4014 The following example tests whether a file named **myfile** exists in the **/tmp** directory.

```
4015 #include <unistd.h>
4016 ...
4017 int result;
4018 const char *filename = "/tmp/myfile";
4019 result = access (filename, F_OK);
```

4020 APPLICATION USAGE

4021 Additional values of *amode* other than the set defined in the description may be valid; for
4022 example, if a system has extended access controls.

4023 The use of the AT_EACCESS value for *flag* enables functionality not available in *access()*.

4024 RATIONALE

4025 In early proposals, some inadequacies in the *access()* function led to the creation of an *eaccess()*
4026 function because:

- 4027 1. Historical implementations of *access()* do not test file access correctly when the process'
4028 real user ID is superuser. In particular, they always return zero when testing execute
4029 permissions without regard to whether the file is executable.
- 4030 2. The superuser has complete access to all files on a system. As a consequence, programs
4031 started by the superuser and switched to the effective user ID with lesser privileges
4032 cannot use *access()* to test their file access permissions.

4033 However, the historical model of *eaccess()* does not resolve problem (1), so this volume of
4034 IEEE Std 1003.1-200x now allows *access()* to behave in the desired way because several
4035 implementations have corrected the problem. It was also argued that problem (2) is more easily
4036 solved by using *open()*, *chdir()*, or one of the *exec* functions as appropriate and responding to the

4037 error, rather than creating a new function that would not be as reliable. Therefore, *eaccess()* is not
4038 included in this volume of IEEE Std 1003.1-200x.

4039 The sentence concerning appropriate privileges and execute permission bits reflects the two
4040 possibilities implemented by historical implementations when checking superuser access for
4041 X_OK.

4042 New implementations are discouraged from returning X_OK unless at least one execution
4043 permission bit is set.

4044 The purpose of the *faccessat()* function is to enable the checking of the accessibility of files in
4045 directories other than the current working directory without exposure to race conditions. Any
4046 part of the path of a file could be changed in parallel to a call to *access()*, resulting in unspecified
4047 behavior. By opening a file descriptor for the target directory and using the *faccessat()* function it
4048 can be guaranteed that the file tested for accessibility is located relative to the desired directory.

4049 FUTURE DIRECTIONS

4050 None.

4051 SEE ALSO

4052 *chmod()*, *fstatat()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<fcntl.h>`, `<unistd.h>`

4053 CHANGE HISTORY

4054 First released in Issue 1. Derived from Issue 1 of the SVID.

4055 Issue 6

4056 The following new requirements on POSIX implementations derive from alignment with the
4057 Single UNIX Specification:

- 4058 • The [ELOOP] mandatory error condition is added.
- 4059 • A second [ENAMETOOLONG] is added as an optional error condition.
- 4060 • The [ETXTBSY] optional error condition is added.

4061 The following changes were made to align with the IEEE P1003.1a draft standard:

- 4062 • The [ELOOP] optional error condition is added.

4063 Issue 7

4064 Austin Group Interpretation 1003.1-2001 #046 is applied.

4065 The *faccessat()* function is added from The Open Group Technical Standard, 2006, Extended API
4066 Set Part 2.

4067 **NAME**

4068 acos, acosf, acosl — arc cosine functions

4069 **SYNOPSIS**

```
4070     #include <math.h>
4071
4071     double acos(double x);
4072     float  acosf(float x);
4073     long double acosl(long double x);
```

4074 **DESCRIPTION**

4075 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
4076 conflict between the requirements described here and the ISO C standard is unintentional. This
4077 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

4078 These functions shall compute the principal value of the arc cosine of their argument x . The
4079 value of x should be in the range $[-1,1]$.

4080 An application wishing to check for error situations should set *errno* to zero and call
4081 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
4082 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
4083 zero, an error has occurred.

4084 **RETURN VALUE**

4085 Upon successful completion, these functions shall return the arc cosine of x , in the range $[0,\pi]$
4086 radians.

4087 **MX** For finite values of x not in the range $[-1,1]$, a domain error shall occur, and either a NaN (if
4088 supported), or an implementation-defined value shall be returned.

4089 **MX** If x is NaN, a NaN shall be returned.

4090 If x is $+1$, $+0$ shall be returned.

4091 If x is $\pm\text{Inf}$, a domain error shall occur, and either a NaN (if supported), or an implementation-
4092 defined value shall be returned.

4093 **ERRORS**

4094 These functions shall fail if:

4095 **MX** Domain Error The x argument is finite and is not in the range $[-1,1]$, or is $\pm\text{Inf}$.

4096 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
4097 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
4098 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
4099 shall be raised.

4100 **EXAMPLES**

4101 None.

4102 **APPLICATION USAGE**

4103 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
4104 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

4105 **RATIONALE**

4106 None.

4107 **FUTURE DIRECTIONS**

4108 None.

4109 **SEE ALSO**4110 *cos()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-200x,
4111 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>4112 **CHANGE HISTORY**

4113 First released in Issue 1. Derived from Issue 1 of the SVID.

4114 **Issue 5**4115 The DESCRIPTION is updated to indicate how an application should check for an error. This
4116 text was previously published in the APPLICATION USAGE section.4117 **Issue 6**4118 The *acosf()* and *acosl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.4119 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
4120 revised to align with the ISO/IEC 9899:1999 standard.4121 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
4122 marked.

DRAFT

4123 **NAME**4124 `acosh`, `acoshf`, `acoshl` — inverse hyperbolic cosine functions4125 **SYNOPSIS**

```
4126 #include <math.h>
4127
4127 double acosh(double x);
4128 float acoshf(float x);
4129 long double acoshl(long double x);
```

4130 **DESCRIPTION**

4131 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 4132 conflict between the requirements described here and the ISO C standard is unintentional. This
 4133 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

4134 These functions shall compute the inverse hyperbolic cosine of their argument x .

4135 An application wishing to check for error situations should set *errno* to zero and call
 4136 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 4137 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 4138 zero, an error has occurred.

4139 **RETURN VALUE**

4140 Upon successful completion, these functions shall return the inverse hyperbolic cosine of their
 4141 argument.

4142 MX For finite values of $x < 1$, a domain error shall occur, and either a NaN (if supported), or an
 4143 implementation-defined value shall be returned.

4144 MX If x is NaN, a NaN shall be returned.

4145 If x is +1, +0 shall be returned.

4146 If x is +Inf, +Inf shall be returned.

4147 If x is -Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-
 4148 defined value shall be returned.

4149 **ERRORS**

4150 These functions shall fail if:

4151 MX Domain Error The x argument is finite and less than +1.0, or is -Inf.

4152 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 4153 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 4154 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 4155 shall be raised.

4156 **EXAMPLES**

4157 None.

4158 **APPLICATION USAGE**

4159 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
 4160 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

4161 **RATIONALE**

4162 None.

4163 **FUTURE DIRECTIONS**

4164 None.

4165 **SEE ALSO**4166 *cosh()*, *feclearexcept()*, *fetestexcept()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section
4167 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>4168 **CHANGE HISTORY**

4169 First released in Issue 4, Version 2.

4170 **Issue 5**

4171 Moved from X/OPEN UNIX extension to BASE.

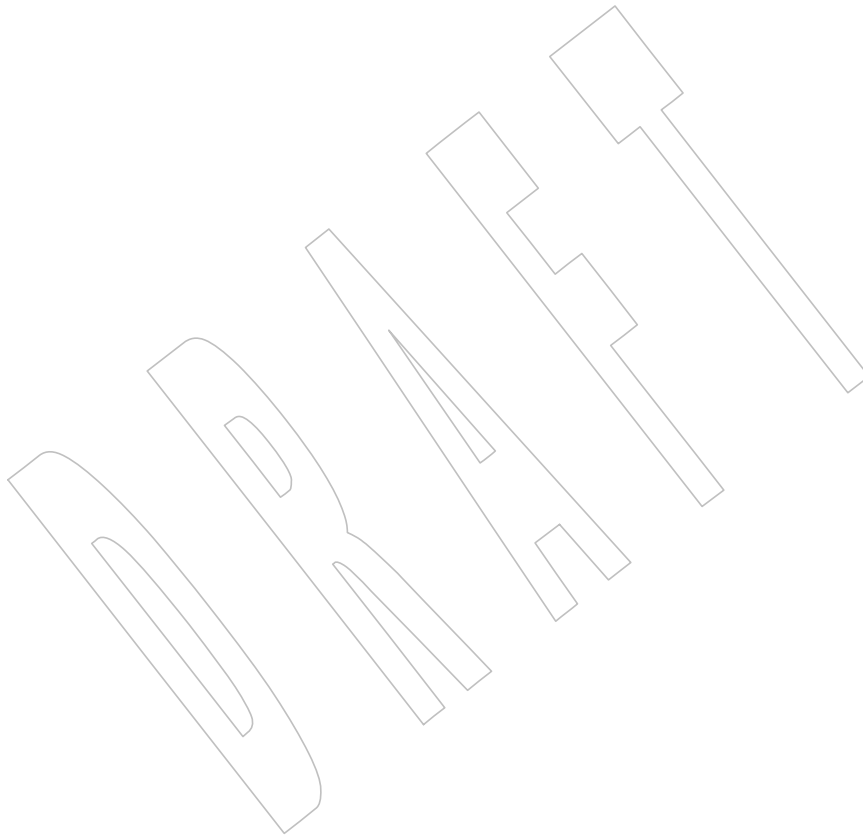
4172 **Issue 6**4173 The *acosh()* function is no longer marked as an extension.4174 The *acoshf()* and *acoshl()* functions are added for alignment with the ISO/IEC 9899:1999
4175 standard.4176 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
4177 revised to align with the ISO/IEC 9899:1999 standard.4178 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
4179 marked.

DRAFT

4180 **NAME**
4181 acosl — arc cosine functions

4182 **SYNOPSIS**
4183 #include <math.h>
4184 long double acosl(long double x);

4185 **DESCRIPTION**
4186 Refer to *acos()*.



4187 **NAME**

4188 aio_cancel — cancel an asynchronous I/O request

4189 **SYNOPSIS**

4190 #include <aio.h>

4191 int aio_cancel(int *fildev*, struct aiocb **aiocbp*);4192 **DESCRIPTION**

4193 The *aio_cancel()* function shall attempt to cancel one or more asynchronous I/O requests
 4194 currently outstanding against file descriptor *fildev*. The *aiocbp* argument points to the
 4195 asynchronous I/O control block for a particular request to be canceled. If *aiocbp* is NULL, then
 4196 all outstanding cancelable asynchronous I/O requests against *fildev* shall be canceled.

4197 Normal asynchronous notification shall occur for asynchronous I/O operations that are
 4198 successfully canceled. If there are requests that cannot be canceled, then the normal
 4199 asynchronous completion process shall take place for those requests when they are completed.

4200 For requested operations that are successfully canceled, the associated error status shall be set to
 4201 [ECANCELED] and the return status shall be -1. For requested operations that are not
 4202 successfully canceled, the *aiocbp* shall not be modified by *aio_cancel()*.

4203 If *aiocbp* is not NULL, then if *fildev* does not have the same value as the file descriptor with which
 4204 the asynchronous operation was initiated, unspecified results occur.

4205 Which operations are cancelable is implementation-defined.

4206 **RETURN VALUE**

4207 The *aio_cancel()* function shall return the value AIO_CANCELED if the requested operation(s)
 4208 were canceled. The value AIO_NOTCANCELED shall be returned if at least one of the
 4209 requested operation(s) cannot be canceled because it is in progress. In this case, the state of the
 4210 other operations, if any, referenced in the call to *aio_cancel()* is not indicated by the return value
 4211 of *aio_cancel()*. The application may determine the state of affairs for these operations by using
 4212 *aio_error()*. The value AIO_ALLDONE is returned if all of the operations have already
 4213 completed. Otherwise, the function shall return -1 and set *errno* to indicate the error.

4214 **ERRORS**

4215 The *aio_cancel()* function shall fail if:

4216 [EBADF] The *fildev* argument is not a valid file descriptor.

4217 **EXAMPLES**

4218 None.

4219 **APPLICATION USAGE**

4220 None.

4221 **RATIONALE**

4222 None.

4223 **FUTURE DIRECTIONS**

4224 None.

4225 **SEE ALSO**

4226 *aio_read()*, *aio_write()*, the Base Definitions volume of IEEE Std 1003.1-200x, <aio.h>

4227

CHANGE HISTORY

4228

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

4229

Issue 6

4230

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

4231

4232

The APPLICATION USAGE section is added.

4233

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/10 is applied, removing the words “to the calling process” in the RETURN VALUE section. The term was unnecessary and precluded threads.

4234

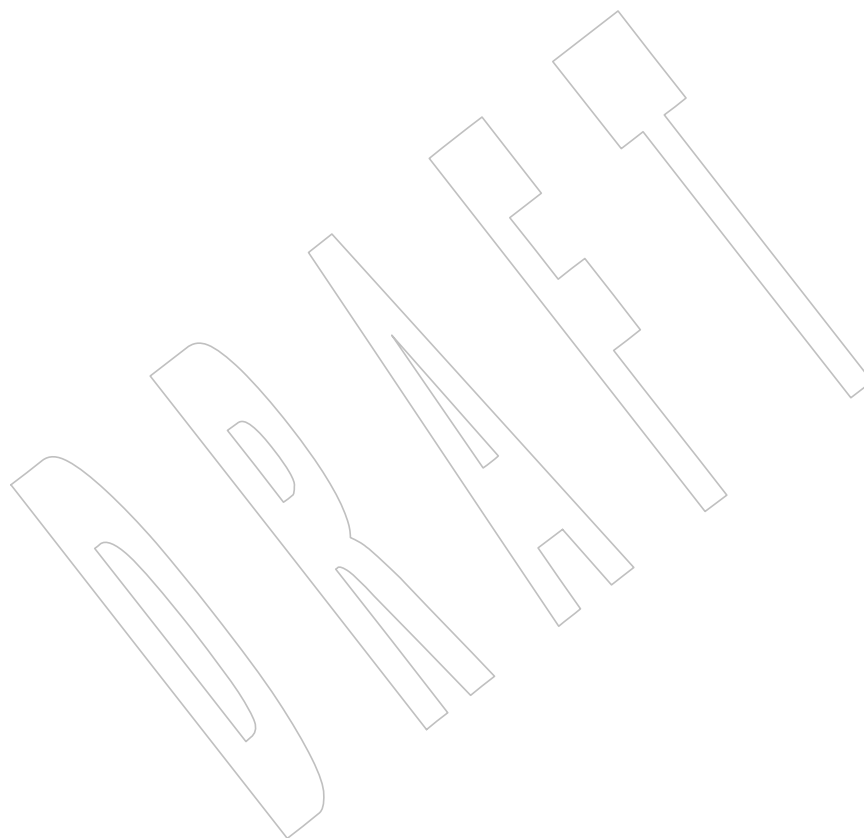
4235

4236

Issue 7

4237

The *aio_cancel()* function is moved from the Asynchronous Input and Output option to the Base.



4238 **NAME**

4239 aio_error — retrieve errors status for an asynchronous I/O operation

4240 **SYNOPSIS**

4241 #include <aio.h>

4242 int aio_error(const struct aiocb *aiocbp);

4243 **DESCRIPTION**

4244 The *aio_error()* function shall return the error status associated with the **aiocb** structure
 4245 referenced by the *aiocbp* argument. The error status for an asynchronous I/O operation is the
 4246 SIO *errno* value that would be set by the corresponding *read()*, *write()*, *fdatasync()*, or *fsync()*
 4247 operation. If the operation has not yet completed, then the error status shall be equal to
 4248 [EINPROGRESS].

4249 If the **aiocb** structure pointed to by *aiocbp* is not associated with an operation that has been
 4250 scheduled, the results are undefined.

4251 **RETURN VALUE**

4252 If the asynchronous I/O operation has completed successfully, then 0 shall be returned. If the
 4253 asynchronous operation has completed unsuccessfully, then the error status, as described for
 4254 SIO *read()*, *write()*, *fdatasync()*, and *fsync()*, shall be returned. If the asynchronous I/O operation has
 4255 not yet completed, then [EINPROGRESS] shall be returned.

4256 **ERRORS**4257 The *aio_error()* function may fail if:

4258 [EINVAL] The *aiocbp* argument does not refer to an asynchronous operation whose
 4259 return status has not yet been retrieved.

4260 **EXAMPLES**

4261 None.

4262 **APPLICATION USAGE**

4263 None.

4264 **RATIONALE**

4265 None.

4266 **FUTURE DIRECTIONS**

4267 None.

4268 **SEE ALSO**

4269 *aio_cancel()*, *aio_fsync()*, *aio_read()*, *aio_return()*, *aio_write()*, *close()*, *exec*, *exit()*, *fork()*, *lio_listio()*,
 4270 *lseek()*, *read()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**aio.h**>

4271 **CHANGE HISTORY**

4272 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

4273 **Issue 6**

4274 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 4275 implementation does not support the Asynchronous Input and Output option.

4276 The APPLICATION USAGE section is added.

4277 **Issue 7**

4278 Austin Group Interpretation 1003.1-2001 #045 is applied.

4279 The *aio_error()* function is moved from the Asynchronous Input and Output option to the Base.

4280 **NAME**

4281 aio_fsync — asynchronous file synchronization

4282 **SYNOPSIS**

4283 #include <aio.h>

4284 int aio_fsync(int op, struct aiocb *aiocbp);

4285 **DESCRIPTION**

4286 The *aio_fsync()* function shall asynchronously force all I/O operations associated with the file
 4287 indicated by the file descriptor *aio_fildes* member of the **aiocb** structure referenced by the *aiocbp*
 4288 argument and queued at the time of the call to *aio_fsync()* to the synchronized I/O completion
 4289 state. The function call shall return when the synchronization request has been initiated or
 4290 queued to the file or device (even when the data cannot be synchronized immediately).

4291 If *op* is O_DSYNC, all currently queued I/O operations shall be completed as if by a call to
 4292 *fdatasync()*; that is, as defined for synchronized I/O data integrity completion. If *op* is O_SYNC,
 4293 all currently queued I/O operations shall be completed as if by a call to *fsync()*; that is, as
 4294 defined for synchronized I/O file integrity completion. If the *aio_fsync()* function fails, or if the
 4295 operation queued by *aio_fsync()* fails, then, as for *fsync()* and *fdatasync()*, outstanding I/O
 4296 operations are not guaranteed to have been completed.

4297 If *aio_fsync()* succeeds, then it is only the I/O that was queued at the time of the call to
 4298 *aio_fsync()* that is guaranteed to be forced to the relevant completion state. The completion of
 4299 subsequent I/O on the file descriptor is not guaranteed to be completed in a synchronized
 4300 fashion.

4301 The *aiocbp* argument refers to an asynchronous I/O control block. The *aiocbp* value may be used
 4302 as an argument to *aio_error()* and *aio_return()* in order to determine the error status and return
 4303 status, respectively, of the asynchronous operation while it is proceeding. When the request is
 4304 queued, the error status for the operation is [EINPROGRESS]. When all data has been
 4305 successfully transferred, the error status shall be reset to reflect the success or failure of the
 4306 operation. If the operation does not complete successfully, the error status for the operation shall
 4307 be set to indicate the error. The *aio_sigevent* member determines the asynchronous notification to
 4308 occur as specified in Section 2.4.1 when all operations have achieved synchronized I/O
 4309 completion. All other members of the structure referenced by *aiocbp* are ignored. If the control
 4310 block referenced by *aiocbp* becomes an illegal address prior to asynchronous I/O completion,
 4311 then the behavior is undefined.

4312 If the *aio_fsync()* function fails or *aiocbp* indicates an error condition, data is not guaranteed to
 4313 have been successfully transferred.

4314 **RETURN VALUE**

4315 The *aio_fsync()* function shall return the value 0 if the I/O operation is successfully queued;
 4316 otherwise, the function shall return the value -1 and set *errno* to indicate the error.

4317 **ERRORS**

4318 The *aio_fsync()* function shall fail if:

- | | | |
|------|----------|--|
| 4319 | [EAGAIN] | The requested asynchronous operation was not queued due to temporary resource limitations. |
| 4321 | [EBADF] | The <i>aio_fildes</i> member of the aiocb structure referenced by the <i>aiocbp</i> argument is not a valid file descriptor open for writing. |
| 4323 | [EINVAL] | This implementation does not support synchronized I/O for this file. |

4324 [EINVAL] A value of *op* other than O_DSYNC or O_SYNC was specified.

4325 In the event that any of the queued I/O operations fail, *aio_fsync()* shall return the error
 4326 condition defined for *read()* and *write()*. The error is returned in the error status for the
 4327 asynchronous *fsync()* operation, which can be retrieved using *aio_error()*.

4328 EXAMPLES

4329 None.

4330 APPLICATION USAGE

4331 None.

4332 RATIONALE

4333 None.

4334 FUTURE DIRECTIONS

4335 None.

4336 SEE ALSO

4337 *fcntl()*, *fdatasync()*, *fsync()*, *open()*, *read()*, *write()*, the Base Definitions volume of
 4338 IEEE Std 1003.1-200x, <**aio.h**>

4339 CHANGE HISTORY

4340 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

4341 Issue 6

4342 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 4343 implementation does not support the Asynchronous Input and Output option.

4344 The APPLICATION USAGE section is added.

4345 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/11 is applied, removing the words “to the
 4346 calling process” in the RETURN VALUE section. The term was unnecessary and precluded
 4347 threads.

4348 Issue 7

4349 The *aio_fsync()* function is moved from the Asynchronous Input and Output option to the Base.

4350 **NAME**

4351 aio_read — asynchronous read from a file

4352 **SYNOPSIS**

4353 #include <aio.h>

4354 int aio_read(struct aiocb *aiocbp);

4355 **DESCRIPTION**

4356 The *aio_read()* function shall read *aiocbp->aio_nbytes* from the file associated with
 4357 *aiocbp->aio_fildes* into the buffer pointed to by *aiocbp->aio_buf*. The function call shall return
 4358 when the read request has been initiated or queued to the file or device (even when the data
 4359 cannot be delivered immediately).

4360 **PIO** If prioritized I/O is supported for this file, then the asynchronous operation shall be submitted
 4361 at a priority equal to a base scheduling priority minus *aiocbp->aio_reqprio*. If Thread Execution
 4362 Scheduling is not supported, then the base scheduling priority is that of the calling process;
 4363 **PIO TPS** otherwise, the base scheduling priority is that of the calling thread.

4364 The *aiocbp* value may be used as an argument to *aio_error()* and *aio_return()* in order to
 4365 determine the error status and return status, respectively, of the asynchronous operation while it
 4366 is proceeding. If an error condition is encountered during queuing, the function call shall return
 4367 without having initiated or queued the request. The requested operation takes place at the
 4368 absolute position in the file as given by *aio_offset*, as if *lseek()* were called immediately prior to
 4369 the operation with an *offset* equal to *aio_offset* and a *whence* equal to *SEEK_SET*. After a
 4370 successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file
 4371 is unspecified.

4372 The *aio_sigevent* member specifies the notification which occurs when the request is completed.

4373 The *aiocbp->aio_lio_opcode* field shall be ignored by *aio_read()*.

4374 The *aiocbp* argument points to an **aiocb** structure. If the buffer pointed to by *aiocbp->aio_buf* or
 4375 the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O
 4376 completion, then the behavior is undefined.

4377 Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

4378 **SIO** If synchronized I/O is enabled on the file associated with *aiocbp->aio_fildes*, the behavior of this
 4379 function shall be according to the definitions of synchronized I/O data integrity completion and
 4380 synchronized I/O file integrity completion.

4381 For any system action that changes the process memory space while an asynchronous I/O is
 4382 outstanding to the address range being changed, the result of that action is undefined.

4383 For regular files, no data transfer shall occur past the offset maximum established in the open
 4384 file description associated with *aiocbp->aio_fildes*.

4385 **RETURN VALUE**

4386 The *aio_read()* function shall return the value zero if the I/O operation is successfully queued;
 4387 otherwise, the function shall return the value -1 and set *errno* to indicate the error.

4388 **ERRORS**

4389 The *aio_read()* function shall fail if:

4390 [EAGAIN] The requested asynchronous I/O operation was not queued due to system
 4391 resource limitations.

4392 Each of the following conditions may be detected synchronously at the time of the call to
 4393 *aio_read()*, or asynchronously. If any of the conditions below are detected synchronously, the

4394 *aio_read()* function shall return `-1` and set *errno* to the corresponding value. If any of the
 4395 conditions below are detected asynchronously, the return status of the asynchronous operation
 4396 is set to `-1`, and the error status of the asynchronous operation is set to the corresponding value.

4397 [EBADF] The *aioctx->aio_fildes* argument is not a valid file descriptor open for reading.

4398 [EINVAL] The file offset value implied by *aioctx->aio_offset* would be invalid,
 4399 PIO *aioctx->aio_reqprio* is not a valid value, or *aioctx->aio_nbytes* is an invalid
 4400 value.

4401 In the case that the *aio_read()* successfully queues the I/O operation but the operation is
 4402 subsequently canceled or encounters an error, the return status of the asynchronous operation is
 4403 one of the values normally returned by the *read()* function call. In addition, the error status of
 4404 the asynchronous operation is set to one of the error statuses normally set by the *read()* function
 4405 call, or one of the following values:

4406 [EBADF] The *aioctx->aio_fildes* argument is not a valid file descriptor open for reading.

4407 [ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit
 4408 *aio_cancel()* request.

4409 [EINVAL] The file offset value implied by *aioctx->aio_offset* would be invalid.

4410 The following condition may be detected synchronously or asynchronously:

4411 [EOVERFLOW] The file is a regular file, *aioctx->aio_nbytes* is greater than 0, and the starting
 4412 offset in *aioctx->aio_offset* is before the end-of-file and is at or beyond the
 4413 offset maximum in the open file description associated with *aioctx->aio_fildes*.

4414 EXAMPLES

4415 None.

4416 APPLICATION USAGE

4417 None.

4418 RATIONALE

4419 None.

4420 FUTURE DIRECTIONS

4421 None.

4422 SEE ALSO

4423 *aio_cancel()*, *aio_error()*, *lio_listio()*, *aio_return()*, *aio_write()*, *close()*, *exec*, *exit()*, *fork()*, *lseek()*,
 4424 *read()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**aio.h**>

4425 CHANGE HISTORY

4426 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

4427 Large File Summit extensions are added.

4428 Issue 6

4429 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 4430 implementation does not support the Asynchronous Input and Output option.

4431 The APPLICATION USAGE section is added.

4432 The following new requirements on POSIX implementations derive from alignment with the
 4433 Single UNIX Specification:

- 4434 • In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open
 4435 file description. This change is to support large files.
- 4436 • In the ERRORS section, the [Eoverflow] condition is added. This change is to support
 4437 large files.

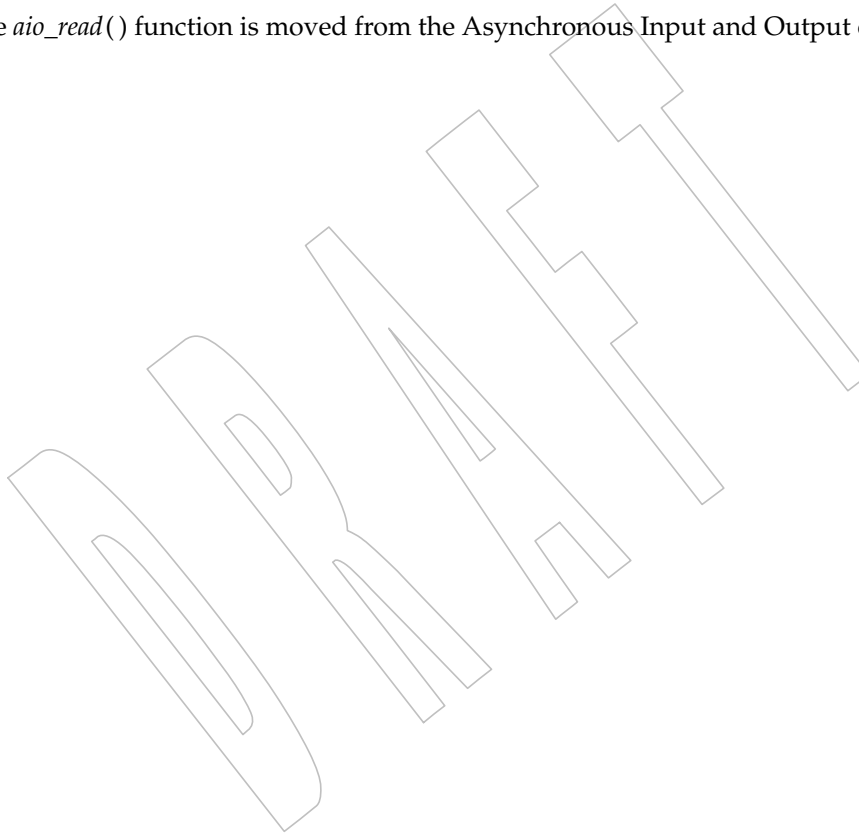
4438 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/12 is applied, rewording the
 4439 DESCRIPTION when prioritized I/O is supported to account for threads, and removing the
 4440 words “to the calling process” in the RETURN VALUE section.

4441 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/13 is applied, updating the [EINVAL]
 4442 error, so that detection of an [EINVAL] error for an invalid value of *aio_cbp*→*aio_reqprio* is only
 4443 required if the Prioritized Input and Output option is supported.

4444 **Issue 7**

4445 Austin Group Interpretation 1003.1-2001 #082 is applied.

4446 The *aio_read()* function is moved from the Asynchronous Input and Output option to the Base.



4447 **NAME**

4448 aio_return — retrieve return status of an asynchronous I/O operation

4449 **SYNOPSIS**

4450 #include <aio.h>

4451 ssize_t aio_return(struct aiocb *aiocbp);

4452 **DESCRIPTION**

4453 The *aio_return()* function shall return the return status associated with the **aiocb** structure
 4454 referenced by the *aiocbp* argument. The return status for an asynchronous I/O operation is the
 4455 value that would be returned by the corresponding *read()*, *write()*, or *fsync()* function call. If the
 4456 error status for the operation is equal to [EINPROGRESS], then the return status for the
 4457 operation is undefined. The *aio_return()* function may be called exactly once to retrieve the
 4458 return status of a given asynchronous operation; thereafter, if the same **aiocb** structure is used in
 4459 a call to *aio_return()* or *aio_error()*, an error may be returned. When the **aiocb** structure referred
 4460 to by *aiocbp* is used to submit another asynchronous operation, then *aio_return()* may be
 4461 successfully used to retrieve the return status of that operation.

4462 **RETURN VALUE**

4463 If the asynchronous I/O operation has completed, then the return status, as described for *read()*,
 4464 *write()*, and *fsync()*, shall be returned. If the asynchronous I/O operation has not yet completed,
 4465 the results of *aio_return()* are undefined.

4466 **ERRORS**4467 The *aio_return()* function may fail if:

4468 [EINVAL] The *aiocbp* argument does not refer to an asynchronous operation whose
 4469 return status has not yet been retrieved.

4470 **EXAMPLES**

4471 None.

4472 **APPLICATION USAGE**

4473 None.

4474 **RATIONALE**

4475 None.

4476 **FUTURE DIRECTIONS**

4477 None.

4478 **SEE ALSO**

4479 *aio_cancel()*, *aio_error()*, *aio_fsync()*, *aio_read()*, *aio_write()*, *close()*, *exec*, *exit()*, *fork()*, *lio_listio()*,
 4480 *lseek()*, *read()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**aio.h**>

4481 **CHANGE HISTORY**

4482 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

4483 **Issue 6**

4484 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 4485 implementation does not support the Asynchronous Input and Output option.

4486 The APPLICATION USAGE section is added.

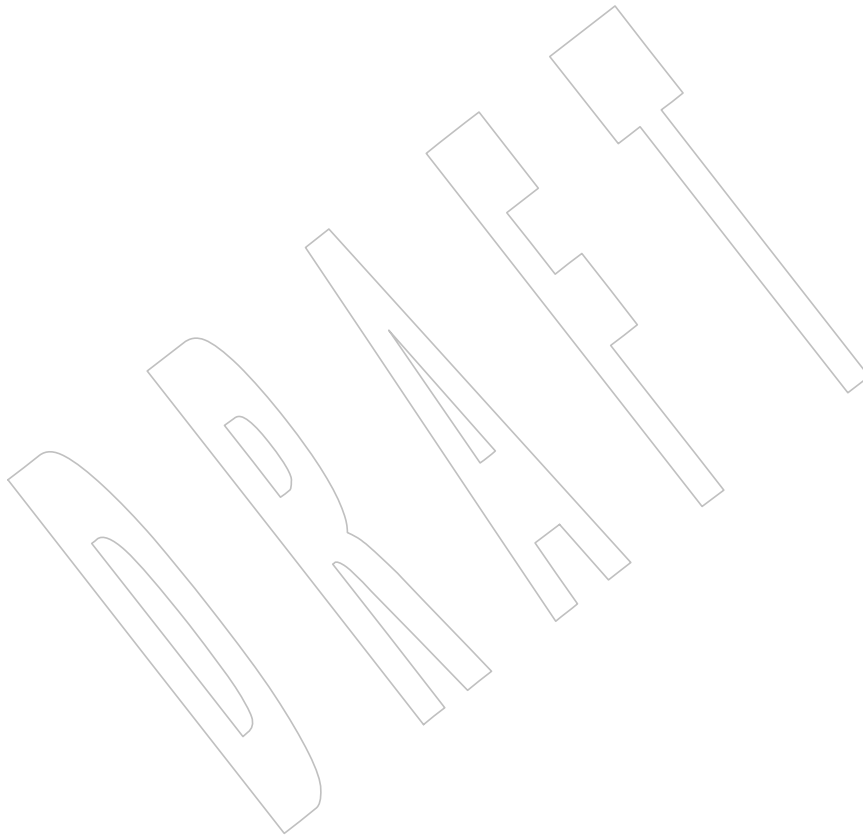
4487 The [EINVAL] error condition is made optional. This is for consistency with the DESCRIPTION.

4488

Issue 7

4489

The *aio_return()* function is moved from the Asynchronous Input and Output option to the Base.



4490 **NAME**
 4491 aio_suspend — wait for an asynchronous I/O request

4492 **SYNOPSIS**
 4493 #include <aio.h>
 4494 int aio_suspend(const struct aiocb * const list[], int nent,
 4495 const struct timespec *timeout);

4496 **DESCRIPTION**
 4497 The *aio_suspend()* function shall suspend the calling thread until at least one of the asynchronous
 4498 I/O operations referenced by the *list* argument has completed, until a signal interrupts the
 4499 function, or, if *timeout* is not NULL, until the time interval specified by *timeout* has passed. If any
 4500 of the **aiocb** structures in the list correspond to completed asynchronous I/O operations (that is,
 4501 the error status for the operation is not equal to [EINPROGRESS]) at the time of the call, the
 4502 function shall return without suspending the calling thread. The *list* argument is an array of
 4503 pointers to asynchronous I/O control blocks. The *nent* argument indicates the number of
 4504 elements in the array. Each **aiocb** structure pointed to has been used in initiating an
 4505 asynchronous I/O request via *aio_read()*, *aio_write()*, or *lio_listio()*. This array may contain
 4506 NULL pointers, which are ignored. If this array contains pointers that refer to **aiocb** structures
 4507 that have not been used in submitting asynchronous I/O, the effect is undefined.

4508 If the time interval indicated in the **timespec** structure pointed to by *timeout* passes before any of
 4509 the I/O operations referenced by *list* are completed, then *aio_suspend()* shall return with an error.

4510 MON If the Monotonic Clock option is supported, the clock that shall be used to measure this time
 4511 interval shall be the CLOCK_MONOTONIC clock.

4512 **RETURN VALUE**
 4513 If the *aio_suspend()* function returns after one or more asynchronous I/O operations have
 4514 completed, the function shall return zero. Otherwise, the function shall return a value of -1 and
 4515 set *errno* to indicate the error.

4516 The application may determine which asynchronous I/O completed by scanning the associated
 4517 error and return status using *aio_error()* and *aio_return()*, respectively.

4518 **ERRORS**
 4519 The *aio_suspend()* function shall fail if:

4520 [EAGAIN] No asynchronous I/O indicated in the list referenced by *list* completed in the
 4521 time interval indicated by *timeout*.

4522 [EINTR] A signal interrupted the *aio_suspend()* function. Note that, since each
 4523 asynchronous I/O operation may possibly provoke a signal when it
 4524 completes, this error return may be caused by the completion of one (or more)
 4525 of the very I/O operations being awaited.

4526 **EXAMPLES**
 4527 None.

4528 **APPLICATION USAGE**
 4529 None.

4530 **RATIONALE**
 4531 None.

4532
4533
4534
4535
4536
4537
4538
4539
4540
4541
4542
4543
4544
4545
4546**FUTURE DIRECTIONS**

None.

SEE ALSO

aioread(), *aiowrite()*, *lio_listio()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**aiio.h**>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

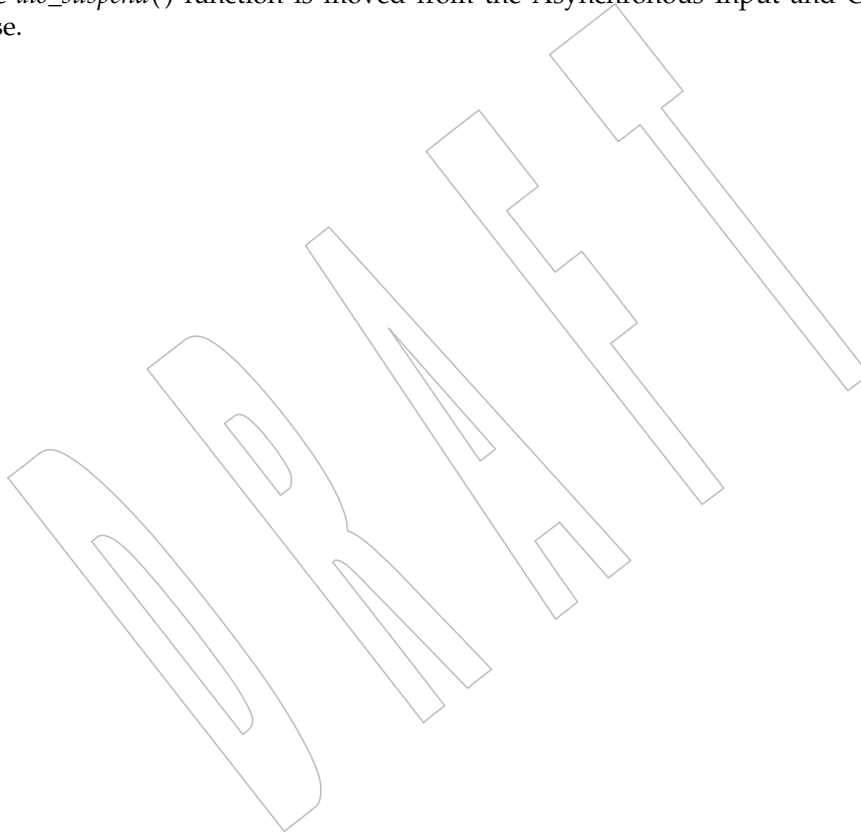
The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

The APPLICATION USAGE section is added.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that the CLOCK_MONOTONIC clock, if supported, is used.

Issue 7

The *aiosuspend()* function is moved from the Asynchronous Input and Output option to the Base.



4547 **NAME**

4548 aio_write — asynchronous write to a file

4549 **SYNOPSIS**

4550 #include <aio.h>

4551 int aio_write(struct aiocb *aiocbp);

4552 **DESCRIPTION**

4553 The *aio_write()* function shall write *aiocbp->aio_nbytes* to the file associated with
 4554 *aiocbp->aio_fildes* from the buffer pointed to by *aiocbp->aio_buf*. The function shall return when
 4555 the write request has been initiated or, at a minimum, queued to the file or device.

4556 **PIO** If prioritized I/O is supported for this file, then the asynchronous operation shall be submitted
 4557 at a priority equal to a base scheduling priority minus *aiocbp->aio_reqprio*. If Thread Execution
 4558 Scheduling is not supported, then the base scheduling priority is that of the calling process;
 4559 **PIO TPS** otherwise, the base scheduling priority is that of the calling thread.

4560 The *aiocbp* argument may be used as an argument to *aio_error()* and *aio_return()* in order to
 4561 determine the error status and return status, respectively, of the asynchronous operation while it
 4562 is proceeding.

4563 The *aiocbp* argument points to an **aiocb** structure. If the buffer pointed to by *aiocbp->aio_buf* or
 4564 the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O
 4565 completion, then the behavior is undefined.

4566 If O_APPEND is not set for the file descriptor *aio_fildes*, then the requested operation shall take
 4567 place at the absolute position in the file as given by *aio_offset*, as if *lseek()* were called
 4568 immediately prior to the operation with an *offset* equal to *aio_offset* and a *whence* equal to
 4569 SEEK_SET. If O_APPEND is set for the file descriptor, write operations append to the file in the
 4570 same order as the calls were made. After a successful call to enqueue an asynchronous I/O
 4571 operation, the value of the file offset for the file is unspecified.

4572 The *aio_sigevent* member specifies the notification which occurs when the request is completed.

4573 The *aiocbp->aio_lio_opcode* field shall be ignored by *aio_write()*.

4574 Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

4575 **SIO** If synchronized I/O is enabled on the file associated with *aiocbp->aio_fildes*, the behavior of this
 4576 function shall be according to the definitions of synchronized I/O data integrity completion, and
 4577 synchronized I/O file integrity completion.

4578 For any system action that changes the process memory space while an asynchronous I/O is
 4579 outstanding to the address range being changed, the result of that action is undefined.

4580 For regular files, no data transfer shall occur past the offset maximum established in the open
 4581 file description associated with *aiocbp->aio_fildes*.

4582 **RETURN VALUE**

4583 The *aio_write()* function shall return the value zero if the I/O operation is successfully queued;
 4584 otherwise, the function shall return the value -1 and set *errno* to indicate the error.

4585 **ERRORS**

4586 The *aio_write()* function shall fail if:

4587 [EAGAIN] The requested asynchronous I/O operation was not queued due to system
 4588 resource limitations.

4589 Each of the following conditions may be detected synchronously at the time of the call to

4590 *aiowrite()*, or asynchronously. If any of the conditions below are detected synchronously, the
 4591 *aiowrite()* function shall return `-1` and set *errno* to the corresponding value. If any of the
 4592 conditions below are detected asynchronously, the return status of the asynchronous operation
 4593 shall be set to `-1`, and the error status of the asynchronous operation is set to the corresponding
 4594 value.

4595 [EBADF] The *aiocbp->aiio_fildes* argument is not a valid file descriptor open for writing.
 4596 [EINVAL] The file offset value implied by *aiocbp->aiio_offset* would be invalid,
 4597 PIO *aiocbp->aiio_reqprio* is not a valid value, or *aiocbp->aiio_nbytes* is an invalid
 4598 value.

4599 In the case that the *aiowrite()* successfully queues the I/O operation, the return status of the
 4600 asynchronous operation shall be one of the values normally returned by the *write()* function call.
 4601 If the operation is successfully queued but is subsequently canceled or encounters an error, the
 4602 error status for the asynchronous operation contains one of the values normally set by the
 4603 *write()* function call, or one of the following:

4604 [EBADF] The *aiocbp->aiio_fildes* argument is not a valid file descriptor open for writing.
 4605 [EINVAL] The file offset value implied by *aiocbp->aiio_offset* would be invalid.
 4606 [ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit
 4607 *aiio_cancel()* request.

4608 The following condition may be detected synchronously or asynchronously:

4609 [EFBIG] The file is a regular file, *aiocbp->aiio_nbytes* is greater than 0, and the starting
 4610 offset in *aiocbp->aiio_offset* is at or beyond the offset maximum in the open file
 4611 description associated with *aiocbp->aiio_fildes*.

EXAMPLES

4612 None.

APPLICATION USAGE

4613 None.

RATIONALE

4614 None.

FUTURE DIRECTIONS

4615 None.

SEE ALSO

4616 *aiio_cancel()*, *aiio_error()*, *aiio_read()*, *aiio_return()*, *close()*, *exec*, *exit()*, *fork()*, *lio_listio()*, *lseek()*,
 4617 *write()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<aiio.h>**

CHANGE HISTORY

4618 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

4619 Large File Summit extensions are added.

Issue 6

4620 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 4621 implementation does not support the Asynchronous Input and Output option.

4622 The APPLICATION USAGE section is added.

4623 The following new requirements on POSIX implementations derive from alignment with the
 4624 Single UNIX Specification:

- 4625 • In the DESCRIPTION, text is added to indicate that for regular files no data transfer occurs
 4626 past the offset maximum established in the open file description associated with
 4627 *aiocbp->aiio_fildes*.

4635

- The [EFBIG] error is added as part of the large file support extensions.

4636

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/14 is applied, rewording the DESCRIPTION when prioritized I/O is supported to account for threads, and removing the words "to the calling process" in the RETURN VALUE section.

4637

4638

4639

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/15 is applied, updating the [EINVAL] error, so that detection of an [EINVAL] error for an invalid value of *aiocbp*→*aio_reqprio* is only required if the Prioritized Input and Output option is supported.

4640

4641

4642

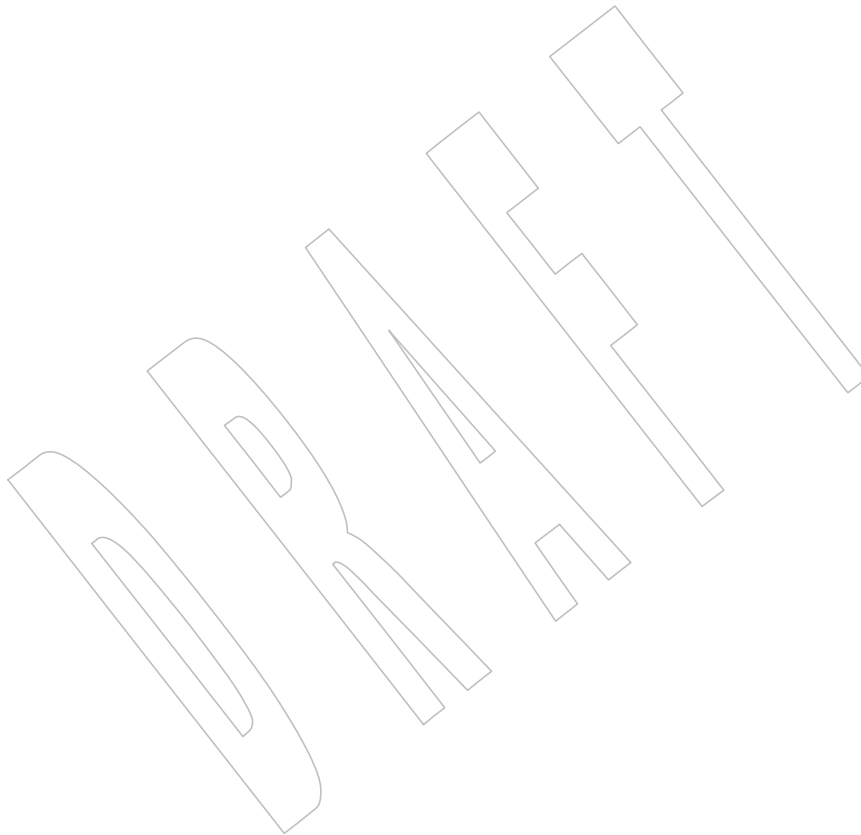
Issue 7

4643

Austin Group Interpretation 1003.1-2001 #082 is applied.

4644

The *aio_write()* function is moved from the Asynchronous Input and Output option to the Base.



4645 **NAME**

4646 alarm — schedule an alarm signal

4647 **SYNOPSIS**

4648 #include <unistd.h>

4649 unsigned alarm(unsigned *seconds*);4650 **DESCRIPTION**

4651 The *alarm()* function shall cause the system to generate a SIGALRM signal for the process after
 4652 the number of realtime seconds specified by *seconds* have elapsed. Processor scheduling delays
 4653 may prevent the process from handling the signal as soon as it is generated.

4654 If *seconds* is 0, a pending alarm request, if any, is canceled.

4655 Alarm requests are not stacked; only one SIGALRM generation can be scheduled in this manner.
 4656 If the SIGALRM signal has not yet been generated, the call shall result in rescheduling the time
 4657 at which the SIGALRM signal is generated.

4658 XSI Interactions between *alarm()* and *setitimer()* are unspecified.

4659 **RETURN VALUE**

4660 If there is a previous *alarm()* request with time remaining, *alarm()* shall return a non-zero value
 4661 that is the number of seconds until the previous request would have generated a SIGALRM
 4662 signal. Otherwise, *alarm()* shall return 0.

4663 **ERRORS**

4664 The *alarm()* function is always successful, and no return value is reserved to indicate an error.

4665 **EXAMPLES**

4666 None.

4667 **APPLICATION USAGE**

4668 The *fork()* function clears pending alarms in the child process. A new process image created by
 4669 one of the *exec* functions inherits the time left to an alarm signal in the image of the old process.

4670 Application writers should note that the type of the argument *seconds* and the return value of
 4671 *alarm()* is **unsigned**. That means that a Strictly Conforming POSIX System Interfaces
 4672 Application cannot pass a value greater than the minimum guaranteed value for {UINT_MAX},
 4673 which the ISO C standard sets as 65 535, and any application passing a larger value is restricting
 4674 its portability. A different type was considered, but historical implementations, including those
 4675 with a 16-bit **int** type, consistently use either **unsigned** or **int**.

4676 Application writers should be aware of possible interactions when the same process uses both
 4677 the *alarm()* and *sleep()* functions.

4678 **RATIONALE**

4679 Many historical implementations (including Version 7 and System V) allow an alarm to occur up
 4680 to a second early. Other implementations allow alarms up to half a second or one clock tick
 4681 early or do not allow them to occur early at all. The latter is considered most appropriate, since it
 4682 gives the most predictable behavior, especially since the signal can always be delayed for an
 4683 indefinite amount of time due to scheduling. Applications can thus choose the *seconds* argument
 4684 as the minimum amount of time they wish to have elapse before the signal.

4685 The term “realtime” here and elsewhere (*sleep()*, *times()*) is intended to mean “wall clock” time
 4686 as common English usage, and has nothing to do with “realtime operating systems”. It is in
 4687 contrast to *virtual time*, which could be misinterpreted if just *time* were used.

4688 In some implementations, including 4.3 BSD, very large values of the *seconds* argument are

4689 silently rounded down to an implementation-specific maximum value. This maximum is large
4690 enough (to the order of several months) that the effect is not noticeable.

4691 There were two possible choices for alarm generation in multi-threaded applications: generation
4692 for the calling thread or generation for the process. The first option would not have been
4693 particularly useful since the alarm state is maintained on a per-process basis and the alarm that
4694 is established by the last invocation of *alarm()* is the only one that would be active.

4695 Furthermore, allowing generation of an asynchronous signal for a thread would have
4696 introduced an exception to the overall signal model. This requires a compelling reason in order
4697 to be justified.

4698 FUTURE DIRECTIONS

4699 None.

4700 SEE ALSO

4701 *alarm()*, *exec*, *fork()*, *getitimer()*, *pause()*, *sigaction()*, *sleep()*, the Base Definitions volume of
4702 IEEE Std 1003.1-200x, **<signal.h>**, **<unistd.h>**

4703 CHANGE HISTORY

4704 First released in Issue 1. Derived from Issue 1 of the SVID.

4705 Issue 6

4706 The following new requirements on POSIX implementations derive from alignment with the
4707 Single UNIX Specification:

- 4708 • The DESCRIPTION is updated to indicate that interactions with the *setitimer()*, *ualarm()*,
4709 and *usleep()* functions are unspecified.

4710 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/16 is applied, replacing “an
4711 implementation-defined maximum value” with “an implementation-specific maximum value”
4712 in the RATIONALE.

4713 **NAME**

4714 alphasort, scandir — scan a directory

4715 **SYNOPSIS**

```
4716     #include <dirent.h>
4717
4717     int alphasort(const struct dirent **d1, const struct dirent **d2);
4718     int scandir(const char *dir, struct dirent ***namelist,
4719                int (*sel)(const struct dirent *),
4720                int (*compar)(const struct dirent **, const struct dirent **));
```

4721 **DESCRIPTION**

4722 The *alphasort()* function can be used as the comparison function for the *scandir()* function to sort
 4723 the directory entries into alphabetical order. Sorting happens as if by calling the *strcoll()*
 4724 function on the *d_name* element of the **dirent** structures passed as the two parameters. Its
 4725 parameters are the two directory entries, *d1* and *d2*, to compare.

4726 The *scandir()* function shall scan the directory *dir*, calling the function referenced by *sel* on each
 4727 directory entry. Entries for which the function referenced by *sel* returns non-zero shall be stored
 4728 in strings allocated as if by a call to *malloc()*, and sorted using *qsort()* with the comparison
 4729 function *compar*, and collected in array *namelist* which shall be allocated as if by a call to *malloc()*.
 4730 If *sel* is a null pointer, all entries shall be selected.

4731 **RETURN VALUE**

4732 Upon successful completion, the *alphasort()* function shall return an integer greater than, equal
 4733 to, or less than 0, according to whether the name of the directory entry pointed to by *d1* is
 4734 lexically greater than, equal to, or less than the directory pointed to by *d2* when both are
 4735 interpreted as appropriate to the current locale. There is no return value reserved to indicate an
 4736 error.

4737 Upon successful completion, the *scandir()* function shall return the number of entries in the
 4738 array and a pointer to the array through the parameter *namelist*. Otherwise, the *scandir()*
 4739 function shall return -1.

4740 **ERRORS**

4741 The *scandir()* function shall fail if:

4742 [EACCES] Search permission is denied for the component of the path prefix of *dir* or read
 4743 permission is denied for *dir*.

4744 [ELOOP] A loop exists in symbolic links encountered during resolution of the *dir*
 4745 argument.

4746 [ENAMETOOLONG] The length of the *dir* argument exceeds {PATH_MAX} or a pathname
 4747 component is longer than {NAME_MAX}.

4749 [ENOENT] A component of *dir* does not name an existing directory or *dir* is an empty
 4750 string.

4751 [ENOMEM] Insufficient storage space is available.

4752 [ENOTDIR] A component of *dir* is not a directory.

4753 The *scandir()* function may fail if:

4754 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 4755 resolution of the *dir* argument.

- 4756 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.
- 4757 [ENAMETOOLONG]
4758 As a result of encountering a symbolic link in resolution of the *dir* argument,
4759 the length of the substituted pathname string exceeded {PATH_MAX}.
- 4760 [ENFILE] Too many files are currently open in the system.

EXAMPLES

4761 An example to print the files in the current directory:

```
4762 #include <dirent.h>
4763 #include <stdio.h>
4764 ...
4765 struct dirent **namelist;
4766 int i,n;
4767
4768     n = scandir(".", &namelist, 0, alphasort);
4769     if (n < 0)
4770         perror("scandir");
4771     else {
4772         for (i = 0; i < n; i++) {
4773             printf("%s\n", namelist[i]->d_name);
4774             free(namelist[i]);
4775         }
4776     }
4777     free(namelist);
4778     ...
```

APPLICATION USAGE

4779 None.

RATIONALE

4781 None.

FUTURE DIRECTIONS

4783 None.

SEE ALSO

4785 *malloc()*, *qsort()*, *strcoll()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**

CHANGE HISTORY

4787 First released in Issue 7.

4789 **NAME**4790 `asctime, asctime_r` — convert date and time to a string4791 **SYNOPSIS**

```
4792 OB #include <time.h>
4793 char *asctime(const struct tm *timeptr);
4794 OB CX char *asctime_r(const struct tm *restrict tm, char *restrict buf);
```

4795 **DESCRIPTION**

4796 CX For `asctime()`: The functionality described on this reference page is aligned with the ISO C
 4797 standard. Any conflict between the requirements described here and the ISO C standard is
 4798 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

4799 The `asctime()` function shall convert the broken-down time in the structure pointed to by `timeptr`
 4800 into a string in the form:

```
4801 Sun Sep 16 01:03:52 1973\n\0
```

4802 using the equivalent of the following algorithm:

```
4803 char *asctime(const struct tm *timeptr)
4804 {
4805     static char wday_name[7][3] = {
4806         "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
4807     };
4808     static char mon_name[12][3] = {
4809         "Jan", "Feb", "Mar", "Apr", "May", "Jun",
4810         "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
4811     };
4812     static char result[26];
4813     sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
4814         wday_name[timeptr->tm_wday],
4815         mon_name[timeptr->tm_mon],
4816         timeptr->tm_mday, timeptr->tm_hour,
4817         timeptr->tm_min, timeptr->tm_sec,
4818         1900 + timeptr->tm_year);
4819     return result;
4820 }
```

4821 However, the behavior is undefined if `timeptr->tm_wday` or `timeptr->tm_mon` are not within the
 4822 normal ranges as defined in `<time.h>`, or if `timeptr->tm_year` exceeds `{INT_MAX}-1990`, or if the
 4823 above algorithm would attempt to generate more than 26 bytes of output (including the
 4824 terminating null).

4825 The `tm` structure is defined in the `<time.h>` header.

4826 CX The `asctime()`, `ctime()`, `gmtime()`, and `localtime()` functions shall return values in one of two static
 4827 objects: a broken-down time structure and an array of type `char`. Execution of any of the
 4828 functions may overwrite the information returned in either of these objects by any of the other
 4829 functions.

4830 The `asctime()` function need not be thread-safe. A function that is not required to be thread-safe
 4831 is not required to be reentrant.

4832 The `asctime_r()` function shall convert the broken-down time in the structure pointed to by `tm`

4833 into a string (of the same form as that returned by *asctime()*, and with the same undefined
 4834 behavior when input or output is out of range) that is placed in the user-supplied buffer pointed
 4835 to by *buf* (which shall contain at least 26 bytes) and then return *buf*.

RETURN VALUE

4836
 4837 CX Upon successful completion, *asctime()* shall return a pointer to the string. If the function is
 4838 unsuccessful, it shall return NULL.

4839 Upon successful completion, *asctime_r()* shall return a pointer to a character string containing
 4840 the date and time. This string is pointed to by the argument *buf*. If the function is unsuccessful,
 4841 it shall return NULL.

ERRORS

4842 No errors are defined.
 4843

EXAMPLES

4844 None.
 4845

APPLICATION USAGE

4846 These functions are included only for compatibility with older implementations. They have
 4847 undefined behavior if the resulting string would be too long, so the use of these functions
 4848 should be discouraged. On implementations that do not detect output string length overflow, it
 4849 is possible to overflow the output buffers in such a way as to cause applications to fail, or
 4850 possible system security violations. Also, these functions do not support localized date and time
 4851 formats. To avoid these problems, applications should use *strftime()* to generate strings from
 4852 broken-down times.
 4853

4854 Values for the broken-down time structure can be obtained by calling *gmtime()* or *localtime()*.

4855 The *asctime_r()* function is thread-safe and shall return values in a user-supplied buffer instead
 4856 of possibly using a static data area that may be overwritten by each call.

RATIONALE

4857 The standards developers decided to mark the *asctime()* and *asctime_r()* functions obsolescent
 4858 even though they are in the ISO C standard due to the possibility of buffer overflow. The ISO C
 4859 standard also provides the *strftime()* function which can be used to avoid these problems.
 4860

FUTURE DIRECTIONS

4861 These functions may be removed in a future version.
 4862

SEE ALSO

4863 *clock()*, *ctime()*, *difftime()*, *gmtime()*, *localtime()*, *mktime()*, *strftime()*, *strptime()*, *time()*, *utime()*,
 4864 the Base Definitions volume of IEEE Std 1003.1-200x, <**time.h**>
 4865

CHANGE HISTORY

4866 First released in Issue 1. Derived from Issue 1 of the SVID.
 4867

Issue 5

4868 Normative text previously in the APPLICATION USAGE section is moved to the
 4869 DESCRIPTION.
 4870

4871 The *asctime_r()* function is included for alignment with the POSIX Threads Extension.

4872 A note indicating that the *asctime()* function need not be reentrant is added to the
 4873 DESCRIPTION.

Issue 6

4874 The *asctime_r()* function is marked as part of the Thread-Safe Functions option.
 4875

4876 Extensions beyond the ISO C standard are marked.

4877 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
 4878 its avoidance of possibly using a static data area.

asctime()

4879

The DESCRIPTION of *asctime_r()* is updated to describe the format of the string returned.

4880

The **restrict** keyword is added to the *asctime_r()* prototype for alignment with the ISO/IEC 9899:1999 standard.

4881

4882

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/17 is applied, adding the CX extension in the RETURN VALUE section requiring that if the *asctime()* function is unsuccessful it returns NULL.

4883

4884

4885

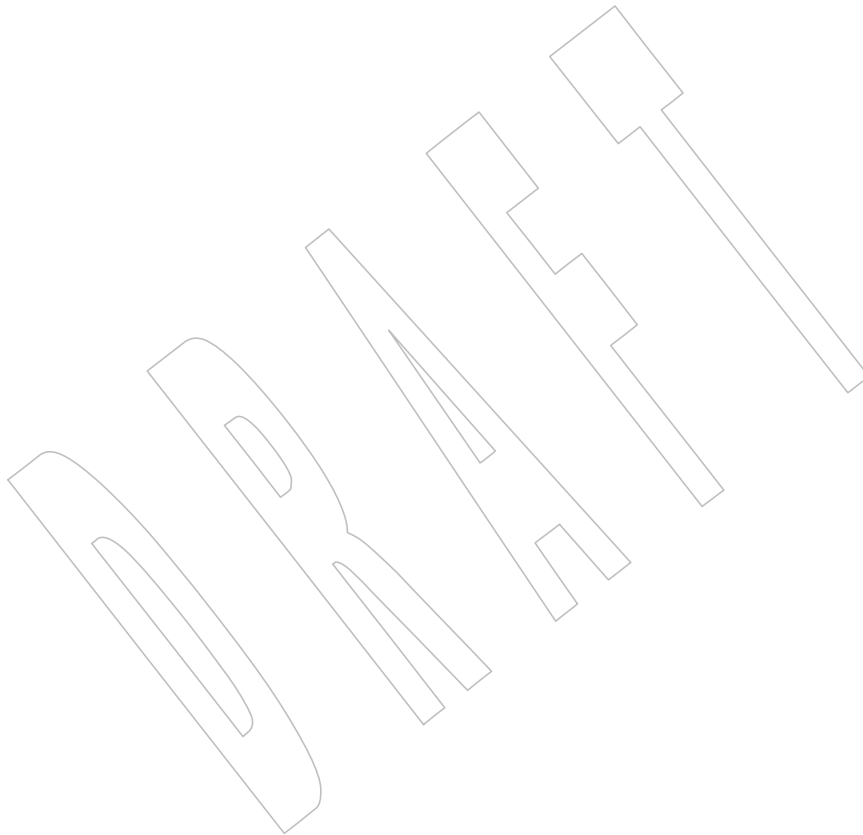
Issue 7

4886

Austin Group Interpretation 1003.1-2001 #053 is applied, marking these functions obsolescent.

4887

The *asctime_r()* function is moved from the Thread-Safe Functions option to the Base.



4888 **NAME**

4889 asin, asinf, asinl — arc sine function

4890 **SYNOPSIS**

```
4891 #include <math.h>
4892
4892 double asin(double x);
4893 float asinf(float x);
4894 long double asinl(long double x);
```

4895 **DESCRIPTION**

4896 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 4897 conflict between the requirements described here and the ISO C standard is unintentional. This
 4898 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

4899 These functions shall compute the principal value of the arc sine of their argument x . The value
 4900 of x should be in the range $[-1,1]$.

4901 An application wishing to check for error situations should set *errno* to zero and call
 4902 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 4903 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 4904 zero, an error has occurred.

4905 **RETURN VALUE**

4906 Upon successful completion, these functions shall return the arc sine of x , in the range
 4907 $[-\pi/2, \pi/2]$ radians.

4908 MX For finite values of x not in the range $[-1,1]$, a domain error shall occur, and either a NaN (if
 4909 supported), or an implementation-defined value shall be returned.

4910 MX If x is NaN, a NaN shall be returned.

4911 If x is ± 0 , x shall be returned.

4912 If x is $\pm\text{Inf}$, a domain error shall occur, and either a NaN (if supported), or an implementation-
 4913 defined value shall be returned.

4914 If x is subnormal, a range error may occur and x should be returned.

4915 **ERRORS**

4916 These functions shall fail if:

4917 MX **Domain Error** The x argument is finite and is not in the range $[-1,1]$, or is $\pm\text{Inf}$.

4918 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 4919 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 4920 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 4921 shall be raised.

4922 These functions may fail if:

4923 MX **Range Error** The value of x is subnormal.

4924 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 4925 then *errno* shall be set to [ERANGE]. If the integer expression
 4926 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 4927 floating-point exception shall be raised.

4928
4929
4930
4931
4932
4933
4934
4935
4936
4937
4938
4939
4940
4941
4942
4943
4944
4945
4946
4947
4948
4949
4950

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept(), *fetestexcept()*, *isnan()*, *sin()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *asinf()* and *asinl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

4951 **NAME**
 4952 asinh, asinhf, asinhl — inverse hyperbolic sine functions

4953 **SYNOPSIS**
 4954 #include <math.h>
 4955 double asinh(double x);
 4956 float asinhf(float x);
 4957 long double asinhl(long double x);

4958 **DESCRIPTION**
 4959 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 4960 conflict between the requirements described here and the ISO C standard is unintentional. This
 4961 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

4962 These functions shall compute the inverse hyperbolic sine of their argument x .

4963 An application wishing to check for error situations should set *errno* to zero and call
 4964 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 4965 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 4966 zero, an error has occurred.

4967 **RETURN VALUE**
 4968 Upon successful completion, these functions shall return the inverse hyperbolic sine of their
 4969 argument.

4970 MX If x is NaN, a NaN shall be returned.
 4971 If x is ± 0 , or $\pm \text{Inf}$, x shall be returned.
 4972 If x is subnormal, a range error may occur and x should be returned.

4973 **ERRORS**
 4974 These functions may fail if:
 4975 MX **Range Error** The value of x is subnormal.
 4976 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 4977 then *errno* shall be set to [ERANGE]. If the integer expression
 4978 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 4979 floating-point exception shall be raised.

4980 **EXAMPLES**
 4981 None.

4982 **APPLICATION USAGE**
 4983 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
 4984 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

4985 **RATIONALE**
 4986 None.

4987 **FUTURE DIRECTIONS**
 4988 None.

4989 **SEE ALSO**
 4990 *feclearexcept()*, *fetestexcept()*, *sinh()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section
 4991 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

4992
4993
4994
4995
4996
4997
4998
4999
5000
5001
5002
5003**CHANGE HISTORY**

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

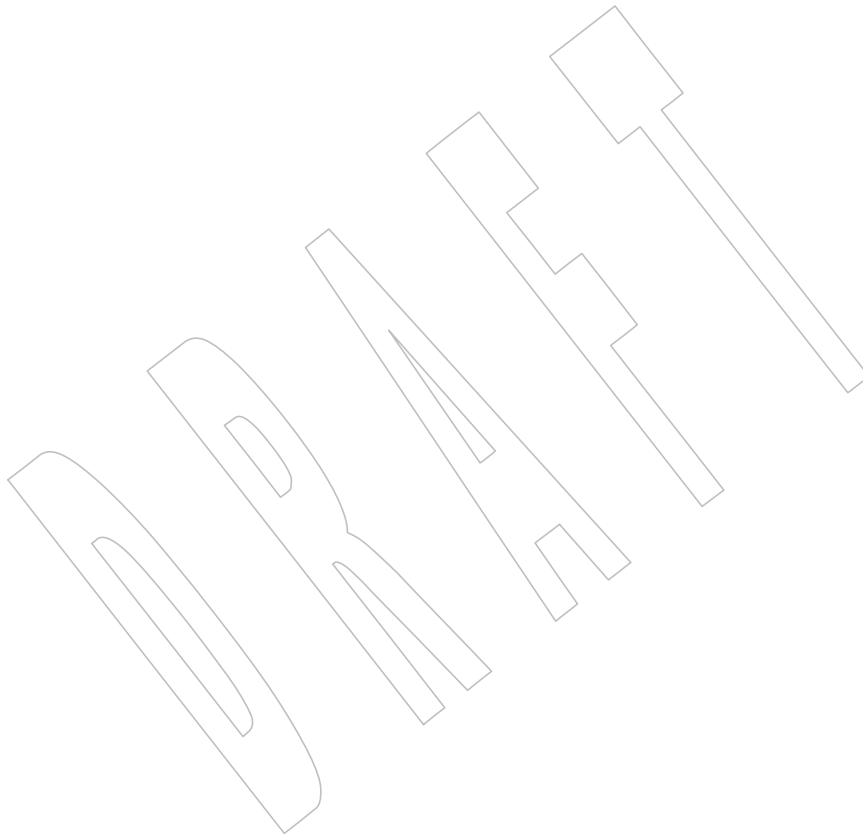
Issue 6

The *asinh()* function is no longer marked as an extension.

The *asinhf()* and *asinhll()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

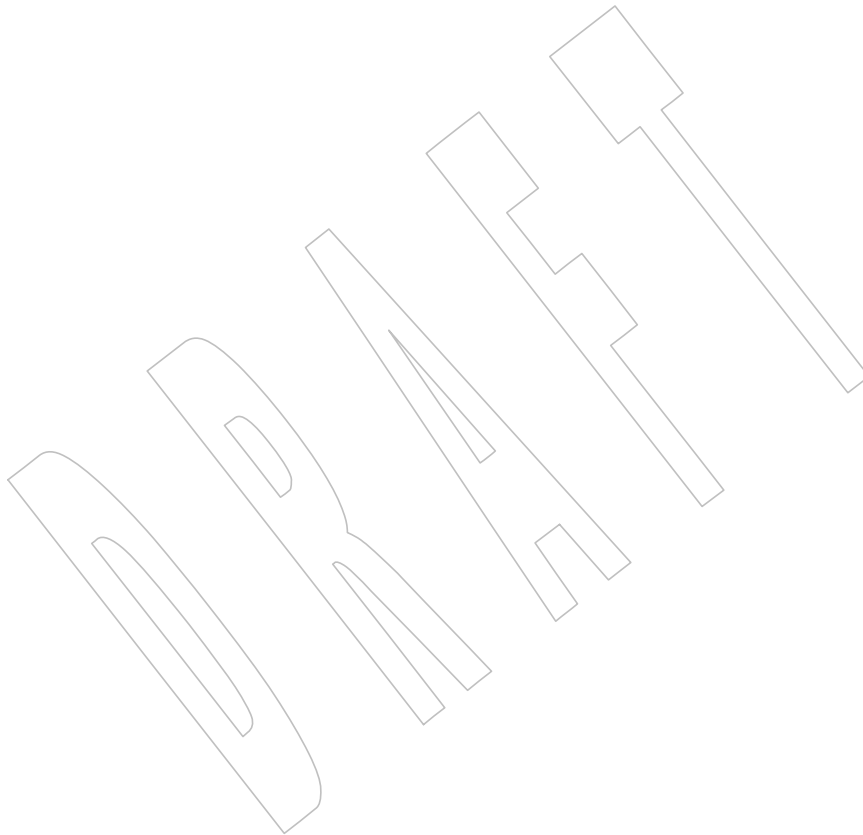
IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.



5004 **NAME**
5005 asinl — arc sine function

5006 **SYNOPSIS**
5007 #include <math.h>
5008 long double asinl(long double x);

5009 **DESCRIPTION**
5010 Refer to *asin()*.



5011 **NAME**5012 `assert` — insert program diagnostics5013 **SYNOPSIS**5014 `#include <assert.h>`5015 `void assert(scalar expression);`5016 **DESCRIPTION**

5017 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 5018 conflict between the requirements described here and the ISO C standard is unintentional. This
 5019 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5020 The `assert()` macro shall insert diagnostics into programs; it shall expand to a **void** expression.
 5021 When it is executed, if *expression* (which shall have a **scalar** type) is false (that is, compares equal
 5022 to 0), `assert()` shall write information about the particular call that failed on `stderr` and shall call
 5023 `abort()`.

5024 The information written about the call that failed shall include the text of the argument, the
 5025 name of the source file, the source file line number, and the name of the enclosing function; the
 5026 latter are, respectively, the values of the preprocessing macros `__FILE__` and `__LINE__` and of
 5027 the identifier `__func__`.

5028 Forcing a definition of the name `NDEBUG`, either from the compiler command line or with the
 5029 preprocessor control statement `#define NDEBUG` ahead of the `#include <assert.h>` statement,
 5030 shall stop assertions from being compiled into the program.

5031 **RETURN VALUE**5032 The `assert()` macro shall not return a value.5033 **ERRORS**

5034 No errors are defined.

5035 **EXAMPLES**

5036 None.

5037 **APPLICATION USAGE**

5038 None.

5039 **RATIONALE**

5040 None.

5041 **FUTURE DIRECTIONS**

5042 None.

5043 **SEE ALSO**5044 `abort()`, `stderr`, the Base Definitions volume of IEEE Std 1003.1-200x, `<assert.h>`5045 **CHANGE HISTORY**

5046 First released in Issue 1. Derived from Issue 1 of the SVID.

5047 **Issue 6**5048 The prototype for the *expression* argument to `assert()` is changed from **int** to **scalar** for alignment
 5049 with the ISO/IEC 9899:1999 standard.5050 The DESCRIPTION of `assert()` is updated for alignment with the ISO/IEC 9899:1999 standard.

5051 **NAME**

5052 atan, atanf, atanl — arc tangent function

5053 **SYNOPSIS**

```
5054 #include <math.h>
5055 double atan(double x);
5056 float atanf(float x);
5057 long double atanl(long double x);
```

5058 **DESCRIPTION**

5059 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 5060 conflict between the requirements described here and the ISO C standard is unintentional. This
 5061 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5062 These functions shall compute the principal value of the arc tangent of their argument x .

5063 An application wishing to check for error situations should set *errno* to zero and call
 5064 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 5065 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 5066 zero, an error has occurred.

5067 **RETURN VALUE**

5068 Upon successful completion, these functions shall return the arc tangent of x in the range
 5069 $[-\pi/2, \pi/2]$ radians.

5070 MX If x is NaN, a NaN shall be returned.

5071 If x is ± 0 , x shall be returned.

5072 If x is $\pm\text{Inf}$, $\pm\pi/2$ shall be returned.

5073 If x is subnormal, a range error may occur and x should be returned.

5074 **ERRORS**

5075 These functions may fail if:

5076 MX **Range Error** The value of x is subnormal.

5077 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 5078 then *errno* shall be set to [ERANGE]. If the integer expression
 5079 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 5080 floating-point exception shall be raised.

5081 **EXAMPLES**

5082 None.

5083 **APPLICATION USAGE**

5084 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
 5085 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

5086 **RATIONALE**

5087 None.

5088 **FUTURE DIRECTIONS**

5089 None.

5090
5091
5092
5093
5094
5095
5096
5097
5098
5099
5100
5101
5102
5103
5104

SEE ALSO

atan2(), *feclearexcept()*, *fetetestexcept()*, *isnan()*, *tan()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

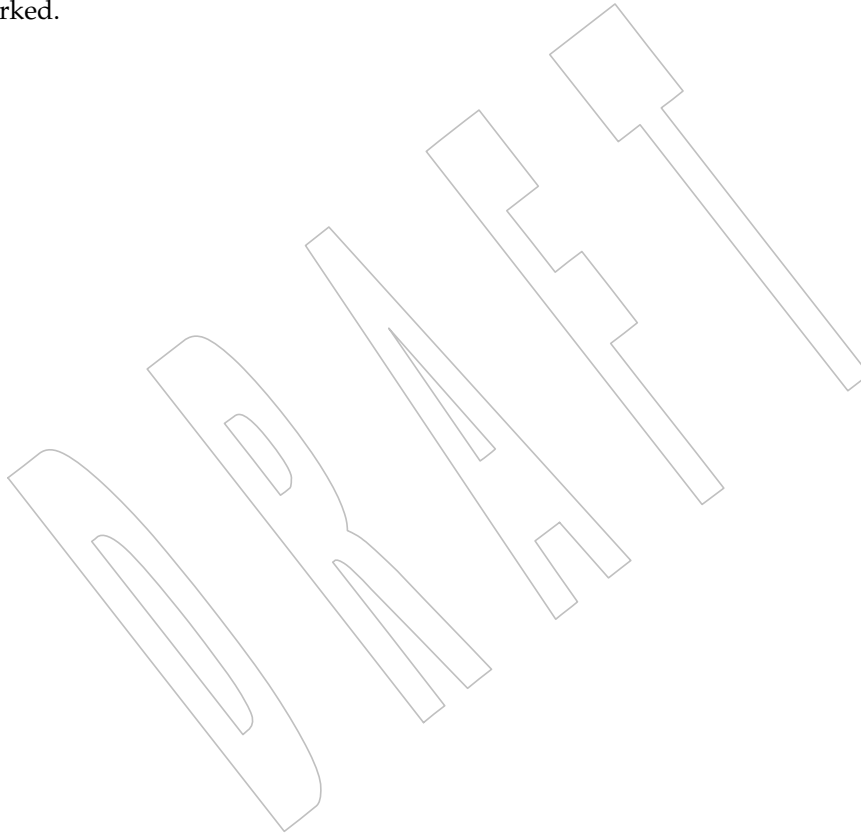
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *atanf()* and *atanl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.



5105 **NAME**

5106 atan2, atan2f, atan2l — arc tangent functions

5107 **SYNOPSIS**

5108 #include <math.h>

5109 double atan2(double y, double x);

5110 float atan2f(float y, float x);

5111 long double atan2l(long double y, long double x);

5112 **DESCRIPTION**

5113 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 5114 conflict between the requirements described here and the ISO C standard is unintentional. This
 5115 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5116 These functions shall compute the principal value of the arc tangent of y/x , using the signs of
 5117 both arguments to determine the quadrant of the return value.

5118 An application wishing to check for error situations should set *errno* to zero and call
 5119 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 5120 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 5121 zero, an error has occurred.

5122 **RETURN VALUE**

5123 Upon successful completion, these functions shall return the arc tangent of y/x in the range
 5124 $[-\pi, \pi]$ radians.

5125 If y is ± 0 and x is < 0 , $\pm\pi$ shall be returned.

5126 If y is ± 0 and x is > 0 , ± 0 shall be returned.

5127 If y is < 0 and x is ± 0 , $-\pi/2$ shall be returned.

5128 If y is > 0 and x is ± 0 , $\pi/2$ shall be returned.

5129 If x is 0, a pole error shall not occur.

5130 MX If either x or y is NaN, a NaN shall be returned.

5131 If the result underflows, a range error may occur and y/x should be returned.

5132 If y is ± 0 and x is -0 , $\pm\pi$ shall be returned.

5133 If y is ± 0 and x is $+0$, ± 0 shall be returned.

5134 For finite values of $\pm y > 0$, if x is $-\text{Inf}$, $\pm\pi$ shall be returned.

5135 For finite values of $\pm y > 0$, if x is $+\text{Inf}$, ± 0 shall be returned.

5136 For finite values of x , if y is $\pm\text{Inf}$, $\pm\pi/2$ shall be returned.

5137 If y is $\pm\text{Inf}$ and x is $-\text{Inf}$, $\pm 3\pi/4$ shall be returned.

5138 If y is $\pm\text{Inf}$ and x is $+\text{Inf}$, $\pm\pi/4$ shall be returned.

5139 If both arguments are 0, a domain error shall not occur.

5140 **ERRORS**

5141 These functions may fail if:

5142 MX **Range Error** The result underflows.

5143 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 5144 then *errno* shall be set to [ERANGE]. If the integer expression

(math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES**Converting Cartesian to Polar Coordinates System**

The function below uses *atan2()* to convert a 2d vector expressed in cartesian coordinates (x,y) to the polar coordinates $(rho,theta)$. There are other ways to compute the angle $theta$, using *asin()* or *acos()*, or *atan()*. However, *atan2()* presents here two advantages:

- The angle's quadrant is automatically determined.
- The singular cases $(0,y)$ are taken into account.

Finally, this example uses *hypot()* rather than *sqrt()* since it is better for special cases; see *hypot()* for more information.

```
#include <math.h>

void
cartesian_to_polar(const double x, const double y,
                  double *rho, double *theta
                  )
{
    *rho  = hypot (x,y); /* better than sqrt(x*x+y*y) */
    *theta = atan2 (y,x);
}
```

APPLICATION USAGE

On error, the expressions *(math_errhandling & MATH_ERRNO)* and *(math_errhandling & MATH_ERREXCEPT)* are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

acos(), *asin()*, *atan()*, *feclearexcept()*, *fetestexcept()*, *hypot()*, *isnan()*, *sqrt()*, *tan()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *atan2f()* and *atan2l()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

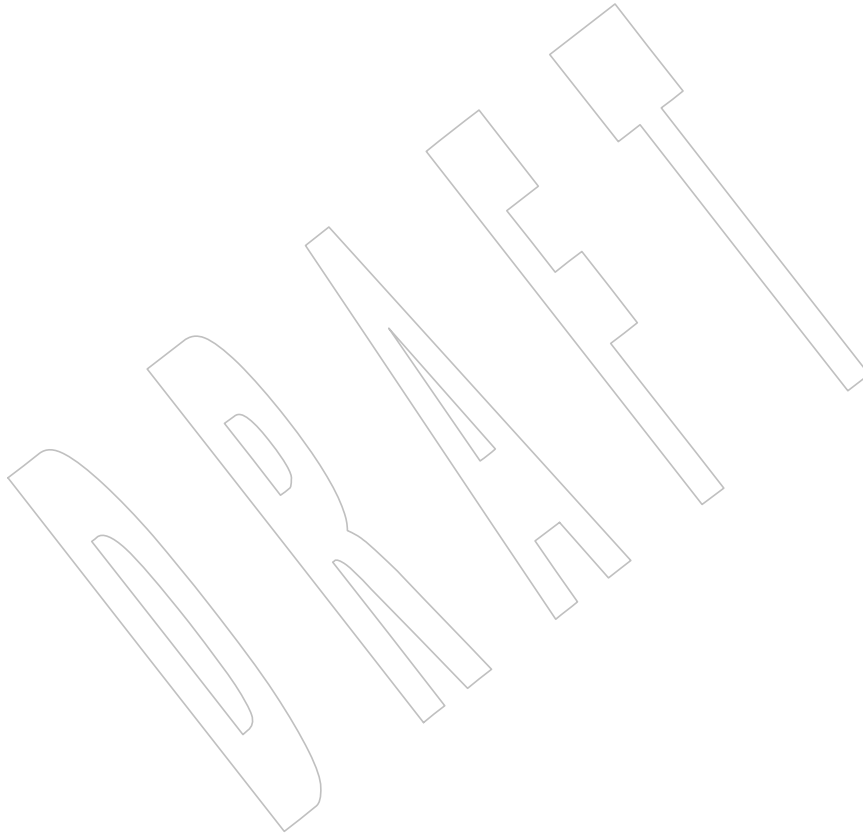
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard, and the IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/18 is applied, adding to the EXAMPLES section.

5189 **NAME**
5190 atanf — arc tangent function

5191 **SYNOPSIS**
5192 #include <math.h>
5193 float atanf(float x);

5194 **DESCRIPTION**
5195 Refer to *atan()*.



5196 **NAME**
 5197 `atanh, atanhf, atanh` — inverse hyperbolic tangent functions

5198 **SYNOPSIS**
 5199 `#include <math.h>`
 5200 `double atanh(double x);`
 5201 `float atanhf(float x);`
 5202 `long double atanh``l(long double x);`

5203 **DESCRIPTION**
 5204 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 5205 conflict between the requirements described here and the ISO C standard is unintentional. This
 5206 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5207 These functions shall compute the inverse hyperbolic tangent of their argument x .

5208 An application wishing to check for error situations should set *errno* to zero and call
 5209 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 5210 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 5211 zero, an error has occurred.

5212 **RETURN VALUE**
 5213 Upon successful completion, these functions shall return the inverse hyperbolic tangent of their
 5214 argument.

5215 If x is ± 1 , a pole error shall occur, and *atanh()*, *atanhf()*, and *atanhl()* shall return the value of the
 5216 macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively, with the same sign as the
 5217 correct value of the function.

5218 MX For finite $|x| > 1$, a domain error shall occur, and either a NaN (if supported), or an
 5219 implementation-defined value shall be returned.

5220 MX If x is NaN, a NaN shall be returned.

5221 If x is ± 0 , x shall be returned.

5222 If x is $\pm \text{Inf}$, a domain error shall occur, and either a NaN (if supported), or an implementation-
 5223 defined value shall be returned.

5224 If x is subnormal, a range error may occur and x should be returned.

5225 ERRORS

5226 These functions shall fail if:

5227 MX Domain Error The x argument is finite and not in the range $[-1, 1]$, or is $\pm \text{Inf}$.
 5228 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 5229 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 5230 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 5231 shall be raised.

5232 Pole Error The x argument is ± 1 .

5233 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 5234 then *errno* shall be set to [ERANGE]. If the integer expression
 5235 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by-zero
 5236 floating-point exception shall be raised.

5237

These functions may fail if:

5238

MX

Range Error The value of x is subnormal.

5239

5240

5241

5242

If the integer expression $(math_errhandling \ \& \ MATH_ERRNO)$ is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression $(math_errhandling \ \& \ MATH_ERREXCEPT)$ is non-zero, then the underflow floating-point exception shall be raised.

5243

EXAMPLES

5244

None.

5245

APPLICATION USAGE

5246

5247

On error, the expressions $(math_errhandling \ \& \ MATH_ERRNO)$ and $(math_errhandling \ \& \ MATH_ERREXCEPT)$ are independent of each other, but at least one of them must be non-zero.

5248

RATIONALE

5249

None.

5250

FUTURE DIRECTIONS

5251

None.

5252

SEE ALSO

5253

5254

feclearexcept(), *fetestexcept()*, *tanh()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

5255

CHANGE HISTORY

5256

First released in Issue 4, Version 2.

5257

Issue 5

5258

Moved from X/OPEN UNIX extension to BASE.

5259

Issue 6

5260

The *atanh()* function is no longer marked as an extension.

5261

5262

The *atanhf()* and *atanhl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

5263

5264

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

5265

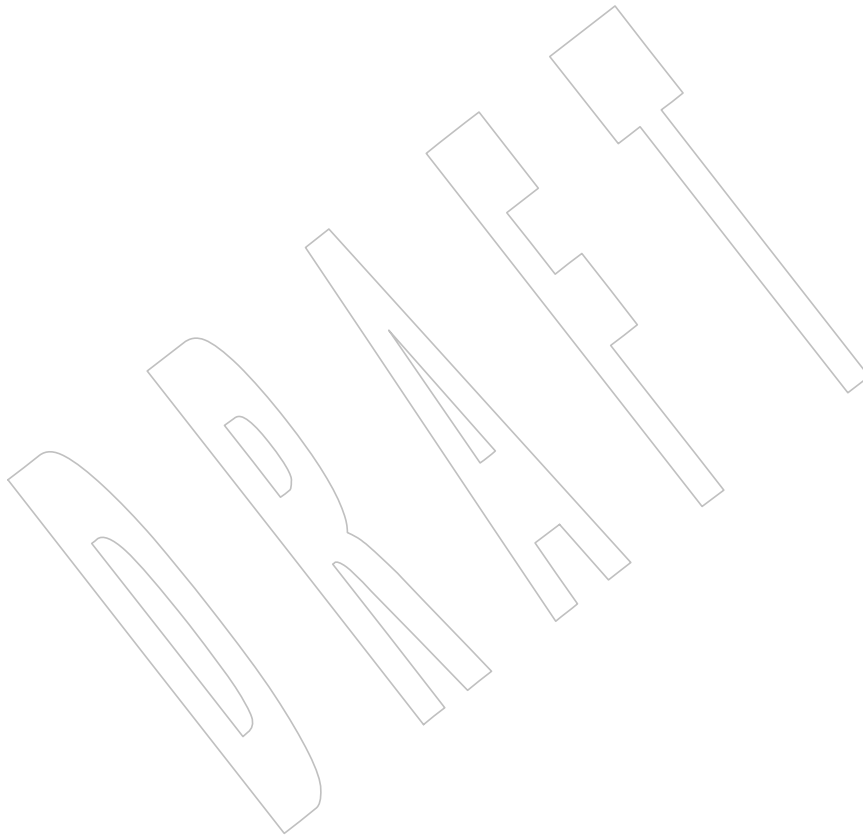
5266

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

5267 **NAME**
5268 atanl — arc tangent function

5269 **SYNOPSIS**
5270 #include <math.h>
5271 long double atanl(long double x);

5272 **DESCRIPTION**
5273 Refer to *atan()*.



5274 **NAME**
 5275 atexit — register a function to run at process termination

5276 **SYNOPSIS**
 5277 #include <stdlib.h>
 5278 int atexit(void (*func)(void));

5279 **DESCRIPTION**
 5280 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 5281 conflict between the requirements described here and the ISO C standard is unintentional. This
 5282 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5283 The *atexit()* function shall register the function pointed to by *func*, to be called without
 5284 arguments at normal program termination. At normal program termination, all functions
 5285 registered by the *atexit()* function shall be called, in the reverse order of their registration, except
 5286 that a function is called after any previously registered functions that had already been called at
 5287 the time it was registered. Normal termination occurs either by a call to *exit()* or a return from
 5288 *main()*.

5289 At least 32 functions can be registered with *atexit()*.

5290 CX After a successful call to any of the *exec* functions, any functions previously registered by *atexit()*
 5291 shall no longer be registered.

5292 **RETURN VALUE**
 5293 Upon successful completion, *atexit()* shall return 0; otherwise, it shall return a non-zero value.

5294 **ERRORS**
 5295 No errors are defined.

5296 **EXAMPLES**
 5297 None.

5298 **APPLICATION USAGE**
 5299 The functions registered by a call to *atexit()* must return to ensure that all registered functions
 5300 are called.

5301 The application should call *sysconf()* to obtain the value of {ATEXIT_MAX}, the number of
 5302 functions that can be registered. There is no way for an application to tell how many functions
 5303 have already been registered with *atexit()*.

5304 Since the behavior is undefined if the *exit()* function is called more than once, portable
 5305 applications calling *atexit()* must ensure that the *exit()* function is not called at normal process
 5306 termination when all functions registered by the *atexit()* function are called.

5307 All functions registered by the *atexit()* function are called at normal process termination, which
 5308 occurs by a call to the *exit()* function or a return from *main()* or on the last thread termination,
 5309 when the behavior is as if the implementation called *exit()* with a zero argument at thread
 5310 termination time.

5311 If, at normal process termination, a function registered by the *atexit()* function is called and a
 5312 portable application needs to stop further *exit()* processing, it must call the *_exit()* function or
 5313 the *_Exit()* function or one of the functions which cause abnormal process termination.

5314

RATIONALE

5315

None.

5316

FUTURE DIRECTIONS

5317

None.

5318

SEE ALSO

5319

exec, *exit()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

5320

CHANGE HISTORY

5321

First released in Issue 4. Derived from the ANSI C standard.

5322

Issue 6

5323

Extensions beyond the ISO C standard are marked.

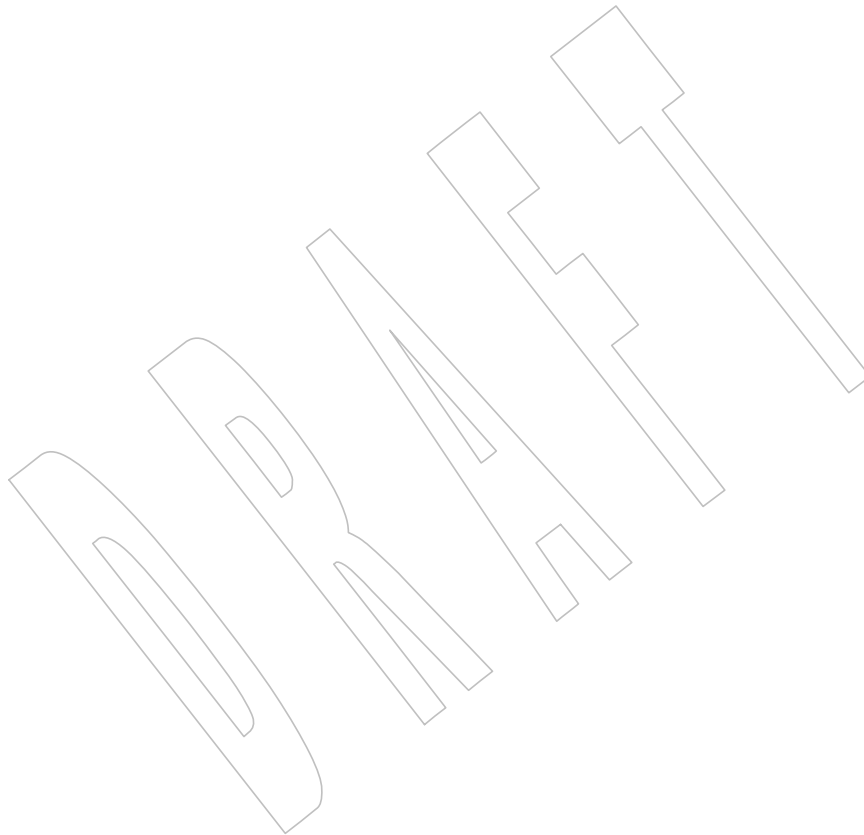
5324

The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.

5325

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/19 is applied, adding further clarification to the APPLICATION USAGE section.

5326



5327 **NAME**
 5328 atof — convert a string to a double-precision number

5329 **SYNOPSIS**
 5330 #include <stdlib.h>
 5331 double atof(const char *str);

5332 **DESCRIPTION**
 5333 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 5334 conflict between the requirements described here and the ISO C standard is unintentional. This
 5335 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5336 The call *atof(str)* shall be equivalent to:
 5337 *strtod(str, (char **)NULL)*,
 5338 except that the handling of errors may differ. If the value cannot be represented, the behavior is
 5339 undefined.

5340 **RETURN VALUE**
 5341 The *atof()* function shall return the converted value if the value can be represented.

5342 **ERRORS**
 5343 No errors are defined.

5344 **EXAMPLES**
 5345 None.

5346 **APPLICATION USAGE**
 5347 The *atof()* function is subsumed by *strtod()* but is retained because it is used extensively in
 5348 existing code. If the number is not known to be in range, *strtod()* should be used because *atof()*
 5349 is not required to perform any error checking.

5350 **RATIONALE**
 5351 None.

5352 **FUTURE DIRECTIONS**
 5353 None.

5354 **SEE ALSO**
 5355 *strtod()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdlib.h>

5356 **CHANGE HISTORY**
 5357 First released in Issue 1. Derived from Issue 1 of the SVID.

5358 **NAME**5359 `atoi` — convert a string to an integer5360 **SYNOPSIS**5361 `#include <stdlib.h>`5362 `int atoi(const char *str);`5363 **DESCRIPTION**5364 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5365 conflict between the requirements described here and the ISO C standard is unintentional. This
5366 volume of IEEE Std 1003.1-200x defers to the ISO C standard.5367 The call `atoi(str)` shall be equivalent to:5368 `(int) strtol(str, (char **)NULL, 10)`5369 except that the handling of errors may differ. If the value cannot be represented, the behavior is
5370 undefined.5371 **RETURN VALUE**5372 The `atoi()` function shall return the converted value if the value can be represented.5373 **ERRORS**

5374 No errors are defined.

5375 **EXAMPLES**5376 **Converting an Argument**5377 The following example checks for proper usage of the program. If there is an argument and the
5378 decimal conversion of this argument (obtained using `atoi()`) is greater than 0, then the program
5379 has a valid number of minutes to wait for an event.5380 `#include <stdlib.h>`
5381 `#include <stdio.h>`
5382 `...`
5383 `int minutes_to_event;`
5384 `...`
5385 `if (argc < 2 || ((minutes_to_event = atoi (argv[1]))) <= 0) {`
5386 `fprintf(stderr, "Usage: %s minutes\n", argv[0]); exit(1);`
5387 `}`
5388 `...`5389 **APPLICATION USAGE**5390 The `atoi()` function is subsumed by `strtol()` but is retained because it is used extensively in
5391 existing code. If the number is not known to be in range, `strtol()` should be used because `atoi()` is
5392 not required to perform any error checking.5393 **RATIONALE**

5394 None.

5395 **FUTURE DIRECTIONS**

5396 None.

5397

SEE ALSO

5398

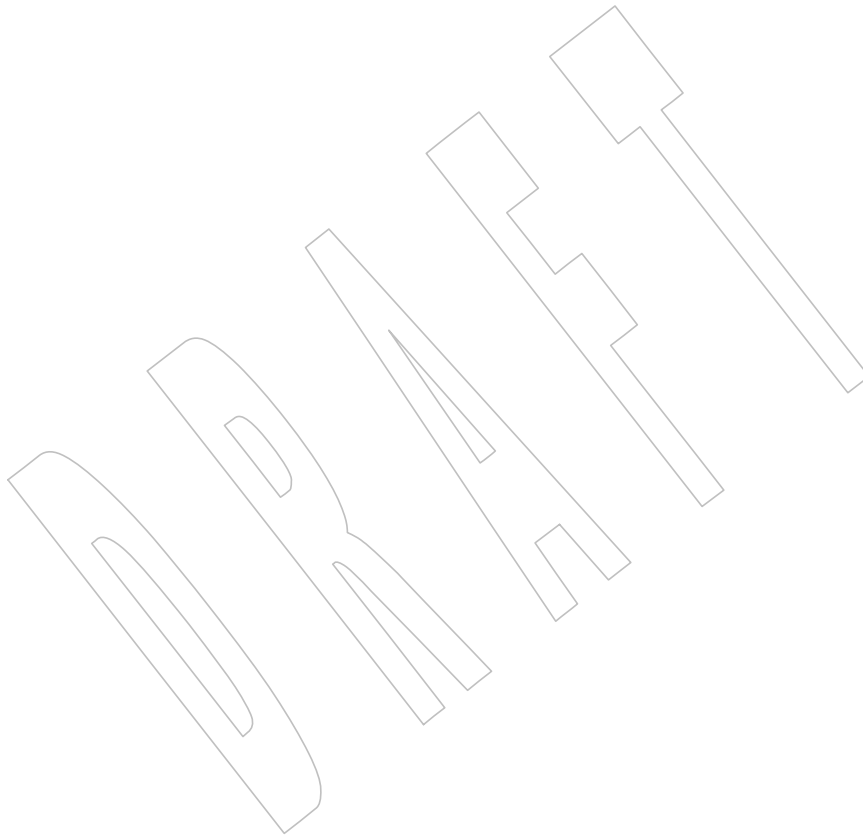
strtol(), the Base Definitions volume of IEEE Std 1003.1-200x, `<stdlib.h>`

5399

CHANGE HISTORY

5400

First released in Issue 1. Derived from Issue 1 of the SVID.



5401 **NAME**5402 `atol, atoll` — convert a string to a long integer5403 **SYNOPSIS**5404 `#include <stdlib.h>`5405 `long atol(const char *str);`5406 `long long atoll(const char *nptr);`5407 **DESCRIPTION**5408 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
5409 conflict between the requirements described here and the ISO C standard is unintentional. This
5410 volume of IEEE Std 1003.1-200x defers to the ISO C standard.5411 The call `atol(str)` shall be equivalent to:5412 `strtol(str, (char **)NULL, 10)`5413 The call `atoll(nptr)` shall be equivalent to:5414 `strtoll(nptr, (char **)NULL, 10)`5415 except that the handling of errors may differ. If the value cannot be represented, the behavior is
5416 undefined.5417 **RETURN VALUE**

5418 These functions shall return the converted value if the value can be represented.

5419 **ERRORS**

5420 No errors are defined.

5421 **EXAMPLES**

5422 None.

5423 **APPLICATION USAGE**5424 The `atol()` function is subsumed by `strtol()` but is retained because it is used extensively in
5425 existing code. If the number is not known to be in range, `strtol()` should be used because `atol()` is
5426 not required to perform any error checking.5427 **RATIONALE**

5428 None.

5429 **FUTURE DIRECTIONS**

5430 None.

5431 **SEE ALSO**5432 `strtol()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdlib.h>`5433 **CHANGE HISTORY**

5434 First released in Issue 1. Derived from Issue 1 of the SVID.

5435 **Issue 6**5436 The `atoll()` function is added for alignment with the ISO/IEC 9899:1999 standard.5437 **Issue 7**5438 SD5-XSH-ERN-61 is applied, correcting the DESCRIPTION of `atoll()`.

basename()5479
5480
5481
5482
5483
5484
5485
5486
5487
5488
5489
5490
5491
5492
5493
5494
5495
5496
5497
5498**APPLICATION USAGE**

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

dirname(), the Base Definitions volume of IEEE Std 1003.1-200x, **<libgen.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *basename*

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/20 is applied, changing the DESCRIPTION to make it clear that the string referenced is the string pointed to by *path*.

5499 **NAME**5500 `bind` — bind a name to a socket5501 **SYNOPSIS**

```
5502 #include <sys/socket.h>
5503
5504 int bind(int socket, const struct sockaddr *address,
5505         socklen_t address_len);
```

5505 **DESCRIPTION**

5506 The `bind()` function shall assign a local socket address *address* to a socket identified by descriptor
 5507 *socket* that has no local socket address assigned. Sockets created with the `socket()` function are
 5508 initially unnamed; they are identified only by their address family.

5509 The `bind()` function takes the following arguments:

5510	<i>socket</i>	Specifies the file descriptor of the socket to be bound.
5511	<i>address</i>	Points to a sockaddr structure containing the address to be bound to the socket. The length and format of the address depend on the address family of the socket.
5512		
5513		
5514	<i>address_len</i>	Specifies the length of the sockaddr structure pointed to by the <i>address</i> argument.
5515		

5516 The socket specified by *socket* may require the process to have appropriate privileges to use the
 5517 `bind()` function.

5518 **RETURN VALUE**

5519 Upon successful completion, `bind()` shall return 0; otherwise, `-1` shall be returned and *errno* set
 5520 to indicate the error.

5521 **ERRORS**

5522 The `bind()` function shall fail if:

5523	[EADDRINUSE]	The specified address is already in use.
5524	[EADDRNOTAVAIL]	The specified address is not available from the local machine.
5525		
5526	[EAFNOSUPPORT]	The specified address is not a valid address for the address family of the specified socket.
5527		
5528		
5529	[EBADF]	The <i>socket</i> argument is not a valid file descriptor.
5530	[EINVAL]	The socket is already bound to an address, and the protocol does not support binding to a new address; or the socket has been shut down.
5531		
5532	[ENOBUFS]	Insufficient resources were available to complete the call.
5533	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
5534	[EOPNOTSUPP]	The socket type of the specified socket does not support binding to an address.
5535		If the address family of the socket is <code>AF_UNIX</code> , then <code>bind()</code> shall fail if:
5536	[EACCES]	A component of the path prefix denies search permission, or the requested name requires writing in a directory with a mode that denies write permission.
5537		
5538		

5539	[EDESTADDRREQ] or [EISDIR]	
5540		The <i>address</i> argument is a null pointer.
5541	[EIO]	An I/O error occurred.
5542	[ELOOP]	A loop exists in symbolic links encountered during resolution of the pathname in <i>address</i> .
5543		
5544	[ENAMETOOLONG]	
5545		A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
5546		
5547	[ENOENT]	A component of the pathname does not name an existing file or the pathname is an empty string.
5548		
5549	[ENOTDIR]	A component of the path prefix of the pathname in <i>address</i> is not a directory.
5550	[EROFS]	The name would reside on a read-only file system.
5551		The <i>bind()</i> function may fail if:
5552	[EACCES]	The specified address is protected and the current user does not have permission to bind to it.
5553		
5554	[EINVAL]	The <i>address_len</i> argument is not a valid length for the address family.
5555	[EISCONN]	The socket is already connected.
5556	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the pathname in <i>address</i> .
5557		
5558	[ENAMETOOLONG]	
5559		Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.
5560		

EXAMPLES

None.

APPLICATION USAGEAn application program can retrieve the assigned socket name with the *getsockname()* function.**RATIONALE**

None.

FUTURE DIRECTIONS

None.

SEE ALSO*connect()*, *getsockname()*, *listen()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-200x, <sys/socket.h>**CHANGE HISTORY**

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

Issue 7

Austin Group Interpretation 1003.1-2001 #044 is applied, changing the “may fail” [ENOBUFS] error to become a “shall fail” error.


```

5621     char *string;
5622     int length;
5623 };
5624 struct node table[TABSIZE];    /* Table to be searched. */
5625     .
5626     .
5627     .
5628 {
5629     struct node *node_ptr, node;
5630     /* Routine to compare 2 nodes. */
5631     int node_compare(const void *, const void *);
5632     char str_space[20];    /* Space to read string into. */
5633     .
5634     .
5635     .
5636     node.string = str_space;
5637     while (scanf("%s", node.string) != EOF) {
5638         node_ptr = (struct node *)bsearch((void *)&node,
5639             (void *)table, TABSIZE,
5640             sizeof(struct node), node_compare);
5641         if (node_ptr != NULL) {
5642             (void)printf("string = %20s, length = %d\n",
5643                 node_ptr->string, node_ptr->length);
5644         } else {
5645             (void)printf("not found: %s\n", node.string);
5646         }
5647     }
5648 }
5649 /*
5650     This routine compares two nodes based on an
5651     alphabetical ordering of the string field.
5652 */
5653 int
5654 node_compare(const void *node1, const void *node2)
5655 {
5656     return strcoll(((const struct node *)node1)->string,
5657         ((const struct node *)node2)->string);
5658 }

```

APPLICATION USAGE

The pointers to the key and the element at the base of the table should be of type pointer-to-element.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

In practice, the array is usually sorted according to the comparison function.

RATIONALE

The requirement that the second argument (hereafter referred to as *p*) to the comparison function is a pointer to an element of the array implies that for every call all of the following expressions are non-zero:

```

5669 ((char *)p - (char *(base) % width == 0
5670 (char *)p >= (char *)base
5671 (char *)p < (char *)base + nel * width

```

5672 **FUTURE DIRECTIONS**

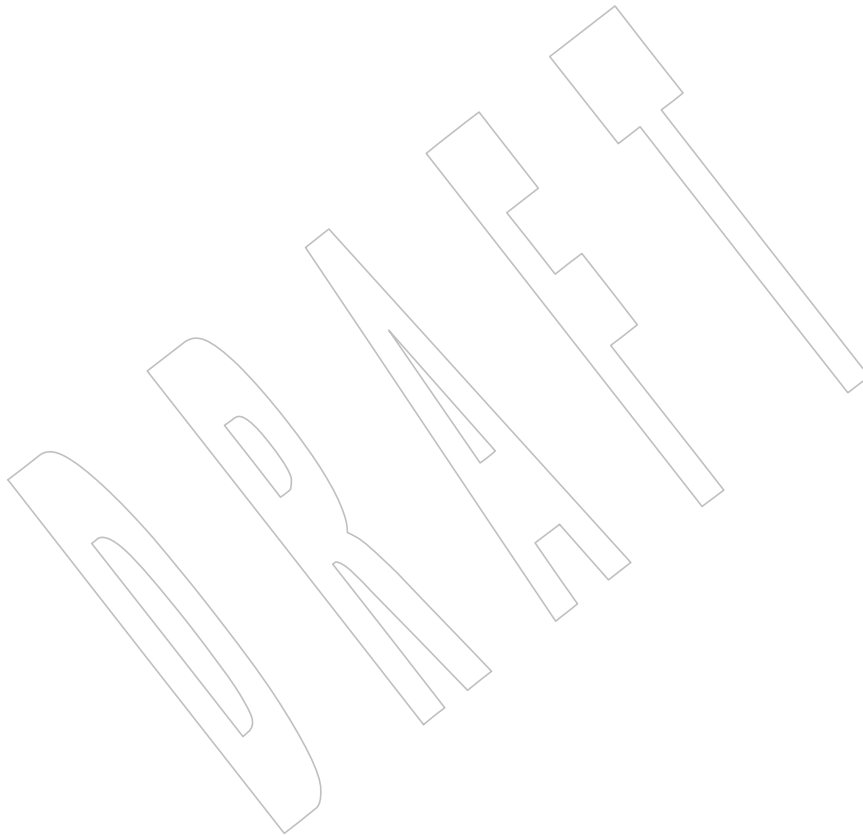
5673 None.

5674 **SEE ALSO**5675 *hcreate()*, *lsearch()*, *qsort()*, *tsearch()*, the Base Definitions volume of IEEE Std 1003.1-200x,
5676 `<stdlib.h>`5677 **CHANGE HISTORY**

5678 First released in Issue 1. Derived from Issue 1 of the SVID.

5679 **Issue 6**

5680 The normative text is updated to avoid use of the term “must” for application requirements.

5681 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/11 is applied, adding to the
5682 DESCRIPTION the last sentence of the first non-shaded paragraph, and the following three
5683 paragraphs. The RATIONALE section is also updated. These changes are for alignment with the
5684 ISO C standard.

5685 **NAME**5686 `btowc` — single byte to wide character conversion5687 **SYNOPSIS**5688 `#include <stdio.h>`5689 `#include <wchar.h>`5690 `wint_t btowc(int c);`5691 **DESCRIPTION**

5692 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 5693 conflict between the requirements described here and the ISO C standard is unintentional. This
 5694 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5695 The `btowc()` function shall determine whether `c` constitutes a valid (one-byte) character in the
 5696 initial shift state.

5697 The behavior of this function shall be affected by the `LC_CTYPE` category of the current locale.

5698 **RETURN VALUE**

5699 The `btowc()` function shall return `WEOF` if `c` has the value `EOF` or if (**unsigned char**) `c` does not
 5700 constitute a valid (one-byte) character in the initial shift state. Otherwise, it shall return the
 5701 wide-character representation of that character.

5702 **ERRORS**

5703 No errors are defined.

5704 **EXAMPLES**

5705 None.

5706 **APPLICATION USAGE**

5707 None.

5708 **RATIONALE**

5709 None.

5710 **FUTURE DIRECTIONS**

5711 None.

5712 **SEE ALSO**5713 [*wctob\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<wchar.h>`5714 **CHANGE HISTORY**

5715 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
 5716 (E).

5717 **NAME**
 5718 cabs, cabsf, cabsl — return a complex absolute value

5719 **SYNOPSIS**
 5720 #include <complex.h>
 5721 double cabs(double complex z);
 5722 float cabsf(float complex z);
 5723 long double cabsl(long double complex z);

5724 **DESCRIPTION**
 5725 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 5726 conflict between the requirements described here and the ISO C standard is unintentional. This
 5727 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5728 These functions shall compute the complex absolute value (also called norm, modulus, or
 5729 magnitude) of z.

5730 **RETURN VALUE**
 5731 These functions shall return the complex absolute value.

5732 **ERRORS**
 5733 No errors are defined.

5734 **EXAMPLES**
 5735 None.

5736 **APPLICATION USAGE**
 5737 None.

5738 **RATIONALE**
 5739 None.

5740 **FUTURE DIRECTIONS**
 5741 None.

5742 **SEE ALSO**
 5743 The Base Definitions volume of IEEE Std 1003.1-200x, <complex.h>

5744 **CHANGE HISTORY**
 5745 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

5746 **NAME**
 5747 `cacos`, `cacosf`, `cacosl` — complex arc cosine functions

5748 **SYNOPSIS**
 5749 `#include <complex.h>`
 5750 `double complex cacos(double complex z);`
 5751 `float complex cacosf(float complex z);`
 5752 `long double complex cacosl(long double complex z);`

5753 **DESCRIPTION**
 5754 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 5755 conflict between the requirements described here and the ISO C standard is unintentional. This
 5756 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5757 These functions shall compute the complex arc cosine of z , with branch cuts outside the interval
 5758 $[-1, +1]$ along the real axis.

5759 **RETURN VALUE**
 5760 These functions shall return the complex arc cosine value, in the range of a strip mathematically
 5761 unbounded along the imaginary axis and in the interval $[0, \pi]$ along the real axis.

5762 **ERRORS**
 5763 No errors are defined.

5764 **EXAMPLES**
 5765 None.

5766 **APPLICATION USAGE**
 5767 None.

5768 **RATIONALE**
 5769 None.

5770 **FUTURE DIRECTIONS**
 5771 None.

5772 **SEE ALSO**
 5773 `ccos()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<complex.h>`

5774 **CHANGE HISTORY**
 5775 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5776 **NAME**
 5777 cacosh, cacoshf, cacoshl — complex arc hyperbolic cosine functions

5778 **SYNOPSIS**
 5779 #include <complex.h>
 5780 double complex cacosh(double complex z);
 5781 float complex cacoshf(float complex z);
 5782 long double complex cacoshl(long double complex z);

5783 **DESCRIPTION**
 5784 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 5785 conflict between the requirements described here and the ISO C standard is unintentional. This
 5786 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5787 These functions shall compute the complex arc hyperbolic cosine of z , with a branch cut at
 5788 values less than 1 along the real axis.

5789 **RETURN VALUE**
 5790 These functions shall return the complex arc hyperbolic cosine value, in the range of a half-strip
 5791 of non-negative values along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.

5792 **ERRORS**
 5793 No errors are defined.

5794 **EXAMPLES**
 5795 None.

5796 **APPLICATION USAGE**
 5797 None.

5798 **RATIONALE**
 5799 None.

5800 **FUTURE DIRECTIONS**
 5801 None.

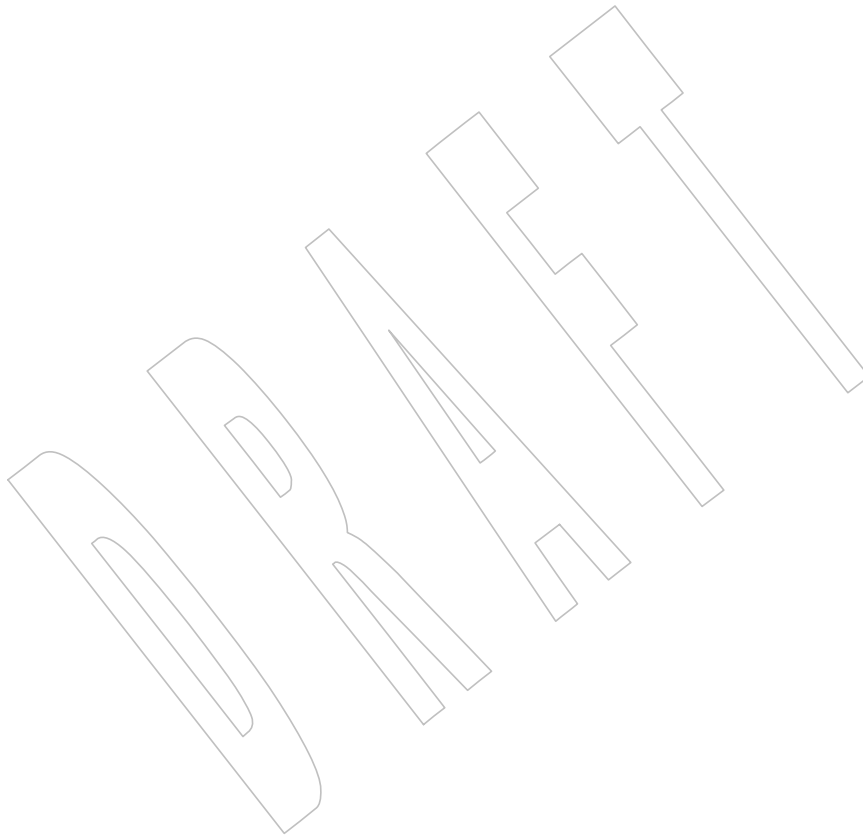
5802 **SEE ALSO**
 5803 *ccosh()*, the Base Definitions volume of IEEE Std 1003.1-200x, <complex.h>

5804 **CHANGE HISTORY**
 5805 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5806 **NAME**
5807 `cacosl` — complex arc cosine functions

5808 **SYNOPSIS**
5809 `#include <complex.h>`
5810 `long double complex cacosl(long double complex z);`

5811 **DESCRIPTION**
5812 Refer to *cacos()*.



5813 **NAME**

5814 calloc — a memory allocator

5815 **SYNOPSIS**

5816 #include <stdlib.h>

5817 void *calloc(size_t *nelem*, size_t *elsize*);5818 **DESCRIPTION**

5819 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 5820 conflict between the requirements described here and the ISO C standard is unintentional. This
 5821 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5822 The *calloc()* function shall allocate unused space for an array of *nelem* elements each of whose
 5823 size in bytes is *elsize*. The space shall be initialized to all bits 0.

5824 The order and contiguity of storage allocated by successive calls to *calloc()* is unspecified. The
 5825 pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to
 5826 a pointer to any type of object and then used to access such an object or an array of such objects
 5827 in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall
 5828 yield a pointer to an object disjoint from any other object. The pointer returned shall point to the
 5829 start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer
 5830 shall be returned. If the size of the space requested is 0, the behavior is implementation-defined:
 5831 the value returned shall be either a null pointer or a unique pointer.

5832 **RETURN VALUE**

5833 Upon successful completion with both *nelem* and *elsize* non-zero, *calloc()* shall return a pointer to
 5834 the allocated space. If either *nelem* or *elsize* is 0, then either a null pointer or a unique pointer
 5835 value that can be successfully passed to *free()* shall be returned. Otherwise, it shall return a null
 5836 pointer and set *errno* to indicate the error.

5837 **ERRORS**5838 The *calloc()* function shall fail if:

5839 CX [ENOMEM] Insufficient memory is available.

5840 **EXAMPLES**

5841 None.

5842 **APPLICATION USAGE**

5843 There is now no requirement for the implementation to support the inclusion of <malloc.h>.

5844 **RATIONALE**

5845 None.

5846 **FUTURE DIRECTIONS**

5847 None.

5848 **SEE ALSO**5849 *free()*, *malloc()*, *realloc()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdlib.h>5850 **CHANGE HISTORY**

5851 First released in Issue 1. Derived from Issue 1 of the SVID.

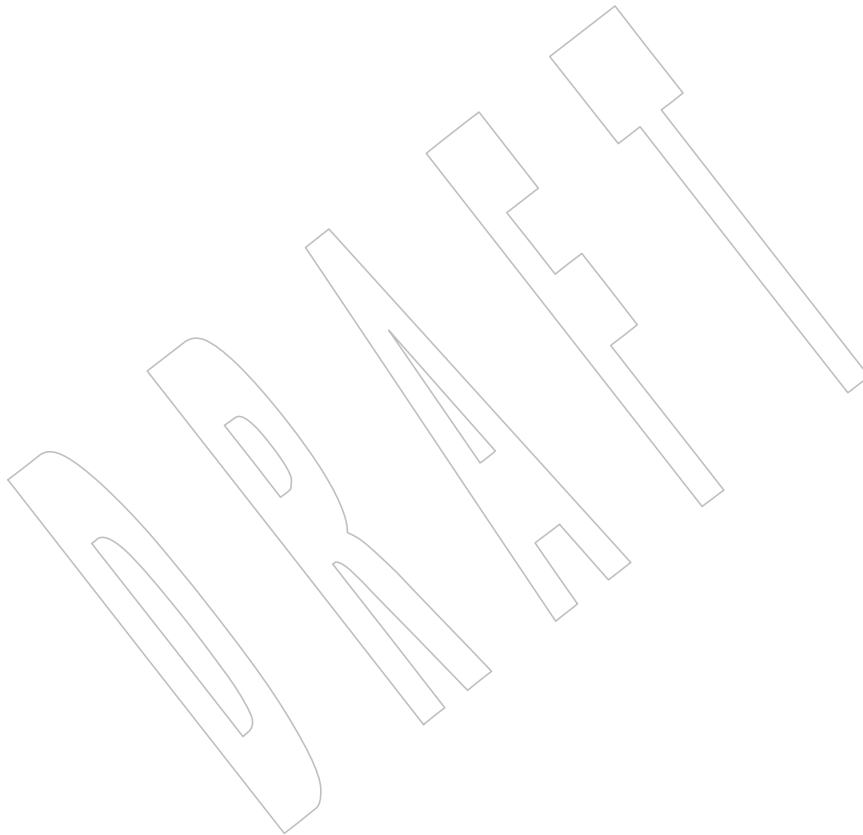
5852 **Issue 6**

5853 Extensions beyond the ISO C standard are marked.

calloc()5854
5855
5856
5857

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The setting of *errno* and the [ENOMEM] error condition are mandatory if an insufficient memory condition occurs.



5858 **NAME**
 5859 `carg, cargf, cargl` — complex argument functions

5860 **SYNOPSIS**
 5861 `#include <complex.h>`
 5862 `double carg(double complex z);`
 5863 `float cargf(float complex z);`
 5864 `long double cargl(long double complex z);`

5865 **DESCRIPTION**
 5866 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 5867 conflict between the requirements described here and the ISO C standard is unintentional. This
 5868 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5869 These functions shall compute the argument (also called phase angle) of z , with a branch cut
 5870 along the negative real axis.

5871 **RETURN VALUE**
 5872 These functions shall return the value of the argument in the interval $[-\pi, +\pi]$.

5873 **ERRORS**
 5874 No errors are defined.

5875 **EXAMPLES**
 5876 None.

5877 **APPLICATION USAGE**
 5878 None.

5879 **RATIONALE**
 5880 None.

5881 **FUTURE DIRECTIONS**
 5882 None.

5883 **SEE ALSO**
 5884 *cimag()*, *conj()*, *cproj()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

5885 **CHANGE HISTORY**
 5886 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5887 **NAME**
 5888 `casin, casinf, casinl` — complex arc sine functions

5889 **SYNOPSIS**
 5890 `#include <complex.h>`
 5891 `double complex casin(double complex z);`
 5892 `float complex casinf(float complex z);`
 5893 `long double complex casinl(long double complex z);`

5894 **DESCRIPTION**
 5895 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 5896 conflict between the requirements described here and the ISO C standard is unintentional. This
 5897 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5898 These functions shall compute the complex arc sine of z , with branch cuts outside the interval
 5899 $[-1, +1]$ along the real axis.

5900 **RETURN VALUE**
 5901 These functions shall return the complex arc sine value, in the range of a strip mathematically
 5902 unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis.

5903 **ERRORS**
 5904 No errors are defined.

5905 **EXAMPLES**
 5906 None.

5907 **APPLICATION USAGE**
 5908 None.

5909 **RATIONALE**
 5910 None.

5911 **FUTURE DIRECTIONS**
 5912 None.

5913 **SEE ALSO**
 5914 [*csin\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<complex.h>`

5915 **CHANGE HISTORY**
 5916 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5917 **NAME**
 5918 casinh, casinhf, casinhl — complex arc hyperbolic sine functions

5919 **SYNOPSIS**
 5920 #include <complex.h>
 5921 double complex casinh(double complex z);
 5922 float complex casinhf(float complex z);
 5923 long double complex casinhl(long double complex z);

5924 **DESCRIPTION**
 5925 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 5926 conflict between the requirements described here and the ISO C standard is unintentional. This
 5927 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5928 These functions shall compute the complex arc hyperbolic sine of z , with branch cuts outside the
 5929 interval $[-i, +i]$ along the imaginary axis.

5930 **RETURN VALUE**
 5931 These functions shall return the complex arc hyperbolic sine value, in the range of a strip
 5932 mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the
 5933 imaginary axis.

5934 **ERRORS**
 5935 No errors are defined.

5936 **EXAMPLES**
 5937 None.

5938 **APPLICATION USAGE**
 5939 None.

5940 **RATIONALE**
 5941 None.

5942 **FUTURE DIRECTIONS**
 5943 None.

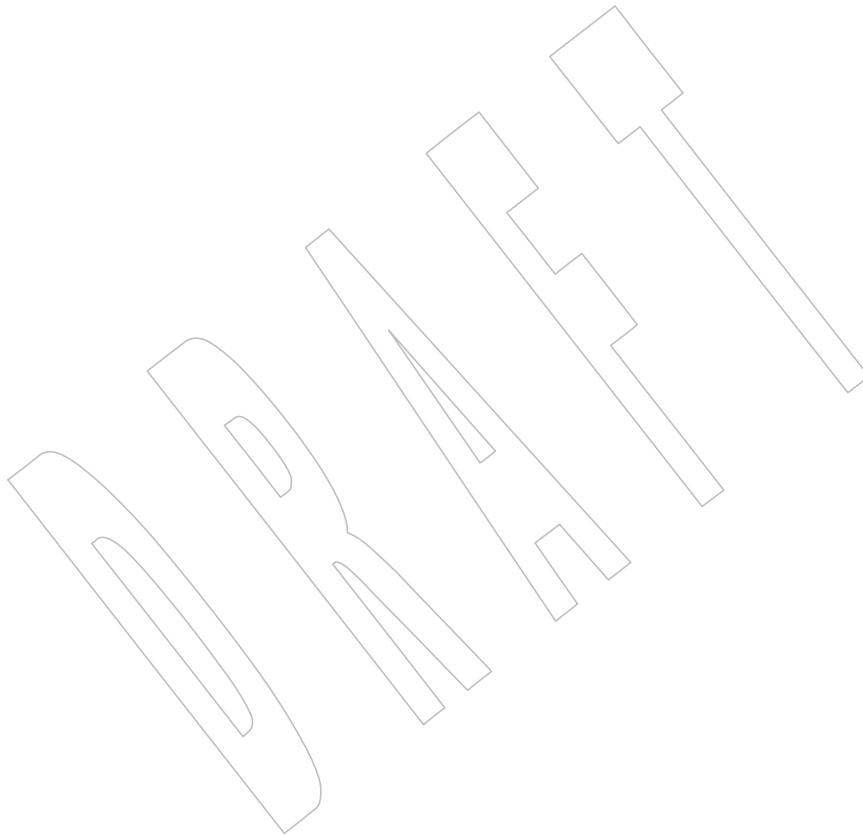
5944 **SEE ALSO**
 5945 *csinh()*, the Base Definitions volume of IEEE Std 1003.1-200x, <complex.h>

5946 **CHANGE HISTORY**
 5947 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

5948 **NAME**
5949 casinl — complex arc sine functions

5950 **SYNOPSIS**
5951 #include <complex.h>
5952 long double complex casinl(long double complex z);

5953 **DESCRIPTION**
5954 Refer to *casin()*.



5955 **NAME**
 5956 catan, catanf, catanl — complex arc tangent functions

5957 **SYNOPSIS**
 5958 #include <complex.h>
 5959 double complex catan(double complex z);
 5960 float complex catanf(float complex z);
 5961 long double complex catanl(long double complex z);

5962 **DESCRIPTION**
 5963 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 5964 conflict between the requirements described here and the ISO C standard is unintentional. This
 5965 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5966 These functions shall compute the complex arc tangent of z , with branch cuts outside the
 5967 interval $[-i, +i]$ along the imaginary axis.

5968 **RETURN VALUE**
 5969 These functions shall return the complex arc tangent value, in the range of a strip
 5970 mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the
 5971 real axis.

5972 **ERRORS**
 5973 No errors are defined.

5974 **EXAMPLES**
 5975 None.

5976 **APPLICATION USAGE**
 5977 None.

5978 **RATIONALE**
 5979 None.

5980 **FUTURE DIRECTIONS**
 5981 None.

5982 **SEE ALSO**
 5983 *ctan()*, the Base Definitions volume of IEEE Std 1003.1-200x, <complex.h>

5984 **CHANGE HISTORY**
 5985 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5986 **NAME**5987 `catanh, catanhf, catanh1` — complex arc hyperbolic tangent functions5988 **SYNOPSIS**5989 `#include <complex.h>`5990 `double complex catanh(double complex z);`5991 `float complex catanhf(float complex z);`5992 `long double complex catanh1(long double complex z);`5993 **DESCRIPTION**5994 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5995 conflict between the requirements described here and the ISO C standard is unintentional. This
5996 volume of IEEE Std 1003.1-200x defers to the ISO C standard.5997 These functions shall compute the complex arc hyperbolic tangent of z , with branch cuts outside
5998 the interval $[-1, +1]$ along the real axis.5999 **RETURN VALUE**6000 These functions shall return the complex arc hyperbolic tangent value, in the range of a strip
6001 mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the
6002 imaginary axis.6003 **ERRORS**

6004 No errors are defined.

6005 **EXAMPLES**

6006 None.

6007 **APPLICATION USAGE**

6008 None.

6009 **RATIONALE**

6010 None.

6011 **FUTURE DIRECTIONS**

6012 None.

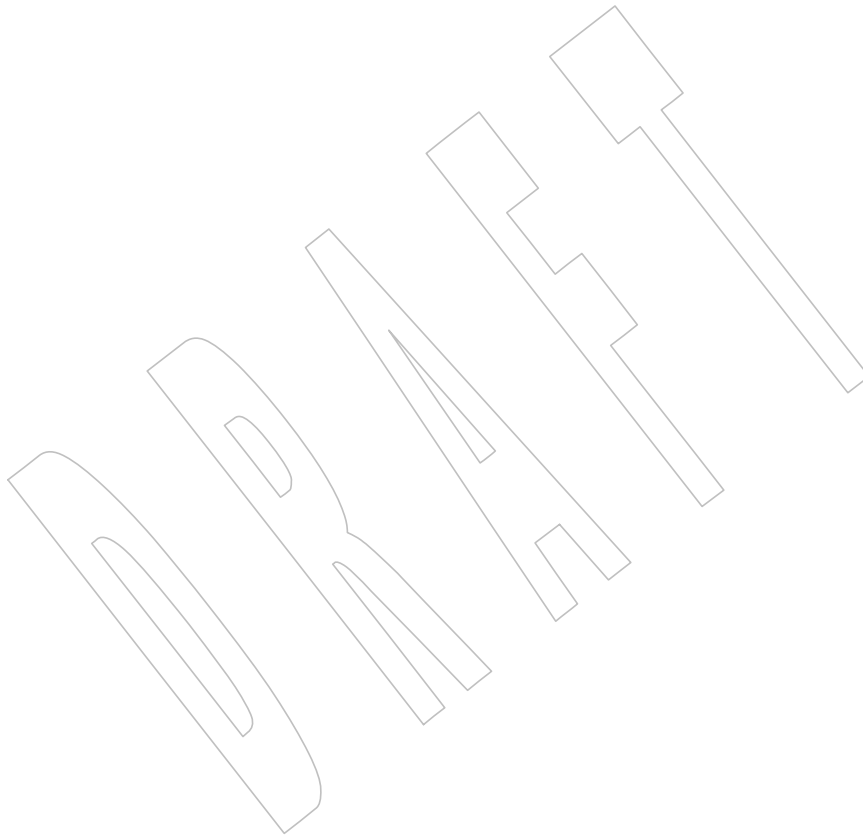
6013 **SEE ALSO**6014 `ctanh()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<complex.h>`6015 **CHANGE HISTORY**

6016 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

6017 **NAME**
6018 catanl — complex arc tangent functions

6019 **SYNOPSIS**
6020 #include <complex.h>
6021 long double complex catanl(long double complex z);

6022 **DESCRIPTION**
6023 Refer to *catan()*.



6024 **NAME**6025 `catclose` — close a message catalog descriptor6026 **SYNOPSIS**6027 `#include <nl_types.h>`6028 `int catclose(nl_catd catd);`6029 **DESCRIPTION**6030 The `catclose()` function shall close the message catalog identified by `catd`. If a file descriptor is
6031 used to implement the type **nl_catd**, that file descriptor shall be closed.6032 **RETURN VALUE**6033 Upon successful completion, `catclose()` shall return 0; otherwise, `-1` shall be returned, and `errno`
6034 set to indicate the error.6035 **ERRORS**6036 The `catclose()` function may fail if:

6037 [EBADF] The catalog descriptor is not valid.

6038 [EINTR] The `catclose()` function was interrupted by a signal.6039 **EXAMPLES**

6040 None.

6041 **APPLICATION USAGE**

6042 None.

6043 **RATIONALE**

6044 None.

6045 **FUTURE DIRECTIONS**

6046 None.

6047 **SEE ALSO**6048 [catgets\(\)](#), [catopen\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, [<nl_types.h>](#)6049 **CHANGE HISTORY**

6050 First released in Issue 2.

6051 **Issue 7**6052 The `catclose()` function is moved from the XSI option to the Base.

6053 **NAME**6054 `catgets` — read a program message6055 **SYNOPSIS**6056 `#include <nl_types.h>`6057 `char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);`6058 **DESCRIPTION**

6059 The `catgets()` function shall attempt to read message `msg_id`, in set `set_id`, from the message
 6060 catalog identified by `catd`. The `catd` argument is a message catalog descriptor returned from an
 6061 earlier call to `catopen()`. The results are undefined if `catd` is not a value returned by `catopen()` for
 6062 a message catalog still open in the process. The `s` argument points to a default message string
 6063 which shall be returned by `catgets()` if it cannot retrieve the identified message.

6064 The `catgets()` function need not be thread-safe. A function that is not required to be thread-safe is
 6065 not required to be reentrant.

6066 **RETURN VALUE**

6067 If the identified message is retrieved successfully, `catgets()` shall return a pointer to an internal
 6068 buffer area containing the null-terminated message string. If the call is unsuccessful for any
 6069 reason, `s` shall be returned and `errno` shall be set to indicate the error.

6070 **ERRORS**6071 The `catgets()` function shall fail if:

6072 [EINTR] The read operation was terminated due to the receipt of a signal, and no data
 6073 was transferred.

6074 [ENOMSG] The message identified by `set_id` and `msg_id` is not in the message catalog.

6075 The `catgets()` function may fail if:

6076 [EBADF] The `catd` argument is not a valid message catalog descriptor open for reading.

6077 [EBADMSG] The message identified by `set_id` and `msg_id` in the specified message catalog
 6078 did not satisfy implementation-defined security criteria.

6079 [EINVAL] The message catalog identified by `catd` is corrupted.

6080 **EXAMPLES**

6081 None.

6082 **APPLICATION USAGE**

6083 None.

6084 **RATIONALE**

6085 None.

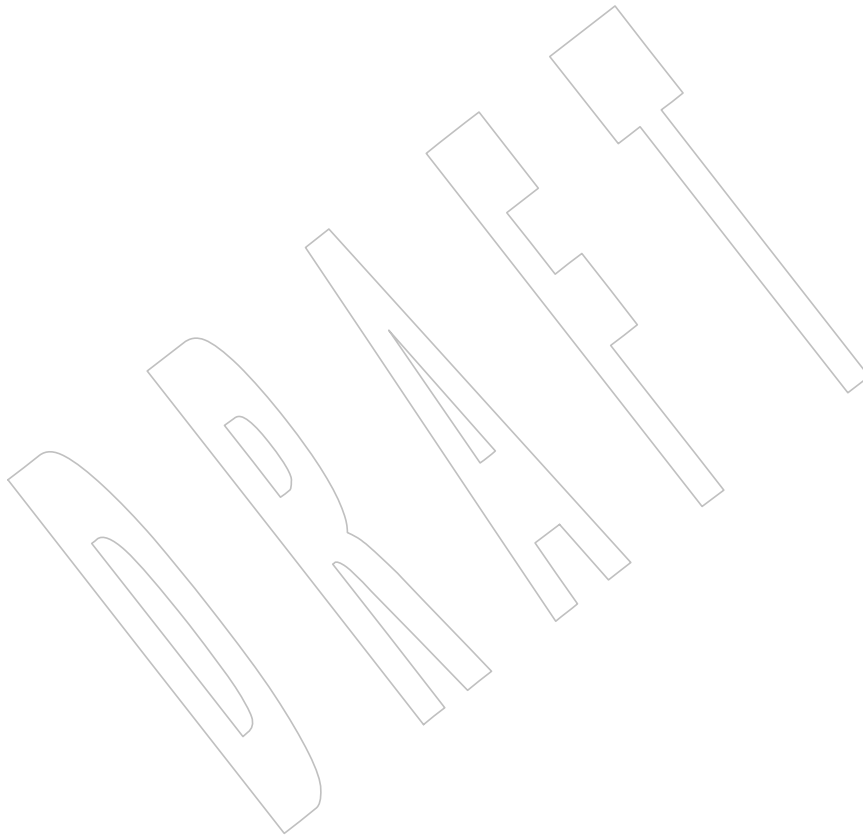
6086 **FUTURE DIRECTIONS**

6087 None.

6088 **SEE ALSO**6089 `catclose()`, `catopen()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<nl_types.h>`6090 **CHANGE HISTORY**

6091 First released in Issue 2.

- 6092 **Issue 5**
- 6093 A note indicating that this function need not be reentrant is added to the DESCRIPTION.
- 6094 **Issue 6**
- 6095 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.
- 6096 **Issue 7**
- 6097 Austin Group Interpretation 1003.1-2001 #044 is applied, changing the “may fail” [EINTR] and
- 6098 [ENOMSG] errors to become “shall fail” errors, updating the RETURN VALUE section, and
- 6099 updating the DESCRIPTION to note that: “The results are undefined if *catd* is not a value
- 6100 returned by *catopen()* for a message catalog still open in the process.
- 6101 The *catgets()* function is moved from the XSI option to the Base.



6102 **NAME**6103 `catopen` — open a message catalog6104 **SYNOPSIS**6105 `#include <nl_types.h>`6106 `nl_catd catopen(const char *name, int oflag);`6107 **DESCRIPTION**

6108 The `catopen()` function shall open a message catalog and return a message catalog descriptor.
 6109 The `name` argument specifies the name of the message catalog to be opened. If `name` contains a
 6110 `'/'`, then `name` specifies a complete name for the message catalog. Otherwise, the environment
 6111 variable `NLSPATH` is used with `name` substituted for the `%N` conversion specification (see the
 6112 Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables). If
 6113 `NLSPATH` exists in the environment when the process starts, then if the process has appropriate
 6114 privileges, the behavior of `catopen()` is undefined. If `NLSPATH` does not exist in the environment,
 6115 or if a message catalog cannot be found in any of the components specified by `NLSPATH`, then
 6116 an implementation-defined default path shall be used. This default may be affected by the
 6117 setting of `LC_MESSAGES` if the value of `oflag` is `NL_CAT_LOCALE`, or the `LANG` environment
 6118 variable if `oflag` is 0.

6119 A message catalog descriptor shall remain valid in a process until that process closes it, or a
 6120 successful call to one of the `exec` functions. A change in the setting of the `LC_MESSAGES`
 6121 category may invalidate existing open catalogs.

6122 If a file descriptor is used to implement message catalog descriptors, the `FD_CLOEXEC` flag
 6123 shall be set; see `<fcntl.h>`.

6124 If the value of the `oflag` argument is 0, the `LANG` environment variable is used to locate the
 6125 catalog without regard to the `LC_MESSAGES` category. If the `oflag` argument is
 6126 `NL_CAT_LOCALE`, the `LC_MESSAGES` category is used to locate the message catalog (see the
 6127 Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables).

6128 **RETURN VALUE**

6129 Upon successful completion, `catopen()` shall return a message catalog descriptor for use on
 6130 subsequent calls to `catgets()` and `catclose()`. Otherwise, `catopen()` shall return `(nl_catd) -1` and set
 6131 `errno` to indicate the error.

6132 **ERRORS**6133 The `catopen()` function may fail if:

6134 [EACCES] Search permission is denied for the component of the path prefix of the
 6135 message catalog or read permission is denied for the message catalog.

6136 [EMFILE] All file descriptors available to the process are currently open.

6137 [ENAMETOOLONG]

6138 The length of a pathname of the message catalog exceeds `{PATH_MAX}` or a
 6139 pathname component is longer than `{NAME_MAX}`.

6140 [ENAMETOOLONG]

6141 Pathname resolution of a symbolic link produced an intermediate result
 6142 whose length exceeds `{PATH_MAX}`.

6143 [ENFILE] Too many files are currently open in the system.

6144 [ENOENT] The message catalog does not exist or the `name` argument points to an empty
 6145 string.

6146 [ENOMEM] Insufficient storage space is available.

6147 [ENOTDIR] A component of the path prefix of the message catalog is not a directory.

6148 **EXAMPLES**

6149 None.

6150 **APPLICATION USAGE**

6151 Some implementations of *catopen()* use *malloc()* to allocate space for internal buffer areas. The
6152 *catopen()* function may fail if there is insufficient storage space available to accommodate these
6153 buffers.

6154 Conforming applications must assume that message catalog descriptors are not valid after a call
6155 to one of the *exec* functions.

6156 Application writers should be aware that guidelines for the location of message catalogs have
6157 not yet been developed. Therefore they should take care to avoid conflicting with catalogs used
6158 by other applications and the standard utilities.

6159 **RATIONALE**

6160 None.

6161 **FUTURE DIRECTIONS**

6162 None.

6163 **SEE ALSO**

6164 *catclose()*, *catgets()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**,
6165 **<nl_types.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x

6166 **CHANGE HISTORY**

6167 First released in Issue 2.

6168 **Issue 7**

6169 SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

6170 The *catopen()* function is moved from the XSI option to the Base.

6171 **NAME**
 6172 cbrt, cbrtf, cbrtl — cube root functions

6173 **SYNOPSIS**
 6174 #include <math.h>
 6175 double cbrt(double x);
 6176 float cbrtf(float x);
 6177 long double cbrtl(long double x);

6178 **DESCRIPTION**
 6179 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 6180 conflict between the requirements described here and the ISO C standard is unintentional. This
 6181 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

6182 These functions shall compute the real cube root of their argument x .

6183 **RETURN VALUE**
 6184 Upon successful completion, these functions shall return the cube root of x .

6185 MX If x is NaN, a NaN shall be returned.
 6186 If x is ± 0 or $\pm \text{Inf}$, x shall be returned.

6187 **ERRORS**
 6188 No errors are defined.

6189 **EXAMPLES**
 6190 None.

6191 **APPLICATION USAGE**
 6192 None.

6193 **RATIONALE**
 6194 For some applications, a true cube root function, which returns negative results for negative
 6195 arguments, is more appropriate than $\text{pow}(x, 1.0/3.0)$, which returns a NaN for x less than 0.

6196 **FUTURE DIRECTIONS**
 6197 None.

6198 **SEE ALSO**
 6199 The Base Definitions volume of IEEE Std 1003.1-200x, <math.h>

6200 **CHANGE HISTORY**
 6201 First released in Issue 4, Version 2.

6202 **Issue 5**
 6203 Moved from X/OPEN UNIX extension to BASE.

6204 **Issue 6**
 6205 The `cbrt()` function is no longer marked as an extension.
 6206 The `cbrtf()` and `cbrtl()` functions are added for alignment with the ISO/IEC 9899:1999 standard.
 6207 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
 6208 revised to align with the ISO/IEC 9899:1999 standard.
 6209 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
 6210 marked.

6211 **NAME**6212 `ccos, ccosf, ccosl` — complex cosine functions6213 **SYNOPSIS**6214 `#include <complex.h>`6215 `double complex ccos(double complex z);`6216 `float complex ccosf(float complex z);`6217 `long double complex ccosl(long double complex z);`6218 **DESCRIPTION**6219 CX The functionality described on this reference page is aligned with the ISO C standard. Any
6220 conflict between the requirements described here and the ISO C standard is unintentional. This
6221 volume of IEEE Std 1003.1-200x defers to the ISO C standard.6222 These functions shall compute the complex cosine of *z*.6223 **RETURN VALUE**

6224 These functions shall return the complex cosine value.

6225 **ERRORS**

6226 No errors are defined.

6227 **EXAMPLES**

6228 None.

6229 **APPLICATION USAGE**

6230 None.

6231 **RATIONALE**

6232 None.

6233 **FUTURE DIRECTIONS**

6234 None.

6235 **SEE ALSO**6236 *ccos()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<complex.h>`6237 **CHANGE HISTORY**

6238 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

6239 **NAME**
 6240 ccosh, ccoshf, ccoshl — complex hyperbolic cosine functions

6241 **SYNOPSIS**
 6242 #include <complex.h>
 6243 double complex ccosh(double complex z);
 6244 float complex ccoshf(float complex z);
 6245 long double complex ccoshl(long double complex z);

6246 **DESCRIPTION**
 6247 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 6248 conflict between the requirements described here and the ISO C standard is unintentional. This
 6249 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

6250 These functions shall compute the complex hyperbolic cosine of z .

6251 **RETURN VALUE**
 6252 These functions shall return the complex hyperbolic cosine value.

6253 **ERRORS**
 6254 No errors are defined.

6255 **EXAMPLES**
 6256 None.

6257 **APPLICATION USAGE**
 6258 None.

6259 **RATIONALE**
 6260 None.

6261 **FUTURE DIRECTIONS**
 6262 None.

6263 **SEE ALSO**
 6264 *ccosh()*, the Base Definitions volume of IEEE Std 1003.1-200x, <complex.h>

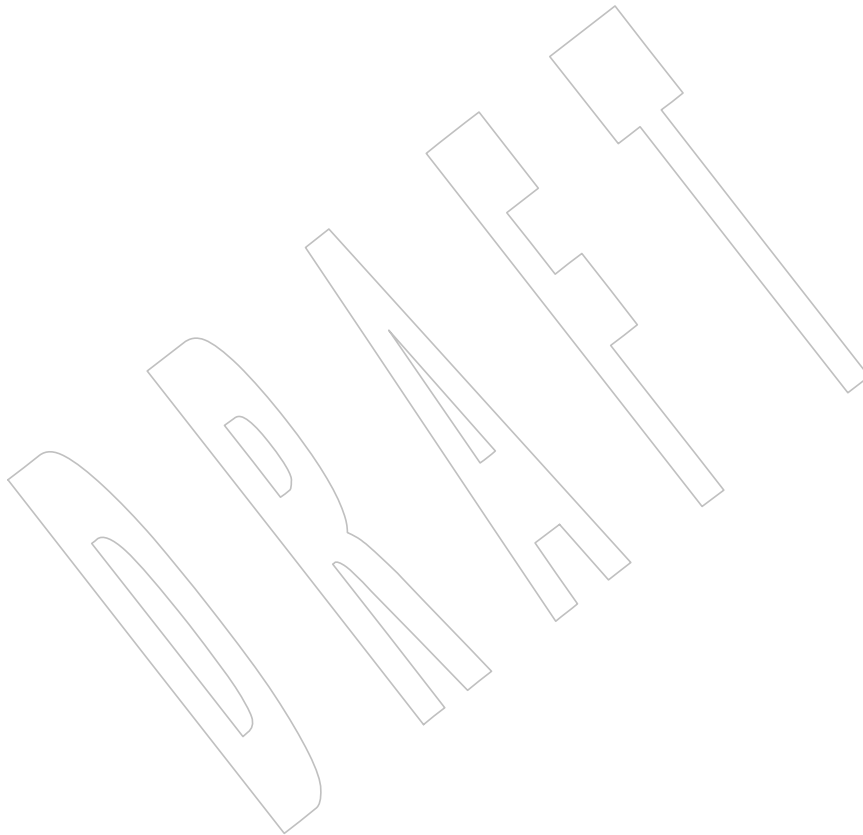
6265 **CHANGE HISTORY**
 6266 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

6267 **NAME**
6268 `ccosl` — complex cosine functions

6269 **SYNOPSIS**
6270 `#include <complex.h>`

6271 `long double complex ccosl(long double complex z);`

6272 **DESCRIPTION**
6273 Refer to `ccos()`.



6274 **NAME**

6275 ceil, ceilf, ceill — ceiling value function

6276 **SYNOPSIS**

```
6277     #include <math.h>
6278
6278     double ceil(double x);
6279     float  ceilf(float x);
6280     long double ceill(long double x);
```

6281 **DESCRIPTION**

6282 CX The functionality described on this reference page is aligned with the ISO C standard. Any
6283 conflict between the requirements described here and the ISO C standard is unintentional. This
6284 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

6285 These functions shall compute the smallest integral value not less than x .

6286 An application wishing to check for error situations should set *errno* to zero and call
6287 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
6288 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
6289 zero, an error has occurred.

6290 **RETURN VALUE**

6291 Upon successful completion, *ceil()*, *ceilf()*, and *ceill()* shall return the smallest integral value not
6292 less than x , expressed as a type **double**, **float**, or **long double**, respectively.

6293 MX If x is NaN, a NaN shall be returned.

6294 If x is ± 0 or $\pm \text{Inf}$, x shall be returned.

6295 XSI If the correct value would cause overflow, a range error shall occur and *ceil()*, *ceilf()*, and *ceill()*
6296 shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

6297 **ERRORS**

6298 These functions shall fail if:

6299 XSI **Range Error** The result overflows.

6300 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
6301 then *errno* shall be set to [ERANGE]. If the integer expression
6302 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
6303 floating-point exception shall be raised.

6304 **EXAMPLES**

6305 None.

6306 **APPLICATION USAGE**

6307 The integral value returned by these functions need not be expressible as an **int** or **long**. The
6308 return value should be tested before assigning it to an integer type to avoid the undefined
6309 results of an integer overflow.

6310 The *ceil()* function can only overflow when the floating-point representation has
6311 DBL_MANT_DIG > DBL_MAX_EXP.

6312 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
6313 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

6314
6315 None.

FUTURE DIRECTIONS

6316
6317 None.

SEE ALSO

6318 *feclearexcept()*, *fetestexcept()*, *floor()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-200x,
6319 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**
6320

CHANGE HISTORY

6321 First released in Issue 1. Derived from Issue 1 of the SVID.
6322

Issue 5

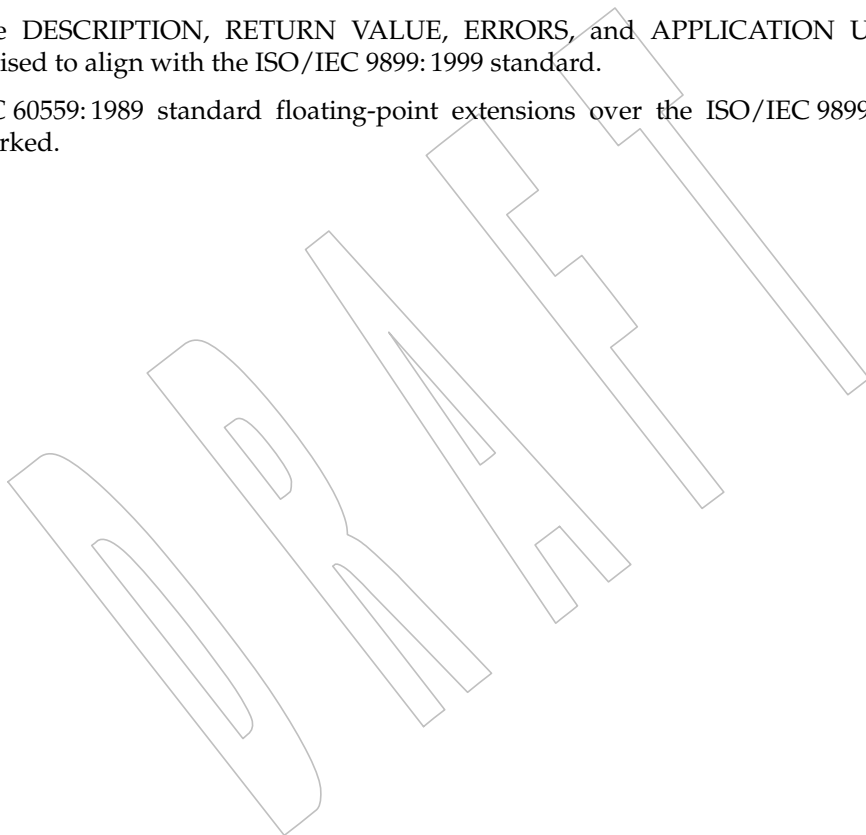
6323 The DESCRIPTION is updated to indicate how an application should check for an error. This
6324 text was previously published in the APPLICATION USAGE section.
6325

Issue 6

6326 The *ceilf()* and *ceill()* functions are added for alignment with the ISO/IEC 9899:1999 standard.
6327

6328 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
6329 revised to align with the ISO/IEC 9899:1999 standard.

6330 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
6331 marked.



6332 **NAME**
 6333 `cexp, cexpf, cexpl` — complex exponential functions

6334 **SYNOPSIS**
 6335 `#include <complex.h>`
 6336 `double complex cexp(double complex z);`
 6337 `float complex cexpf(float complex z);`
 6338 `long double complex cexpl(long double complex z);`

6339 **DESCRIPTION**
 6340 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 6341 conflict between the requirements described here and the ISO C standard is unintentional. This
 6342 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

6343 These functions shall compute the complex exponent of z , defined as e^z .

6344 **RETURN VALUE**
 6345 These functions shall return the complex exponential value of z .

6346 **ERRORS**
 6347 No errors are defined.

6348 **EXAMPLES**
 6349 None.

6350 **APPLICATION USAGE**
 6351 None.

6352 **RATIONALE**
 6353 None.

6354 **FUTURE DIRECTIONS**
 6355 None.

6356 **SEE ALSO**
 6357 [clog\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<complex.h>`

6358 **CHANGE HISTORY**
 6359 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

6360 **NAME**

6361 cfgetispeed — get input baud rate

6362 **SYNOPSIS**

6363 #include <termios.h>

6364 speed_t cfgetispeed(const struct termios *termios_p);

6365 **DESCRIPTION**6366 The *cfgetispeed()* function shall extract the input baud rate from the **termios** structure to which
6367 the *termios_p* argument points.6368 This function shall return exactly the value in the **termios** data structure, without interpretation.6369 **RETURN VALUE**6370 Upon successful completion, *cfgetispeed()* shall return a value of type **speed_t** representing the
6371 input baud rate.6372 **ERRORS**

6373 No errors are defined.

6374 **EXAMPLES**

6375 None.

6376 **APPLICATION USAGE**

6377 None.

6378 **RATIONALE**6379 The term “baud” is used historically here, but is not technically correct. This is properly “bits per
6380 second”, which may not be the same as baud. However, the term is used because of the
6381 historical usage and understanding.6382 The *cfgetospeed()*, *cfgetispeed()*, *cfsetospeed()*, and *cfsetispeed()* functions do not take arguments as
6383 numbers, but rather as symbolic names. There are two reasons for this:

- 6384 1. Historically, numbers were not used because of the way the rate was stored in the data
-
- 6385 structure. This is retained even though a function is now used.
-
- 6386 2. More importantly, only a limited set of possible rates is at all portable, and this constrains
-
- 6387 the application to that set.

6388 There is nothing to prevent an implementation accepting as an extension a number (such as 126),
6389 and since the encoding of the Bxxx symbols is not specified, this can be done to avoid
6390 introducing ambiguity.6391 Setting the input baud rate to zero was a mechanism to allow for split baud rates. Clarifications
6392 in this volume of IEEE Std 1003.1-200x have made it possible to determine whether split rates
6393 are supported and to support them without having to treat zero as a special case. Since this
6394 functionality is also confusing, it has been declared obsolescent. The 0 argument referred to is
6395 the literal constant 0, not the symbolic constant B0. This volume of IEEE Std 1003.1-200x does not
6396 preclude B0 from being defined as the value 0; in fact, implementations would likely benefit
6397 from the two being equivalent. This volume of IEEE Std 1003.1-200x does not fully specify
6398 whether the previous *cfsetispeed()* value is retained after a *tcgetattr()* as the actual value or as
6399 zero. Therefore, conforming applications should always set both the input speed and output
6400 speed when setting either.6401 In historical implementations, the baud rate information is traditionally kept in **c_cflag**.
6402 Applications should be written to presume that this might be the case (and thus not blindly copy
6403 **c_cflag**), but not to rely on it in case it is in some other field of the structure. Setting the **c_cflag**

6404 field absolutely after setting a baud rate is a non-portable action because of this. In general, the
6405 unused parts of the flag fields might be used by the implementation and should not be blindly
6406 copied from the descriptions of one terminal device to another.

FUTURE DIRECTIONS

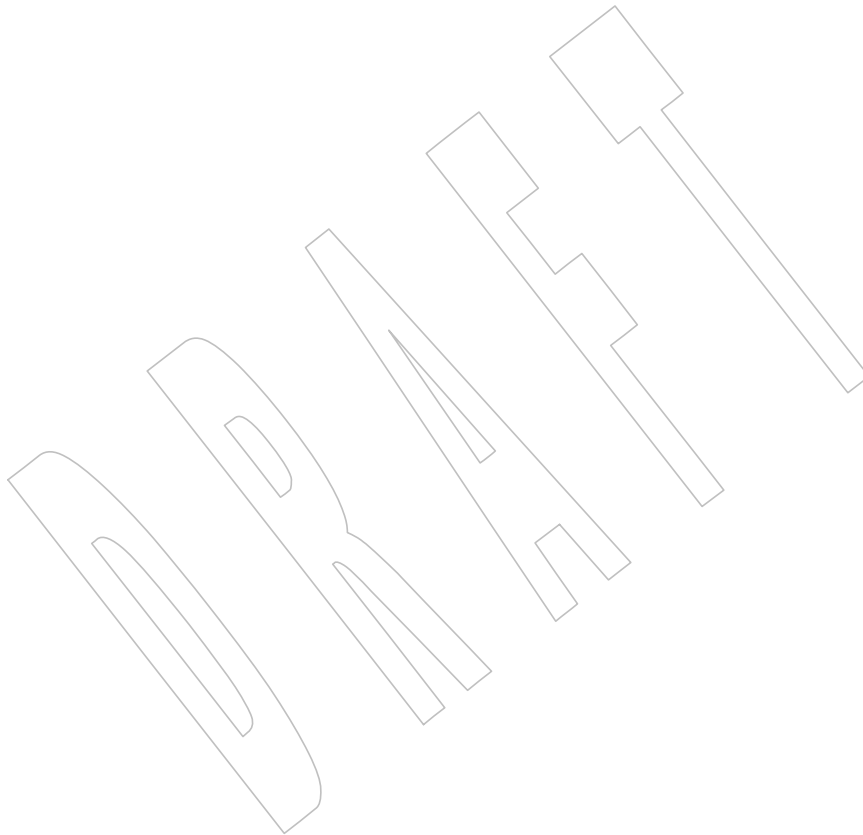
6407 None.

SEE ALSO

6409 *cfgetospeed()*, *cfsetispeed()*, *cfsetospeed()*, *tcgetattr()*, the Base Definitions volume of
6410 IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface, **<termios.h>**

CHANGE HISTORY

6412 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.
6413



6414 **NAME**

6415 cfgetospeed — get output baud rate

6416 **SYNOPSIS**

6417 #include <termios.h>

6418 speed_t cfgetospeed(const struct termios *termios_p);

6419 **DESCRIPTION**6420 The *cfgetospeed()* function shall extract the output baud rate from the **termios** structure to which
6421 the *termios_p* argument points.6422 This function shall return exactly the value in the **termios** data structure, without interpretation.6423 **RETURN VALUE**6424 Upon successful completion, *cfgetospeed()* shall return a value of type **speed_t** representing the
6425 output baud rate.6426 **ERRORS**

6427 No errors are defined.

6428 **EXAMPLES**

6429 None.

6430 **APPLICATION USAGE**

6431 None.

6432 **RATIONALE**6433 Refer to *cfgetispeed()*.6434 **FUTURE DIRECTIONS**

6435 None.

6436 **SEE ALSO**6437 *cfgetispeed()*, *cfsetispeed()*, *cfsetospeed()*, *tcgetattr()*, the Base Definitions volume of
6438 IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface, <**termios.h**>6439 **CHANGE HISTORY**

6440 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

6441 **NAME**

6442 cfsetispeed — set input baud rate

6443 **SYNOPSIS**

6444 #include <termios.h>

6445 int cfsetispeed(struct termios *termios_p, speed_t speed);

6446 **DESCRIPTION**6447 The *cfsetispeed()* function shall set the input baud rate stored in the structure pointed to by
6448 *termios_p* to *speed*.6449 There shall be no effect on the baud rates set in the hardware until a subsequent successful call
6450 to *tcsetattr()* with the same **termios** structure. Similarly, errors resulting from attempts to set
6451 baud rates not supported by the terminal device need not be detected until the *tcsetattr()*
6452 function is called.6453 **RETURN VALUE**6454 Upon successful completion, *cfsetispeed()* shall return 0; otherwise, -1 shall be returned, and
6455 *errno* may be set to indicate the error.6456 **ERRORS**6457 The *cfsetispeed()* function may fail if:6458 [EINVAL] The *speed* value is not a valid baud rate.6459 [EINVAL] The value of *speed* is outside the range of possible speed values as specified in
6460 <termios.h>.6461 **EXAMPLES**

6462 None.

6463 **APPLICATION USAGE**

6464 None.

6465 **RATIONALE**6466 Refer to *cfgetispeed()*.6467 **FUTURE DIRECTIONS**

6468 None.

6469 **SEE ALSO**6470 *cfgetispeed()*, *cfgetospeed()*, *cfsetospeed()*, *tcsetattr()*, the Base Definitions volume of
6471 IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface, <termios.h>6472 **CHANGE HISTORY**

6473 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

6474 **Issue 6**6475 The following new requirements on POSIX implementations derive from alignment with the
6476 Single UNIX Specification:

- 6477
- The optional setting of *errno* and the [EINVAL] error conditions are added.

6478 **NAME**6479 `cfsetospeed` — set output baud rate6480 **SYNOPSIS**6481 `#include <termios.h>`6482 `int cfsetospeed(struct termios *termios_p, speed_t speed);`6483 **DESCRIPTION**6484 The `cfsetospeed()` function shall set the output baud rate stored in the structure pointed to by
6485 `termios_p` to `speed`.6486 There shall be no effect on the baud rates set in the hardware until a subsequent successful call
6487 to `tcsetattr()` with the same **termios** structure. Similarly, errors resulting from attempts to set
6488 baud rates not supported by the terminal device need not be detected until the `tcsetattr()`
6489 function is called.6490 **RETURN VALUE**6491 Upon successful completion, `cfsetospeed()` shall return 0; otherwise, it shall return -1 and `errno`
6492 may be set to indicate the error.6493 **ERRORS**6494 The `cfsetospeed()` function may fail if:6495 [EINVAL] The `speed` value is not a valid baud rate.6496 [EINVAL] The value of `speed` is outside the range of possible speed values as specified in
6497 **<termios.h>**.6498 **EXAMPLES**

6499 None.

6500 **APPLICATION USAGE**

6501 None.

6502 **RATIONALE**6503 Refer to `cfgetispeed()`.6504 **FUTURE DIRECTIONS**

6505 None.

6506 **SEE ALSO**6507 `cfgetispeed()`, `cfgetospeed()`, `cfsetispeed()`, `tcsetattr()`, the Base Definitions volume of
6508 IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface, **<termios.h>**6509 **CHANGE HISTORY**

6510 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

6511 **Issue 6**6512 The following new requirements on POSIX implementations derive from alignment with the
6513 Single UNIX Specification:

- 6514
- The optional setting of `errno` and the [EINVAL] error conditions are added.

6515 **NAME**

6516 chdir — change working directory

6517 **SYNOPSIS**6518 #include <unistd.h>
6519 int chdir(const char *path);6520 **DESCRIPTION**6521 The *chdir()* function shall cause the directory named by the pathname pointed to by the *path*
6522 argument to become the current working directory; that is, the starting point for path searches
6523 for pathnames not beginning with '/ '.6524 **RETURN VALUE**6525 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned, the current
6526 working directory shall remain unchanged, and *errno* shall be set to indicate the error.6527 **ERRORS**6528 The *chdir()* function shall fail if:

- 6529 [EACCES] Search permission is denied for any component of the pathname.
- 6530 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
6531 argument.
- 6532 [ENAMETOOLONG]
6533 The length of the *path* argument exceeds {PATH_MAX} or a pathname
6534 component is longer than {NAME_MAX}.
- 6535 [ENOENT] A component of *path* does not name an existing directory or *path* is an empty
6536 string.
- 6537 [ENOTDIR] A component of the pathname is not a directory.
- 6538 The *chdir()* function may fail if:
- 6539 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
6540 resolution of the *path* argument.
- 6541 [ENAMETOOLONG]
6542 As a result of encountering a symbolic link in resolution of the *path* argument,
6543 the length of the substituted pathname string exceeded {PATH_MAX}.

6544 **EXAMPLES**6545 **Changing the Current Working Directory**6546 The following example makes the value pointed to by **directory**, */tmp*, the current working
6547 directory.6548 #include <unistd.h>
6549 ...
6550 char *directory = "/tmp";
6551 int ret;
6552 ret = chdir (directory);

6553
6554
6555
6556
6557
6558
6559
6560
6561
6562
6563
6564
6565
6566
6567
6568
6569
6570

APPLICATION USAGE

None.

RATIONALE

The *chdir()* function only affects the working directory of the current process.

FUTURE DIRECTIONS

None.

SEE ALSO

getcwd(), the Base Definitions volume of IEEE Std 1003.1-200x, <**unistd.h**>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The APPLICATION USAGE section is added.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

NAME

chmod, fchmodat — change mode of a file relative to directory file descriptor

SYNOPSIS

```
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
int fchmodat(int fd, const char *path, mode_t mode, int flag);
```

DESCRIPTION

XSI The *chmod()* function shall change S_ISUID, S_ISGID, S_ISVTX, and the file permission bits of the file named by the pathname pointed to by the *path* argument to the corresponding bits in the *mode* argument. The application shall ensure that the effective user ID of the process matches the owner of the file or the process has appropriate privileges in order to do this.

XSI S_ISUID, S_ISGID,

- 6614 [ENOTDIR] A component of the path prefix is not a directory.
- 6615 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.
- 6616 [EPERM] The effective user ID does not match the owner of the file and the process does
6617 not have appropriate privileges.
- 6618 [EROFS] The named file resides on a read-only file system.
- 6619 The *fchmodat()* function shall fail if:
- 6620 [EBADF] The *path* argument does not specify an absolute path and the *fd* argument is
6621 neither AT_FDCWD nor a valid file descriptor open for searching.
- 6622 These functions may fail if:
- 6623 [EINTR] A signal was caught during execution of the function.
- 6624 [EINVAL] The value of the *mode* argument is invalid.
- 6625 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
6626 resolution of the *path* argument.
- 6627 [ENAMETOOLONG]
6628 As a result of encountering a symbolic link in resolution of the *path* argument,
6629 the length of the substituted pathname strings exceeded {PATH_MAX}.
- 6630 The *fchmodat()* function may fail if:
- 6631 [EINVAL] The value of the *flag* argument is invalid.
- 6632 [ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a
6633 file descriptor associated with a directory.
- 6634 [EOPNOTSUPP] The AT_SYMLINK_NOFOLLOW bit is set in the *flag* argument, *path* names a
6635 symbolic link, and the system does not support changing the mode of a
6636 symbolic link.

EXAMPLES**Setting Read Permissions for User, Group, and Others**

6638 The following example sets read permissions for the owner, group, and others.

```
6639 #include <sys/stat.h>
6640 const char *path;
6641 ...
6642 chmod(path, S_IRUSR | S_IRGRP | S_IROTH);
```

Setting Read, Write, and Execute Permissions for the Owner Only

6644 The following example sets read, write, and execute permissions for the owner, and no
6645 permissions for group and others.

```
6646 #include <sys/stat.h>
6647 const char *path;
6648 ...
6649 chmod(path, S_IRWXU);
```

6651 **Setting Different Permissions for Owner, Group, and Other**

6652 The following example sets owner permissions for CHANGEFILE to read, write, and execute,
6653 group permissions to read and execute, and other permissions to read.

```
6654 #include <sys/stat.h>
6655 #define CHANGEFILE "/etc/myfile"
6656 ...
6657 chmod(CHANGEFILE, S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH);
```

6658 **Setting and Checking File Permissions**

6659 The following example sets the file permission bits for a file named `/home/cnd/mod1`, then calls
6660 the `stat()` function to verify the permissions.

```
6661 #include <sys/types.h>
6662 #include <sys/stat.h>
6663
6664 int status;
6665 struct stat buffer
6666 ...
6667 chmod("home/cnd/mod1", S_IRWXU|S_IRWXG|S_IROTH|S_IWOTH);
6668 status = stat("home/cnd/mod1", &buffer);
```

6668 **APPLICATION USAGE**

6669 In order to ensure that the `S_ISUID` and `S_ISGID` bits are set, an application requiring this
6670 should use `stat()` after a successful `chmod()` to verify this.

6671 Any file descriptors currently open by any process on the file could possibly become invalid if
6672 the mode of the file is changed to a value which would deny access to that process. One
6673 situation where this could occur is on a stateless file system. This behavior will not occur in a
6674 conforming environment.

6675 **RATIONALE**

6676 This volume of IEEE Std 1003.1-200x specifies that the `S_ISGID` bit is cleared by `chmod()` on a
6677 regular file under certain conditions. This is specified on the assumption that regular files may
6678 be executed, and the system should prevent users from making executable `setgid()` files perform
6679 with privileges that the caller does not have. On implementations that support execution of
6680 other file types, the `S_ISGID` bit should be cleared for those file types under the same
6681 circumstances.

6682 Implementations that use the `S_ISUID` bit to indicate some other function (for example,
6683 mandatory record locking) on non-executable files need not clear this bit on writing. They
6684 should clear the bit for executable files and any other cases where the bit grants special powers
6685 to processes that change the file contents. Similar comments apply to the `S_ISGID` bit.

6686 The purpose of the `fchmodat()` function is to enable changing the mode of files in directories
6687 other than the current working directory without exposure to race conditions. Any part of the
6688 path of a file could be changed in parallel to a call to `chmod()`, resulting in unspecified behavior.
6689 By opening a file descriptor for the target directory and using the `fchmodat()` function it can be
6690 guaranteed that the changed file is located relative to the desired directory. Some
6691 implementations might allow changing the mode of symbolic links. This is not supported by the
6692 interfaces in the POSIX specification. Systems with such support provide an interface named
6693 `lchmod()`. To support such implementations `fchmodat()` has a `flag` parameter.

6694
6695
6696
6697
6698
6699
6700
6701
6702
6703
6704
6705
6706
6707
6708
6709
6710
6711
6712
6713
6714

FUTURE DIRECTIONS

None.

SEE ALSO

access(), *chown()*, *exec*, *fstatat()*, *mkdir()*, *mkfifo()*, *mknod()*, *open()*, *statfs()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<fcntl.h>`, `<sys/stat.h>`, `<sys/types.h>`

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The [EINVAL] and [EINTR] optional error conditions are added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

The normative text is updated to avoid use of the term “must” for application requirements.

Issue 7

The *fchmodat()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

6715 **NAME**

6716 chown, fchownat — change owner and group of a file relative to directory file descriptor

6717 **SYNOPSIS**

6718 #include <unistd.h>

```
6719 int chown(const char *path, uid_t owner, gid_t group);
6720 int fchownat(int fd, const char *path, uid_t owner, gid_t group,
6721             int flag);
```

6722 **DESCRIPTION**6723 The *chown()* function shall change the user and group ownership of a file.6724 The *path* argument points to a pathname naming a file. The user ID and group ID of the named file shall be set to the numeric values contained in *owner* and *group*, respectively.6725
6726 Only processes with an effective user ID equal to the user ID of the file or with appropriate privileges may change the ownership of a file. If `_POSIX_CHOWN_RESTRICTED` is in effect for *path*:

- 6729 • Changing the user ID is restricted to processes with appropriate privileges.
- 6730 • Changing the group ID is permitted to a process with an effective user ID equal to the user ID of the file, but without appropriate privileges, if and only if *owner* is equal to the file's user ID or `(uid_t)-1` and *group* is equal either to the calling process' effective group ID or to one of its supplementary group IDs.

6734 If the specified file is a regular file, one or more of the `S_IXUSR`, `S_IXGRP`, or `S_IXOTH` bits of the file mode are set, and the process does not have appropriate privileges, the set-user-ID (`S_ISUID`) and set-group-ID (`S_ISGID`) bits of the file mode shall be cleared upon successful return from *chown()*. If the specified file is a regular file, one or more of the `S_IXUSR`, `S_IXGRP`, or `S_IXOTH` bits of the file mode are set, and the process has appropriate privileges, it is implementation-defined whether the set-user-ID and set-group-ID bits are altered. If the *chown()* function is successfully invoked on a file that is not a regular file and one or more of the `S_IXUSR`, `S_IXGRP`, or `S_IXOTH` bits of the file mode are set, the set-user-ID and set-group-ID bits may be cleared.

6743 If *owner* or *group* is specified as `(uid_t)-1` or `(gid_t)-1`, respectively, the corresponding ID of the file shall not be changed. If both *owner* and *group* are `-1`, the times need not be updated.6745 Upon successful completion, *chown()* shall mark for update the *st_ctime* field of the file.6746 The *fchownat()* function shall be equivalent to the *chown()* and *lchown()* functions except in the case where *path* specifies a relative path. In this case the file to be changed is determined relative to the directory associated with the file descriptor *fd* instead of the current working directory. It is unspecified whether directory searches are permitted based on whether the file was opened with search permission or on the current permissions of the directory underlying the file descriptor.6752 Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in `<fcntl.h>`:6754 `AT_SYMLINK_NOFOLLOW`6755 If *path* names a symbolic link, ownership of the symbolic link is changed.6756 If *fchownat()* is passed the special value `AT_FDCWD` in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *chown()* or *lchown()* respectively, depending on whether or not the `AT_SYMLINK_NOFOLLOW` bit is set in the *flag* argument.

6759

RETURN VALUE

6760

Upon successful completion, these functions shall return 0. Otherwise, these functions shall return -1 and set *errno* to indicate the error. If -1 is returned, no changes are made in the user ID and group ID of the file.

6761

6762

6763

ERRORS

6764

These functions shall fail if:

6765

[EACCES] Search permission is denied on a component of the path prefix.

6766

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

6767

6768

[ENAMETOOLONG]

6769

The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

6770

6771

[ENOTDIR] A component of the path prefix is not a directory.

6772

[ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

6773

[EPERM] The effective user ID does not match the owner of the file, or the calling process does not have appropriate privileges and _POSIX_CHOWN_RESTRICTED indicates that such privilege is required.

6774

6775

6776

[EROFS] The named file resides on a read-only file system.

6777

The *fchownat*() function shall fail if:

6778

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

6779

6780

These functions may fail if:

6781

[EIO] An I/O error occurred while reading or writing to the file system.

6782

[EINTR] The *chown*() function was interrupted by a signal which was caught.

6783

[EINVAL] The owner or group ID supplied is not a value supported by the implementation.

6784

6785

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

6786

6787

[ENAMETOOLONG]

6788

As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

6789

6790

The *fchownat*() function may fail if:

6791

[EINVAL] The value of the *flag* argument is not valid.

6792

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

6793

6794

[EOPNOTSUPP] The *path* argument names a symbolic link and the implementation does not support setting the owner or group of a symbolic link.

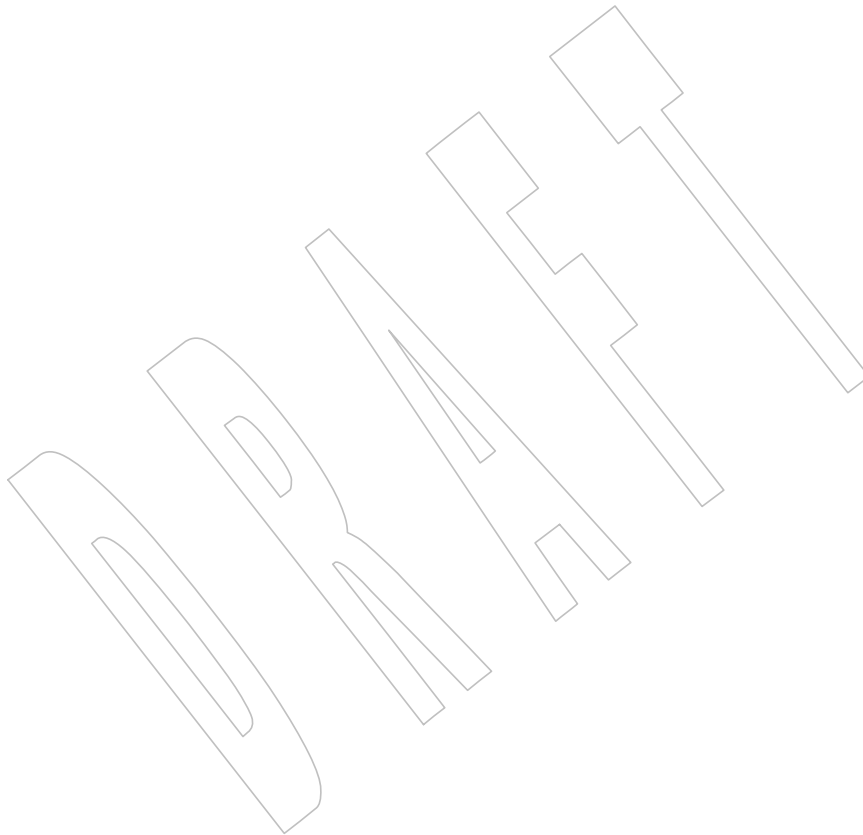
6795

EXAMPLES

None.

APPLICATION USAGE

Although *chown*



- 6843 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
- 6844 required for conforming implementations of previous POSIX specifications, it was not
- 6845 required for UNIX applications.
- 6846 • The value for *owner* of `(uid_t)-1` allows the use of `-1` by the owner of a file to change the
- 6847 group ID only. A corresponding change is made for group.
- 6848 • The [ELOOP] mandatory error condition is added.
- 6849 • The [EIO] and [EINTR] optional error conditions are added.
- 6850 • A second [ENAMETOOLONG] is added as an optional error condition.

6851 The following changes were made to align with the IEEE P1003.1a draft standard:

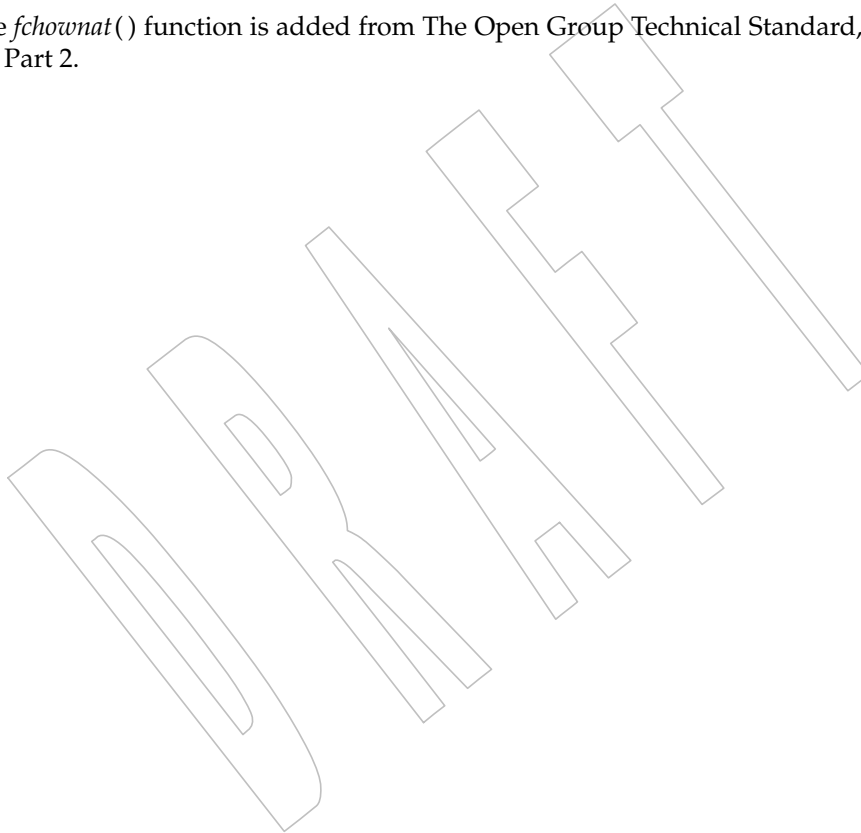
- 6852 • Clarification is added that the `S_ISUID` and `S_ISGID` bits do not need to be cleared when
- 6853 the process has appropriate privileges.
- 6854 • The [ELOOP] optional error condition is added.

Issue 7

6855 The `fchownat()` function is added from The Open Group Technical Standard, 2006, Extended API

6856 Set Part 2.

6857



6858 **NAME**
 6859 `cimag`, `cimagf`, `cimagl` — complex imaginary functions

6860 **SYNOPSIS**
 6861 `#include <complex.h>`
 6862 `double cimag(double complex z);`
 6863 `float cimagf(float complex z);`
 6864 `long double cimagl(long double complex z);`

6865 **DESCRIPTION**
 6866 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 6867 conflict between the requirements described here and the ISO C standard is unintentional. This
 6868 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

6869 These functions shall compute the imaginary part of `z`.

6870 **RETURN VALUE**
 6871 These functions shall return the imaginary part value (as a real).

6872 **ERRORS**
 6873 No errors are defined.

6874 **EXAMPLES**
 6875 None.

6876 **APPLICATION USAGE**
 6877 For a variable `z` of complex type:
 6878 `z == creal(z) + cimag(z)*I`

6879 **RATIONALE**
 6880 None.

6881 **FUTURE DIRECTIONS**
 6882 None.

6883 **SEE ALSO**
 6884 [carg\(\)](#), [conj\(\)](#), [cproj\(\)](#), [creal\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<complex.h>`

6885 **CHANGE HISTORY**
 6886 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

6887 **NAME**
 6888 clearerr — clear indicators on a stream

6889 **SYNOPSIS**
 6890 #include <stdio.h>
 6891 void clearerr(FILE *stream);

6892 **DESCRIPTION**
 6893 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 6894 conflict between the requirements described here and the ISO C standard is unintentional. This
 6895 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

6896 The *clearerr()* function shall clear the end-of-file and error indicators for the stream to which
 6897 *stream* points.

6898 **RETURN VALUE**
 6899 The *clearerr()* function shall not return a value.

6900 **ERRORS**
 6901 No errors are defined.

6902 **EXAMPLES**
 6903 None.

6904 **APPLICATION USAGE**
 6905 None.

6906 **RATIONALE**
 6907 None.

6908 **FUTURE DIRECTIONS**
 6909 None.

6910 **SEE ALSO**
 6911 The Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>

6912 **CHANGE HISTORY**
 6913 First released in Issue 1. Derived from Issue 1 of the SVID.

6914 **NAME**

6915 clock — report CPU time used

6916 **SYNOPSIS**

6917 #include <time.h>

6918 clock_t clock(void);

6919 **DESCRIPTION**

6920 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 6921 conflict between the requirements described here and the ISO C standard is unintentional. This
 6922 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

6923 The *clock()* function shall return the implementation's best approximation to the processor time
 6924 used by the process since the beginning of an implementation-defined era related only to the
 6925 process invocation.

6926 **RETURN VALUE**

6927 To determine the time in seconds, the value returned by *clock()* should be divided by the value
 6928 XSI of the macro `CLOCKS_PER_SEC`. `CLOCKS_PER_SEC` is defined to be one million in `<time.h>`.
 6929 If the processor time used is not available or its value cannot be represented, the function shall
 6930 return the value `(clock_t)-1`.

6931 **ERRORS**

6932 No errors are defined.

6933 **EXAMPLES**

6934 None.

6935 **APPLICATION USAGE**

6936 In order to measure the time spent in a program, *clock()* should be called at the start of the
 6937 program and its return value subtracted from the value returned by subsequent calls. The value
 6938 returned by *clock()* is defined for compatibility across systems that have clocks with different
 6939 resolutions. The resolution on any particular system need not be to microsecond accuracy.

6940 The value returned by *clock()* may wrap around on some implementations. For example, on a
 6941 machine with 32-bit values for `clock_t`, it wraps after 2 147 seconds or 36 minutes.

6942 **RATIONALE**

6943 None.

6944 **FUTURE DIRECTIONS**

6945 None.

6946 **SEE ALSO**

6947 *asctime()*, *ctime()*, *difftime()*, *gmtime()*, *localtime()*, *mktime()*, *strftime()*, *strptime()*, *time()*, *utime()*,
 6948 the Base Definitions volume of IEEE Std 1003.1-200x, `<time.h>`

6949 **CHANGE HISTORY**

6950 First released in Issue 1. Derived from Issue 1 of the SVID.

6951 **NAME**6952 clock_getcpuclockid — access a process CPU-time clock (**ADVANCED REALTIME**)6953 **SYNOPSIS**

```
6954 CPT #include <time.h>
6955 int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);
```

6956 **DESCRIPTION**

6957 The *clock_getcpuclockid()* function shall return the clock ID of the CPU-time clock of the process
 6958 specified by *pid*. If the process described by *pid* exists and the calling process has permission, the
 6959 clock ID of this clock shall be returned in *clock_id*.

6960 If *pid* is zero, the *clock_getcpuclockid()* function shall return the clock ID of the CPU-time clock of
 6961 the process making the call, in *clock_id*.

6962 The conditions under which one process has permission to obtain the CPU-time clock ID of
 6963 other processes are implementation-defined.

6964 **RETURN VALUE**

6965 Upon successful completion, *clock_getcpuclockid()* shall return zero; otherwise, an error number
 6966 shall be returned to indicate the error.

6967 **ERRORS**

6968 The *clock_getcpuclockid()* function shall fail if:

6969 [EPERM] The requesting process does not have permission to access the CPU-time clock
 6970 for the process.

6971 The *clock_getcpuclockid()* function may fail if:

6972 [ESRCH] No process can be found corresponding to the process specified by *pid*.

6973 **EXAMPLES**

6974 None.

6975 **APPLICATION USAGE**

6976 The *clock_getcpuclockid()* function is part of the Process CPU-Time Clocks option and need not be
 6977 provided on all implementations.

6978 **RATIONALE**

6979 None.

6980 **FUTURE DIRECTIONS**

6981 None.

6982 **SEE ALSO**

6983 *clock_getres()*, *timer_create()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

6984 **CHANGE HISTORY**

6985 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

6986 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

6987 **NAME**
 6988 clock_getres, clock_gettime, clock_settime — clock and timer functions

6989 **SYNOPSIS**

```
6990 CX #include <time.h>
6991
6991 int clock_getres(clockid_t clock_id, struct timespec *res);
6992 int clock_gettime(clockid_t clock_id, struct timespec *tp);
6993 int clock_settime(clockid_t clock_id, const struct timespec *tp);
```

6994 **DESCRIPTION**

6995 The *clock_getres()* function shall return the resolution of any clock. Clock resolutions are
 6996 implementation-defined and cannot be set by a process. If the argument *res* is not NULL, the
 6997 resolution of the specified clock shall be stored in the location pointed to by *res*. If *res* is NULL,
 6998 the clock resolution is not returned. If the *time* argument of *clock_settime()* is not a multiple of *res*,
 6999 then the value is truncated to a multiple of *res*.

7000 The *clock_gettime()* function shall return the current value *tp* for the specified clock, *clock_id*.

7001 The *clock_settime()* function shall set the specified clock, *clock_id*, to the value specified by *tp*.
 7002 Time values that are between two consecutive non-negative integer multiples of the resolution of
 7003 the specified clock shall be truncated down to the smaller multiple of the resolution.

7004 A clock may be system-wide (that is, visible to all processes) or per-process (measuring time that
 7005 is meaningful only within a process). All implementations shall support a *clock_id* of
 7006 CLOCK_REALTIME as defined in **<time.h>**. This clock represents the realtime clock for the
 7007 system. For this clock, the values returned by *clock_gettime()* and specified by *clock_settime()*
 7008 represent the amount of time (in seconds and nanoseconds) since the Epoch. An implementation
 7009 may also support additional clocks. The interpretation of time values for these clocks is
 7010 unspecified.

7011 If the value of the CLOCK_REALTIME clock is set via *clock_settime()*, the new value of the clock
 7012 shall be used to determine the time of expiration for absolute time services based upon the
 7013 CLOCK_REALTIME clock. This applies to the time at which armed absolute timers expire. If the
 7014 absolute time requested at the invocation of such a time service is before the new value of the
 7015 clock, the time service shall expire immediately as if the clock had reached the requested time
 7016 normally.

7017 Setting the value of the CLOCK_REALTIME clock via *clock_settime()* shall have no effect on
 7018 threads that are blocked waiting for a relative time service based upon this clock, including the
 7019 *nanosleep()* function; nor on the expiration of relative timers based upon this clock.
 7020 Consequently, these time services shall expire when the requested relative interval elapses,
 7021 independently of the new or old value of the clock.

7022 MON If the Monotonic Clock option is supported, all implementations shall support a *clock_id* of
 7023 CLOCK_MONOTONIC defined in **<time.h>**. This clock represents the monotonic clock for the
 7024 system. For this clock, the value returned by *clock_gettime()* represents the amount of time (in
 7025 seconds and nanoseconds) since an unspecified point in the past (for example, system start-up
 7026 time, or the Epoch). This point does not change after system start-up time. The value of the
 7027 CLOCK_MONOTONIC clock cannot be set via *clock_settime()*. This function shall fail if it is
 7028 invoked with a *clock_id* argument of CLOCK_MONOTONIC.

7029 The effect of setting a clock via *clock_settime()* on armed per-process timers associated with a
 7030 clock other than CLOCK_REALTIME is implementation-defined.

7031 If the value of the CLOCK_REALTIME clock is set via *clock_settime()*, the new value of the clock

clock_getres()

7032 shall be used to determine the time at which the system shall awaken a thread blocked on an
 7033 absolute *clock_nanosleep()* call based upon the CLOCK_REALTIME clock. If the absolute time
 7034 requested at the invocation of such a time service is before the new value of the clock, the call
 7035 shall return immediately as if the clock had reached the requested time normally.

7036 Setting the value of the CLOCK_REALTIME clock via *clock_settime()* shall have no effect on any
 7037 thread that is blocked on a relative *clock_nanosleep()* call. Consequently, the call shall return
 7038 when the requested relative interval elapses, independently of the new or old value of the clock.

7039 The appropriate privilege to set a particular clock is implementation-defined.

7040 CPT If `_POSIX_CPUTIME` is defined, implementations shall support clock ID values obtained by
 7041 invoking *clock_getcpuclockid()*, which represent the CPU-time clock of a given process.
 7042 Implementations shall also support the special `clockid_t` value
 7043 `CLOCK_PROCESS_CPUTIME_ID`, which represents the CPU-time clock of the calling process
 7044 when invoking one of the *clock_**() or *timer_**() functions. For these clock IDs, the values
 7045 returned by *clock_gettime()* and specified by *clock_settime()* represent the amount of execution
 7046 time of the process associated with the clock. Changing the value of a CPU-time clock via
 7047 *clock_settime()* shall have no effect on the behavior of the sporadic server scheduling policy (see
 7048 [Scheduling Policies](#) (on page 44)).

7049 TCT If `_POSIX_THREAD_CPUTIME` is defined, implementations shall support clock ID values
 7050 obtained by invoking *pthread_getcpuclockid()*, which represent the CPU-time clock of a given
 7051 thread. Implementations shall also support the special `clockid_t` value
 7052 `CLOCK_THREAD_CPUTIME_ID`, which represents the CPU-time clock of the calling thread
 7053 when invoking one of the *clock_**() or *timer_**() functions. For these clock IDs, the values
 7054 returned by *clock_gettime()* and specified by *clock_settime()* shall represent the amount of
 7055 execution time of the thread associated with the clock. Changing the value of a CPU-time clock
 7056 via *clock_settime()* shall have no effect on the behavior of the sporadic server scheduling policy
 7057 (see [Scheduling Policies](#) (on page 44)).

RETURN VALUE

7058 A return value of 0 shall indicate that the call succeeded. A return value of -1 shall indicate that
 7059 an error occurred, and *errno* shall be set to indicate the error.

ERRORS

7061 The *clock_getres()*, *clock_gettime()*, and *clock_settime()* functions shall fail if:

7062 [EINVAL] The *clock_id* argument does not specify a known clock.

7063 The *clock_settime()* function shall fail if:

7064 [EINVAL] The *tp* argument to *clock_settime()* is outside the range for the given clock ID.

7065 [EINVAL] The *tp* argument specified a nanosecond value less than zero or greater than or
 7066 equal to 1 000 million.

7067 [EINVAL] The value of the *clock_id* argument is `CLOCK_MONOTONIC`.

7068 The *clock_settime()* function may fail if:

7069 [EPERM] The requesting process does not have the appropriate privilege to set the
 7070 specified clock.
 7071

EXAMPLES

None.

APPLICATION USAGE

Note that the absolute value of the monotonic clock is meaningless (because its origin is arbitrary), and thus there is no need to set it. Furthermore, realtime applications can rely on the fact that the value of this clock is never set and, therefore, that time intervals measured with this clock will not be affected by calls to *clock_settime()*.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clock_getcpuclockid(), *clock_nanosleep()*, *ctime()*, *mq_timedreceive()*, *mq_timedsend()*, *nanosleep()*, *pthread_mutex_timedlock()*, *sem_timedwait()*, *time()*, *timer_create()*, *timer_getoverrun()*, the Base Definitions volume of IEEE Std 1003.1-200x, <time.h>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

The APPLICATION USAGE section is added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Clarification is added of the effect of resetting the clock resolution.

CPU-time clocks and the *clock_getcpuclockid()* function are added for alignment with IEEE Std 1003.1d-1999.

The following changes are added for alignment with IEEE Std 1003.1j-2000:

- The DESCRIPTION is updated as follows:
 - The value returned by *clock_gettime()* for CLOCK_MONOTONIC is specified.
 - The *clock_settime()* function failing for CLOCK_MONOTONIC is specified.
 - The effects of *clock_settime()* on the *clock_nanosleep()* function with respect to CLOCK_REALTIME are specified.
- An [EINVAL] error is added to the ERRORS section, indicating that *clock_settime()* fails for CLOCK_MONOTONIC.
- The APPLICATION USAGE section notes that the CLOCK_MONOTONIC clock need not and shall not be set by *clock_settime()* since the absolute value of the CLOCK_MONOTONIC clock is meaningless.
- The *clock_nanosleep()*, *mq_timedreceive()*, *mq_timedsend()*, *pthread_mutex_timedlock()*, *sem_timedwait()*, *timer_create()*, and *timer_settime()* functions are added to the SEE ALSO section.

Issue 7

Functionality relating to the Clock Selection option is moved to the Base.

The *clock_getres()*, *clock_gettime()*, and *clock_settime()* functions are moved from the Timers option to the Base.

7115 **NAME**

7116 clock_nanosleep — high resolution sleep with specifiable clock

7117 **SYNOPSIS**

```
7118 CX #include <time.h>
7119
7119 int clock_nanosleep(clockid_t clock_id, int flags,
7120                    const struct timespec *rqtp, struct timespec *rmtp);
```

7121 **DESCRIPTION**

7122 If the flag `TIMER_ABSTIME` is not set in the *flags* argument, the `clock_nanosleep()` function shall
 7123 cause the current thread to be suspended from execution until either the time interval specified
 7124 by the *rqtp* argument has elapsed, or a signal is delivered to the calling thread and its action is to
 7125 invoke a signal-catching function, or the process is terminated. The clock used to measure the
 7126 time shall be the clock specified by *clock_id*.

7127 If the flag `TIMER_ABSTIME` is set in the *flags* argument, the `clock_nanosleep()` function shall
 7128 cause the current thread to be suspended from execution until either the time value of the clock
 7129 specified by *clock_id* reaches the absolute time specified by the *rqtp* argument, or a signal is
 7130 delivered to the calling thread and its action is to invoke a signal-catching function, or the
 7131 process is terminated. If, at the time of the call, the time value specified by *rqtp* is less than or
 7132 equal to the time value of the specified clock, then `clock_nanosleep()` shall return immediately
 7133 and the calling process shall not be suspended.

7134 The suspension time caused by this function may be longer than requested because the
 7135 argument value is rounded up to an integer multiple of the sleep resolution, or because of the
 7136 scheduling of other activity by the system. But, except for the case of being interrupted by a
 7137 signal, the suspension time for the relative `clock_nanosleep()` function (that is, with the
 7138 `TIMER_ABSTIME` flag not set) shall not be less than the time interval specified by *rqtp*, as
 7139 measured by the corresponding clock. The suspension for the absolute `clock_nanosleep()` function
 7140 (that is, with the `TIMER_ABSTIME` flag set) shall be in effect at least until the value of the
 7141 corresponding clock reaches the absolute time specified by *rqtp*, except for the case of being
 7142 interrupted by a signal.

7143 The use of the `clock_nanosleep()` function shall have no effect on the action or blockage of any
 7144 signal.

7145 The `clock_nanosleep()` function shall fail if the *clock_id* argument refers to the CPU-time clock of
 7146 the calling thread. It is unspecified whether *clock_id* values of other CPU-time clocks are allowed.

7147 **RETURN VALUE**

7148 If the `clock_nanosleep()` function returns because the requested time has elapsed, its return value
 7149 shall be zero.

7150 If the `clock_nanosleep()` function returns because it has been interrupted by a signal, it shall
 7151 return the corresponding error value. For the relative `clock_nanosleep()` function, if the *rmtp*
 7152 argument is non-NULL, the **timespec** structure referenced by it shall be updated to contain the
 7153 amount of time remaining in the interval (the requested time minus the time actually slept). If
 7154 the *rmtp* argument is NULL, the remaining time is not returned. The absolute `clock_nanosleep()`
 7155 function has no effect on the structure referenced by *rmtp*.

7156 If `clock_nanosleep()` fails, it shall return the corresponding error value.

7157 **ERRORS**7158 The *clock_nanosleep()* function shall fail if:7159 [EINTR] The *clock_nanosleep()* function was interrupted by a signal.7160 [EINVAL] The *rqtpt* argument specified a nanosecond value less than zero or greater than
7161 or equal to 1 000 million; or the `TIMER_ABSTIME` flag was specified in *flags*
7162 and the *rqtpt* argument is outside the range for the clock specified by *clock_id*;
7163 or the *clock_id* argument does not specify a known clock, or specifies the CPU-
7164 time clock of the calling thread.7165 [ENOTSUP] The *clock_id* argument specifies a clock for which *clock_nanosleep()* is not
7166 supported, such as a CPU-time clock.7167 **EXAMPLES**

7168 None.

7169 **APPLICATION USAGE**7170 Calling *clock_nanosleep()* with the value `TIMER_ABSTIME` not set in the *flags* argument and with
7171 a *clock_id* of `CLOCK_REALTIME` is equivalent to calling *nanosleep()* with the same *rqtpt* and *rmtpt*
7172 arguments.7173 **RATIONALE**7174 The *nanosleep()* function specifies that the system-wide clock `CLOCK_REALTIME` is used to
7175 measure the elapsed time for this time service. However, with the introduction of the monotonic
7176 clock `CLOCK_MONOTONIC` a new relative sleep function is needed to allow an application to
7177 take advantage of the special characteristics of this clock.7178 There are many applications in which a process needs to be suspended and then activated
7179 multiple times in a periodic way; for example, to poll the status of a non-interrupting device or
7180 to refresh a display device. For these cases, it is known that precise periodic activation cannot be
7181 achieved with a relative *sleep()* or *nanosleep()* function call. Suppose, for example, a periodic
7182 process that is activated at time T_0 , executes for a while, and then wants to suspend itself until
7183 time T_0+T , the period being T . If this process wants to use the *nanosleep()* function, it must first
7184 call *clock_gettime()* to get the current time, then calculate the difference between the current time
7185 and T_0+T and, finally, call *nanosleep()* using the computed interval. However, the process could
7186 be preempted by a different process between the two function calls, and in this case the interval
7187 computed would be wrong; the process would wake up later than desired. This problem would
7188 not occur with the absolute *clock_nanosleep()* function, since only one function call would be
7189 necessary to suspend the process until the desired time. In other cases, however, a relative sleep
7190 is needed, and that is why both functionalities are required.7191 Although it is possible to implement periodic processes using the timers interface, this
7192 implementation would require the use of signals, and the reservation of some signal numbers. In
7193 this regard, the reasons for including an absolute version of the *clock_nanosleep()* function in
7194 IEEE Std 1003.1-200x are the same as for the inclusion of the relative *nanosleep()*.7195 It is also possible to implement precise periodic processes using *pthread_cond_timedwait()*, in
7196 which an absolute timeout is specified that takes effect if the condition variable involved is never
7197 signaled. However, the use of this interface is unnatural, and involves performing other
7198 operations on mutexes and condition variables that imply an unnecessary overhead.
7199 Furthermore, *pthread_cond_timedwait()* is not available in implementations that do not support
7200 threads.7201 Although the interface of the relative and absolute versions of the new high resolution sleep
7202 service is the same *clock_nanosleep()* function, the *rmtpt* argument is only used in the relative
7203 sleep. This argument is needed in the relative *clock_nanosleep()* function to reissue the function
7204 call if it is interrupted by a signal, but it is not needed in the absolute *clock_nanosleep()* function
7205 call; if the call is interrupted by a signal, the absolute *clock_nanosleep()* function can be invoked

clock_nanosleep()*System Interfaces*

7206 again with the same *rqtp* argument used in the interrupted call.

FUTURE DIRECTIONS

7208 None.

SEE ALSO

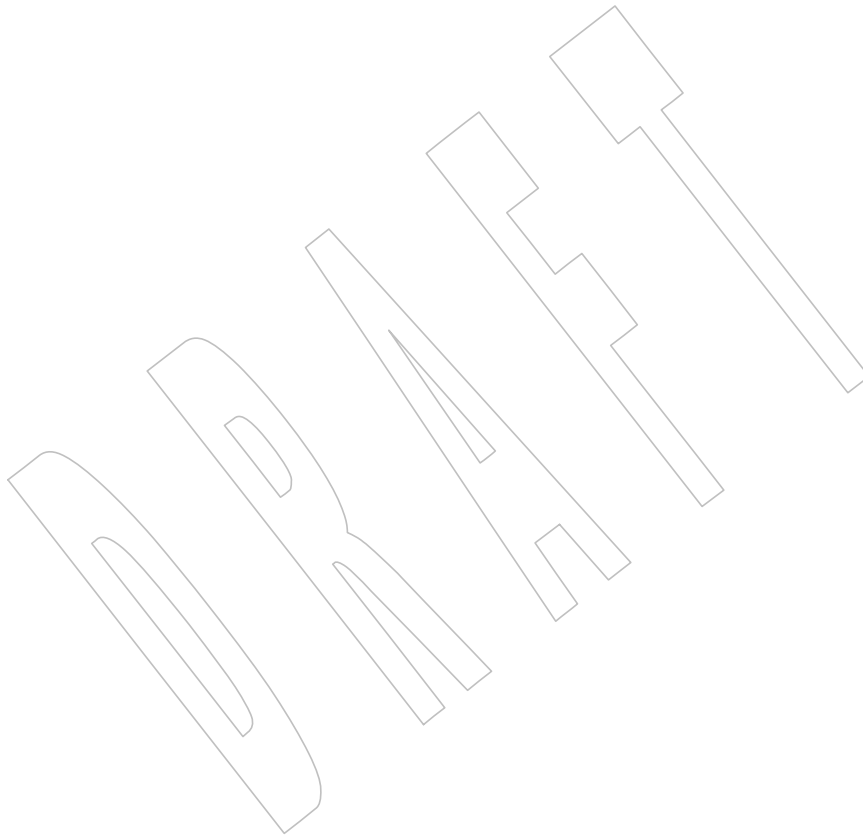
7210 [clock_getres\(\)](#), [nanosleep\(\)](#), [pthread_cond_timedwait\(\)](#), [sleep\(\)](#), the Base Definitions volume of
7211 IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

7212 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

Issue 7

7215 The *clock_nanosleep()* function is moved from the Clock Selection option to the Base.



7216 **NAME**
7217 `clock_settime` — clock and timer functions

7218 **SYNOPSIS**

```
7219 CX #include <time.h>  
7220 int clock_settime(clockid_t clock_id, const struct timespec *tp);
```

7221 **DESCRIPTION**

7222 Refer to [*clock_getres\(\)*](#).

7223 **NAME**
 7224 `clog, clogf, clogl` — complex natural logarithm functions

7225 **SYNOPSIS**
 7226 `#include <complex.h>`
 7227 `double complex clog(double complex z);`
 7228 `float complex clogf(float complex z);`
 7229 `long double complex clogl(long double complex z);`

7230 **DESCRIPTION**
 7231 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 7232 conflict between the requirements described here and the ISO C standard is unintentional. This
 7233 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

7234 These functions shall compute the complex natural (base e) logarithm of z , with a branch cut
 7235 along the negative real axis.

7236 **RETURN VALUE**
 7237 These functions shall return the complex natural logarithm value, in the range of a strip
 7238 mathematically unbounded along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary
 7239 axis.

7240 **ERRORS**
 7241 No errors are defined.

7242 **EXAMPLES**
 7243 None.

7244 **APPLICATION USAGE**
 7245 None.

7246 **RATIONALE**
 7247 None.

7248 **FUTURE DIRECTIONS**
 7249 None.

7250 **SEE ALSO**
 7251 `cexp()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<complex.h>`

7252 **CHANGE HISTORY**
 7253 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7254 **NAME**

7255 close — close a file descriptor

7256 **SYNOPSIS**

7257 #include <unistd.h>

7258 int close(int *fdes*);7259 **DESCRIPTION**

7260 The *close()* function shall deallocate the file descriptor indicated by *fdes*. To deallocate means to
 7261 make the file descriptor available for return by subsequent calls to *open()* or other functions that
 7262 allocate file descriptors. All outstanding record locks owned by the process on the file associated
 7263 with the file descriptor shall be removed (that is, unlocked).

7264 If *close()* is interrupted by a signal that is to be caught, it shall return -1 with *errno* set to [EINTR]
 7265 and the state of *fdes* is unspecified. If an I/O error occurred while reading from or writing to
 7266 the file system during *close()*, it may return -1 with *errno* set to [EIO]; if this error is returned, the
 7267 state of *fdes* is unspecified.

7268 When all file descriptors associated with a pipe or FIFO special file are closed, any data
 7269 remaining in the pipe or FIFO shall be discarded.

7270 When all file descriptors associated with an open file description have been closed, the open file
 7271 description shall be freed.

7272 If the link count of the file is 0, when all file descriptors associated with the file are closed, the
 7273 space occupied by the file shall be freed and the file shall no longer be accessible.

7274 OB XSR If a STREAMS-based *fdes* is closed and the calling process was previously registered to receive
 7275 a SIGPOLL signal for events associated with that STREAM, the calling process shall be
 7276 unregistered for events associated with the STREAM. The last *close()* for a STREAM shall cause
 7277 the STREAM associated with *fdes* to be dismantled. If O_NONBLOCK is not set and there have
 7278 been no signals posted for the STREAM, and if there is data on the module's write queue, *close()*
 7279 shall wait for an unspecified time (for each module and driver) for any output to drain before
 7280 dismantling the STREAM. The time delay can be changed via an L_SETCLTIME *ioctl()* request. If
 7281 the O_NONBLOCK flag is set, or if there are any pending signals, *close()* shall not wait for
 7282 output to drain, and shall dismantle the STREAM immediately.

7283 If the implementation supports STREAMS-based pipes, and *fdes* is associated with one end of a
 7284 pipe, the last *close()* shall cause a hangup to occur on the other end of the pipe. In addition, if the
 7285 other end of the pipe has been named by *fattach()*, then the last *close()* shall force the named end
 7286 to be detached by *fdetach()*. If the named end has no open file descriptors associated with it and
 7287 gets detached, the STREAM associated with that end shall also be dismantled.

7288 XSI If *fdes* refers to the master side of a pseudo-terminal, and this is the last close, a SIGHUP signal
 7289 shall be sent to the controlling process, if any, for which the slave side of the pseudo-terminal is
 7290 the controlling terminal. It is unspecified whether closing the master side of the pseudo-terminal
 7291 flushes all queued input and output.

7292 OB XSR If *fdes* refers to the slave side of a STREAMS-based pseudo-terminal, a zero-length message
 7293 may be sent to the master.

7294 When there is an outstanding cancelable asynchronous I/O operation against *fdes* when *close()*
 7295 is called, that I/O operation may be canceled. An I/O operation that is not canceled completes
 7296 as if the *close()* operation had not yet occurred. All operations that are not canceled shall
 7297 complete as if the *close()* blocked until the operations completed. The *close()* operation itself
 7298 need not block awaiting such I/O completion. Whether any I/O operation is canceled, and
 7299 which I/O operation may be canceled upon *close()*, is implementation-defined.

close()

7300 SHM If a memory mapped file or a shared memory object remains referenced at the last close (that is,
 7301 a process has it mapped), then the entire contents of the memory object shall persist until the
 7302 SHM memory object becomes unreferenced. If this is the last close of a memory mapped file or a
 7303 shared memory object and the close results in the memory object becoming unreferenced, and
 7304 the memory object has been unlinked, then the memory object shall be removed.

7305 If *fdes* refers to a socket, *close()* shall cause the socket to be destroyed. If the socket is in
 7306 connection-mode, and the `SO_LINGER` option is set for the socket with non-zero linger time,
 7307 and the socket has untransmitted data, then *close()* shall block for up to the current linger
 7308 interval until all data is transmitted.

RETURN VALUE

7309 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and *errno* set to
 7310 indicate the error.
 7311

ERRORS

7312 The *close()* function shall fail if:

7314 [EBADF] The *fdes* argument is not a valid file descriptor.

7315 [EINTR] The *close()* function was interrupted by a signal.

7316 The *close()* function may fail if:

7317 [EIO] An I/O error occurred while reading from or writing to the file system.

EXAMPLES**Reassigning a File Descriptor**

7319 The following example closes the file descriptor associated with standard output for the current
 7320 process, re-assigns standard output to a new file descriptor, and closes the original file descriptor
 7321 to clean up. This example assumes that the file descriptor 0 (which is the descriptor for standard
 7322 input) is not closed.
 7323

```
7324 #include <unistd.h>
7325 ...
7326 int pfd;
7327 ...
7328 close(1);
7329 dup(pfd);
7330 close(pfd);
7331 ...
```

7332 Incidentally, this is exactly what could be achieved using:

```
7333 dup2(pfd, 1);
7334 close(pfd);
```

Closing a File Descriptor

7336 In the following example, *close()* is used to close a file descriptor after an unsuccessful attempt is
 7337 made to associate that file descriptor with a stream.

```
7338 #include <stdio.h>
7339 #include <unistd.h>
7340 #include <stdlib.h>
7341 #define LOCKFILE "/etc/ptmp"
7342 ...
7343 int pfd;
```

```

7344     FILE *fpfd;
7345     ...
7346     if ((fpfd = fdopen (pfd, "w")) == NULL) {
7347         close(pfd);
7348         unlink(LOCKFILE);
7349         exit(1);
7350     }
7351     ...

```

APPLICATION USAGE

An application that had used the *stdio* routine *fopen()* to open a file should use the corresponding *fclose()* routine rather than *close()*. Once a file is closed, the file descriptor no longer exists, since the integer corresponding to it no longer refers to a file.

RATIONALE

The use of interruptible device close routines should be discouraged to avoid problems with the implicit closes of file descriptors by *exec* and *exit()*. This volume of IEEE Std 1003.1-200x only intends to permit such behavior by specifying the [EINTR] error condition.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.6 (on page 38), *exec*, *fattach()*, *fclose()*, *fdetach()*, *fopen()*, *ioctl()*, *open()*, *unlink()*, the Base Definitions volume of IEEE Std 1003.1-200x, <unistd.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

Issue 6

The DESCRIPTION related to a STREAMS-based file or pseudo-terminal is marked as part of the XSI STREAMS Option Group.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] error condition is added as an optional error.
- The DESCRIPTION is updated to describe the state of the *fildes* file descriptor as unspecified if an I/O error occurs and an [EIO] error condition is returned.

Text referring to sockets is added to the DESCRIPTION.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that shared memory objects and memory mapped files (and not typed memory objects) are the types of memory objects to which the paragraph on last closes applies.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/12 is applied, correcting the XSH shaded text relating to the master side of a pseudo-terminal. The reason for the change is that the behavior of pseudo-terminals and regular terminals should be as much alike as possible in this case; the change achieves that and matches historical behavior.

Issue 7

Functionality relating to the XSI STREAMS option is marked obsolescent.

Functionality relating to the Asynchronous Input and Output and Memory Mapped Files options is moved to the Base.

7389 **NAME**7390 `closedir` — close a directory stream7391 **SYNOPSIS**7392 `#include <dirent.h>`7393 `int closedir(DIR *dirp);`7394 **DESCRIPTION**7395 The `closedir()` function shall close the directory stream referred to by the argument `dirp`. Upon
7396 return, the value of `dirp` may no longer point to an accessible object of the type **DIR**. If a file
7397 descriptor is used to implement type **DIR**, that file descriptor shall be closed.7398 **RETURN VALUE**7399 Upon successful completion, `closedir()` shall return 0; otherwise, `-1` shall be returned and `errno`
7400 set to indicate the error.7401 **ERRORS**7402 The `closedir()` function may fail if:7403 [EBADF] The `dirp` argument does not refer to an open directory stream.7404 [EINTR] The `closedir()` function was interrupted by a signal.7405 **EXAMPLES**7406 **Closing a Directory Stream**7407 The following program fragment demonstrates how the `closedir()` function is used.7408

```
...
7409     DIR *dir;
7410     struct dirent *dp;
7411 ...
7412     if ((dir = opendir(".")) == NULL) {
7413     ...
7414     }
7415     while ((dp = readdir(dir)) != NULL) {
7416     ...
7417     }
7418     closedir(dir);
7419     ...
```

7420 **APPLICATION USAGE**

7421 None.

7422 **RATIONALE**

7423 None.

7424 **FUTURE DIRECTIONS**

7425 None.

7426 **SEE ALSO**7427 [*dirfd\(\)*](#), [*fdopendir\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, [**<dirent.h>**](#)

7428

CHANGE HISTORY

7429

First released in Issue 2.

7430

Issue 6

7431

In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

7432

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

7433

7434

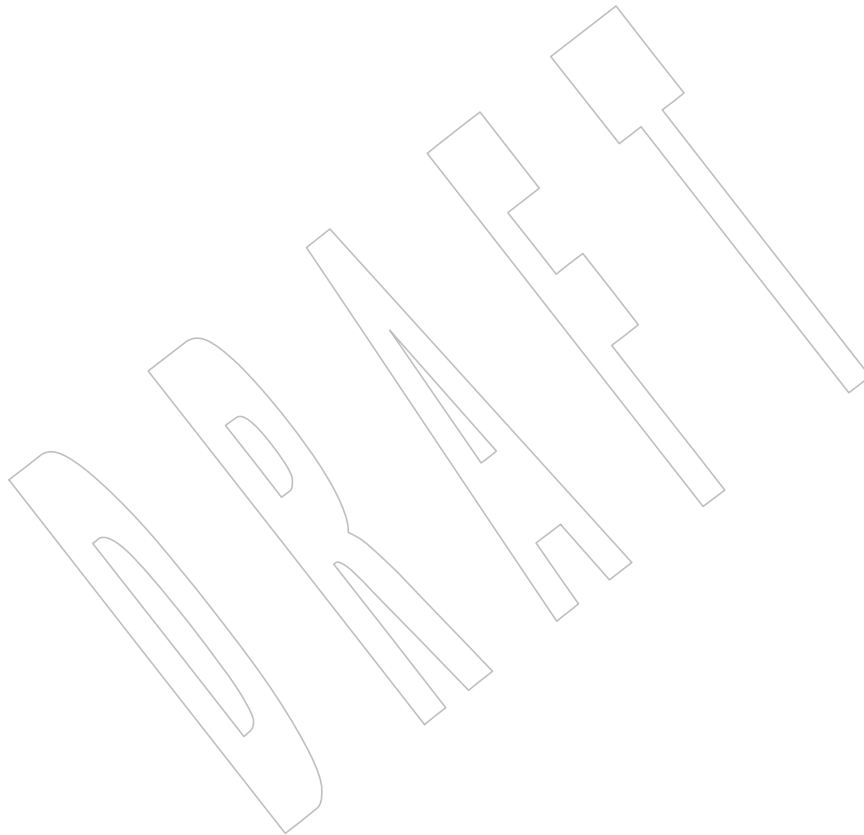
- The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

7435

7436

7437

- The [EINTR] error condition is added as an optional error condition.



7438 **NAME**

7439 closelog, openlog, setlogmask, syslog — control system log

7440 **SYNOPSIS**

```

7441 XSI #include <syslog.h>
7442
7442 void closelog(void);
7443 void openlog(const char *ident, int logopt, int facility);
7444 int setlogmask(int maskpri);
7445 void syslog(int priority, const char *message, ... /* arguments */);

```

7446 **DESCRIPTION**

7447 The *syslog()* function shall send a message to an implementation-defined logging facility, which
 7448 may log it in an implementation-defined system log, write it to the system console, forward it to
 7449 a list of users, or forward it to the logging facility on another host over the network. The logged
 7450 message shall include a message header and a message body. The message header contains at
 7451 least a timestamp and a tag string.

7452 The message body is generated from the *message* and following arguments in the same manner
 7453 as if these were arguments to *printf()*, except that the additional conversion specification *%m*
 7454 shall be recognized; it shall convert no arguments, shall cause the output of the error message
 7455 string associated with the value of *errno* on entry to *syslog()*, and may be mixed with argument
 7456 specifications of the "*%n\$*" form. If a complete conversion specification with the *m* conversion
 7457 specifier character is not just *%m*, the behavior is undefined. A trailing <newline> may be added
 7458 if needed.

7459 Values of the *priority* argument are formed by OR'ing together a severity-level value and an
 7460 optional facility value. If no facility value is specified, the current default facility value is used.

7461 Possible values of severity level include:

7462	LOG_EMERG	A panic condition.
7463	LOG_ALERT	A condition that should be corrected immediately, such as a corrupted system 7464 database.
7465	LOG_CRIT	Critical conditions, such as hard device errors.
7466	LOG_ERR	Errors.
7467	LOG_WARNING	
7468		Warning messages.
7469	LOG_NOTICE	Conditions that are not error conditions, but that may require special 7470 handling.
7471	LOG_INFO	Informational messages.
7472	LOG_DEBUG	Messages that contain information normally of use only when debugging a 7473 program.

7474 The facility indicates the application or system component generating the message. Possible
 7475 facility values include:

7476	LOG_USER	Messages generated by arbitrary processes. This is the default facility 7477 identifier if none is specified.
------	----------	--

7478	LOG_LOCAL0	Reserved for local use.
7479	LOG_LOCAL1	Reserved for local use.
7480	LOG_LOCAL2	Reserved for local use.
7481	LOG_LOCAL3	Reserved for local use.
7482	LOG_LOCAL4	Reserved for local use.
7483	LOG_LOCAL5	Reserved for local use.
7484	LOG_LOCAL6	Reserved for local use.
7485	LOG_LOCAL7	Reserved for local use.

7486 The *openlog()* function shall set process attributes that affect subsequent calls to *syslog()*. The
 7487 *ident* argument is a string that is prepended to every message. The *logopt* argument indicates
 7488 logging options. Values for *logopt* are constructed by a bitwise-inclusive OR of zero or more of
 7489 the following:

7490	LOG_PID	Log the process ID with each message. This is useful for identifying specific 7491 processes.
7492	LOG_CONS	Write messages to the system console if they cannot be sent to the logging 7493 facility. The <i>syslog()</i> function ensures that the process does not acquire the 7494 console as a controlling terminal in the process of writing the message.
7495	LOG_NDELAY	Open the connection to the logging facility immediately. Normally the open is 7496 delayed until the first message is logged. This is useful for programs that need 7497 to manage the order in which file descriptors are allocated.
7498	LOG_ODELAY	Delay open until <i>syslog()</i> is called.
7499	LOG_NOWAIT	Do not wait for child processes that may have been created during the course 7500 of logging the message. This option should be used by processes that enable 7501 notification of child termination using SIGCHLD, since <i>syslog()</i> may 7502 otherwise block waiting for a child whose exit status has already been 7503 collected.

7504 The *facility* argument encodes a default facility to be assigned to all messages that do not have an
 7505 explicit facility already encoded. The initial default facility is LOG_USER.

7506 The *openlog()* and *syslog()* functions may allocate a file descriptor. It is not necessary to call
 7507 *openlog()* prior to calling *syslog()*.

7508 The *closelog()* function shall close any open file descriptors allocated by previous calls to
 7509 *openlog()* or *syslog()*.

7510 The *setlogmask()* function shall set the log priority mask for the current process to *maskpri* and
 7511 return the previous mask. If the *maskpri* argument is 0, the current log mask is not modified.
 7512 Calls by the current process to *syslog()* with a priority not set in *maskpri* shall be rejected. The
 7513 default log mask allows all priorities to be logged. A call to *openlog()* is not required prior to
 7514 calling *setlogmask()*.

7515 Symbolic constants for use as values of the *logopt*, *facility*, *priority*, and *maskpri* arguments are
 7516 defined in the **<syslog.h>** header.

7517 RETURN VALUE

7518 The *setlogmask()* function shall return the previous log priority mask. The *closelog()*, *openlog()*,
 7519 and *syslog()* functions shall not return a value.

7520 ERRORS

7521 No errors are defined.

7522 EXAMPLES**7523 Using openlog()**

7524 The following example causes subsequent calls to *syslog()* to log the process ID with each
7525 message, and to write messages to the system console if they cannot be sent to the logging
7526 facility.

```
7527 #include <syslog.h>
7528 char *ident = "Process demo";
7529 int logopt = LOG_PID | LOG_CONS;
7530 int facility = LOG_USER;
7531 ...
7532 openlog(ident, logopt, facility);
```

7533 Using setlogmask()

7534 The following example causes subsequent calls to *syslog()* to accept error messages, and to reject
7535 all other messages.

```
7536 #include <syslog.h>
7537 int result;
7538 int mask = LOG_MASK (LOG_ERR);
7539 ...
7540 result = setlogmask(mask);
```

7541 Using syslog

7542 The following example sends the message "This is a message" to the default logging
7543 facility, marking the message as an error message generated by random processes.

```
7544 #include <syslog.h>
7545 char *message = "This is a message";
7546 int priority = LOG_ERR | LOG_USER;
7547 ...
7548 syslog(priority, message);
```

7549 APPLICATION USAGE

7550 None.

7551 RATIONALE

7552 None.

7553 FUTURE DIRECTIONS

7554 None.

7555 SEE ALSO

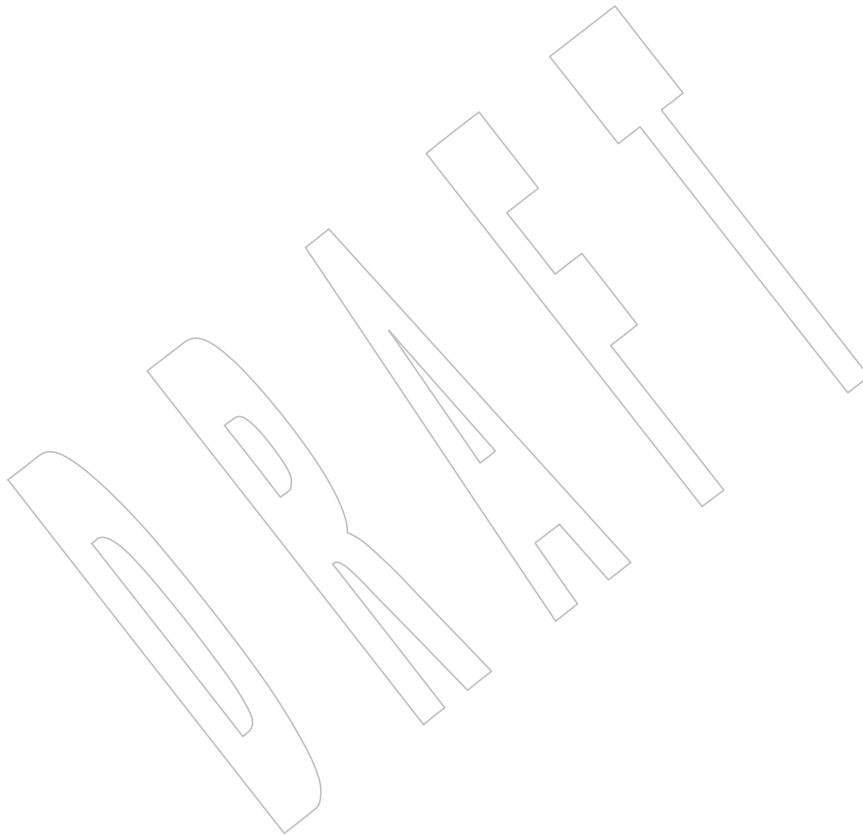
7556 *printf()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<syslog.h>**

7557 CHANGE HISTORY

7558 First released in Issue 4, Version 2.

7559 **Issue 5**
7560 Moved from X/OPEN UNIX extension to BASE.

7561 **Issue 6**
7562 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/13 is applied, correcting the EXAMPLES
7563 section.



7564 **NAME**7565 `confstr` — get configurable variables7566 **SYNOPSIS**7567 `#include <unistd.h>`7568 `size_t confstr(int name, char *buf, size_t len);`7569 **DESCRIPTION**7570 The `confstr()` function shall return configuration-defined string values. Its use and purpose are
7571 similar to `sysconf()`, but it is used where string values rather than numeric values are returned.7572 The `name` argument represents the system variable to be queried. The implementation shall
7573 support the following name values, defined in **<unistd.h>**. It may support others:7574 `_CS_PATH`
7575 `_CS_POSIX_V7_ILP32_OFF32_CFLAGS`
7576 `_CS_POSIX_V7_ILP32_OFF32_LDFLAGS`
7577 `_CS_POSIX_V7_ILP32_OFF32_LIBS`
7578 `_CS_POSIX_V7_ILP32_OFFBIG_CFLAGS`
7579 `_CS_POSIX_V7_ILP32_OFFBIG_LDFLAGS`
7580 `_CS_POSIX_V7_ILP32_OFFBIG_LIBS`
7581 `_CS_POSIX_V7_LP64_OFF64_CFLAGS`
7582 `_CS_POSIX_V7_LP64_OFF64_LDFLAGS`
7583 `_CS_POSIX_V7_LP64_OFF64_LIBS`
7584 `_CS_POSIX_V7_LPBIG_OFFBIG_CFLAGS`
7585 `_CS_POSIX_V7_LPBIG_OFFBIG_LDFLAGS`
7586 `_CS_POSIX_V7_LPBIG_OFFBIG_LIBS`
7587 `_CS_POSIX_V7_WIDTH_RESTRICTED_ENVS`
7588 `_CS_V7_ENV`

7589 OB

7589 `_CS_POSIX_V6_ILP32_OFF32_CFLAGS`
7590 `_CS_POSIX_V6_ILP32_OFF32_LDFLAGS`
7591 `_CS_POSIX_V6_ILP32_OFF32_LIBS`
7592 `_CS_POSIX_V6_ILP32_OFFBIG_CFLAGS`
7593 `_CS_POSIX_V6_ILP32_OFFBIG_LDFLAGS`
7594 `_CS_POSIX_V6_ILP32_OFFBIG_LIBS`
7595 `_CS_POSIX_V6_LP64_OFF64_CFLAGS`
7596 `_CS_POSIX_V6_LP64_OFF64_LDFLAGS`
7597 `_CS_POSIX_V6_LP64_OFF64_LIBS`
7598 `_CS_POSIX_V6_LPBIG_OFFBIG_CFLAGS`
7599 `_CS_POSIX_V6_LPBIG_OFFBIG_LDFLAGS`
7600 `_CS_POSIX_V6_LPBIG_OFFBIG_LIBS`
7601 `_CS_POSIX_V6_WIDTH_RESTRICTED_ENVS`
7602 `_CS_V6_ENV`7603 If `len` is not 0, and if `name` has a configuration-defined value, `confstr()` shall copy that value into
7604 the `len`-byte buffer pointed to by `buf`. If the string to be returned is longer than `len` bytes,
7605 including the terminating null, then `confstr()` shall truncate the string to `len-1` bytes and null-
7606 terminate the result. The application can detect that the string was truncated by comparing the
7607 value returned by `confstr()` with `len`.7608 If `len` is 0 and `buf` is a null pointer, then `confstr()` shall still return the integer value as defined
7609 below, but shall not return a string. If `len` is 0 but `buf` is not a null pointer, the result is
7610 unspecified.

7611 After a call to:

7612 `confstr(_CS_V7_ENV, buf, sizeof(buf))`

7613 the string stored in *buf* will contain the space-separated list of variable=value environment
7614 variable pairs required by the implementation to create a conforming environment, as described
7615 in the implementations' conformance documentation.

7616 If the implementation supports the POSIX shell option, the string stored in *buf* after a call to:

7617 `confstr(_CS_PATH, buf, sizeof(buf))`

7618 can be used as a value of the *PATH* environment variable that accesses all of the standard
7619 utilities of IEEE Std 1003.1-200x, if the return value is less than or equal to *sizeof(buf)*.

7620 RETURN VALUE

7621 If *name* has a configuration-defined value, *confstr()* shall return the size of buffer that would be
7622 needed to hold the entire configuration-defined value including the terminating null. If this
7623 return value is greater than *len*, the string returned in *buf* is truncated.

7624 If *name* is invalid, *confstr()* shall return 0 and set *errno* to indicate the error.

7625 If *name* does not have a configuration-defined value, *confstr()* shall return 0 and leave *errno*
7626 unchanged.

7627 ERRORS

7628 The *confstr()* function shall fail if:

7629 [EINVAL] The value of the *name* argument is invalid.

7630 EXAMPLES

7631 None.

7632 APPLICATION USAGE

7633 An application can distinguish between an invalid *name* parameter value and one that
7634 corresponds to a configurable variable that has no configuration-defined value by checking if
7635 *errno* is modified. This mirrors the behavior of *sysconf()*.

7636 The original need for this function was to provide a way of finding the configuration-defined
7637 default value for the environment variable *PATH*. Since *PATH* can be modified by the user to
7638 include directories that could contain utilities replacing the standard utilities in the Shell and
7639 Utilities volume of IEEE Std 1003.1-200x, applications need a way to determine the system-
7640 supplied *PATH* environment variable value that contains the correct search path for the standard
7641 utilities.

7642 An application could use:

7643 `confstr(name, (char *)NULL, (size_t)0)`

7644 to find out how big a buffer is needed for the string value; use *malloc()* to allocate a buffer to
7645 hold the string; and call *confstr()* again to get the string. Alternately, it could allocate a fixed,
7646 static buffer that is big enough to hold most answers (perhaps 512 or 1024 bytes), but then use
7647 *malloc()* to allocate a larger buffer if it finds that this is too small.

7648 RATIONALE

7649 Application developers can normally determine any configuration variable by means of reading
7650 from the stream opened by a call to:

7651 `popen("command -p getconf variable", "r");`

7652 The *confstr()* function with a *name* argument of *_CS_PATH* returns a string that can be used as a
7653 *PATH* environment variable setting that will reference the standard shell and utilities as
7654 described in the Shell and Utilities volume of IEEE Std 1003.1-200x.

7655 The *confstr()* function copies the returned string into a buffer supplied by the application instead
 7656 of returning a pointer to a string. This allows a cleaner function in some implementations (such
 7657 as those with lightweight threads) and resolves questions about when the application must copy
 7658 the string returned.

FUTURE DIRECTIONS

7659 None.
 7660

SEE ALSO

7661 *exec*, *pathconf()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**, the
 7662 Shell and Utilities volume of IEEE Std 1003.1-200x, *c99*
 7663

CHANGE HISTORY

7664 First released in Issue 4. Derived from the ISO POSIX-2 standard.
 7665

Issue 5

7666 A table indicating the permissible values of *name* is added to the DESCRIPTION. All those
 7667 marked EX are new in this issue.
 7668

Issue 6

7669 The Open Group Corrigendum U033/7 is applied. The return value for the case returning the
 7670 size of the buffer now explicitly states that this includes the terminating null.
 7671

7672 The following new requirements on POSIX implementations derive from alignment with the
 7673 Single UNIX Specification:

- 7674 • The DESCRIPTION is updated with new arguments which can be used to determine
 7675 configuration strings for C compiler flags, linker/loader flags, and libraries for each
 7676 different supported programming environment. This is a change to support data size
 7677 neutrality.

7678 The following changes were made to align with the IEEE P1003.1a draft standard:

- 7679 • The DESCRIPTION is updated to include text describing how `_CS_PATH` can be used to
 7680 obtain a *PATH* to access the standard utilities.

7681 The macros associated with the *c89* programming models are marked LEGACY and new
 7682 equivalent macros associated with *c99* are introduced.

Issue 7

7683 Austin Group Interpretation 1003.1-2001 #047 is applied, adding the `_CS_V7_ENV` variable.
 7684

7685 The V6 variables for the supported programming environments are marked obsolescent.

7686 The variables for the supported programming environments are updated to be V7.

7687 The LEGACY variables and obsolescent values are removed.

7688 **NAME**
 7689 conj, conjf, conjl — complex conjugate functions

7690 **SYNOPSIS**
 7691 #include <complex.h>
 7692 double complex conj(double complex z);
 7693 float complex conjf(float complex z);
 7694 long double complex conjl(long double complex z);

7695 **DESCRIPTION**
 7696 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 7697 conflict between the requirements described here and the ISO C standard is unintentional. This
 7698 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

7699 These functions shall compute the complex conjugate of *z*, by reversing the sign of its imaginary
 7700 part.

7701 **RETURN VALUE**
 7702 These functions return the complex conjugate value.

7703 **ERRORS**
 7704 No errors are defined.

7705 **EXAMPLES**
 7706 None.

7707 **APPLICATION USAGE**
 7708 None.

7709 **RATIONALE**
 7710 None.

7711 **FUTURE DIRECTIONS**
 7712 None.

7713 **SEE ALSO**
 7714 *carg()*, *cimag()*, *cproj()*, *creal()*, the Base Definitions volume of IEEE Std 1003.1-200x,
 7715 <complex.h>

7716 **CHANGE HISTORY**
 7717 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7718 **NAME**

7719 connect — connect a socket

7720 **SYNOPSIS**

7721 #include <sys/socket.h>

7722 int connect(int *socket*, const struct sockaddr **address*,
7723 socklen_t *address_len*);7724 **DESCRIPTION**7725 The *connect()* function shall attempt to make a connection on a connection-mode socket or to set
7726 or reset the peer address of a connectionless-mode socket. The function takes the following
7727 arguments:

7728	<i>socket</i>	Specifies the file descriptor associated with the socket.
7729	<i>address</i>	Points to a sockaddr structure containing the peer address. The length and 7730 format of the address depend on the address family of the socket.
7731	<i>address_len</i>	Specifies the length of the sockaddr structure pointed to by the <i>address</i> 7732 argument.

7733 If the socket has not already been bound to a local address, *connect()* shall bind it to an address
7734 which, unless the socket's address family is AF_UNIX, is an unused local address.7735 If the initiating socket is not connection-mode, then *connect()* shall set the socket's peer address,
7736 and no connection is made. For SOCK_DGRAM sockets, the peer address identifies where all
7737 datagrams are sent on subsequent *send()* functions, and limits the remote sender for subsequent
7738 *recv()* functions. If *address* is a null address for the protocol, the socket's peer address shall be
7739 reset. Note that despite no connection being made, the term "connected" is used to describe a
7740 connectionless-mode socket for which a peer address has been set.7741 If the initiating socket is connection-mode, then *connect()* shall attempt to establish a connection
7742 to the address specified by the *address* argument. If the connection cannot be established
7743 immediately and O_NONBLOCK is not set for the file descriptor for the socket, *connect()* shall
7744 block for up to an unspecified timeout interval until the connection is established. If the timeout
7745 interval expires before the connection is established, *connect()* shall fail and the connection
7746 attempt shall be aborted. If *connect()* is interrupted by a signal that is caught while blocked
7747 waiting to establish a connection, *connect()* shall fail and set *errno* to [EINTR], but the connection
7748 request shall not be aborted, and the connection shall be established asynchronously.7749 If the connection cannot be established immediately and O_NONBLOCK is set for the file
7750 descriptor for the socket, *connect()* shall fail and set *errno* to [EINPROGRESS], but the connection
7751 request shall not be aborted, and the connection shall be established asynchronously. Subsequent
7752 calls to *connect()* for the same socket, before the connection is established, shall fail and set *errno*
7753 to [EALREADY].7754 When the connection has been established asynchronously, *select()* and *poll()* shall indicate that
7755 the file descriptor for the socket is ready for writing.7756 The socket in use may require the process to have appropriate privileges to use the *connect()*
7757 function.7758 **RETURN VALUE**7759 Upon successful completion, *connect()* shall return 0; otherwise, -1 shall be returned and *errno*
7760 set to indicate the error.

ERRORS

- 7761 The *connect()* function shall fail if:
- 7762 [EADDRNOTAVAIL] The specified address is not available from the local machine.
- 7763 [EAFNOSUPPORT] The specified address is not a valid address for the address family of the
- 7764 specified socket.
- 7765 [EALREADY] A connection request is already in progress for the specified socket.
- 7766 [EBADF] The *socket* argument is not a valid file descriptor.
- 7767 [ECONNREFUSED] The target address was not listening for connections or refused the connection
- 7768 request.
- 7769 [EINPROGRESS] O_NONBLOCK is set for the file descriptor for the socket and the connection
- 7770 cannot be immediately established; the connection shall be established
- 7771 asynchronously.
- 7772 [EINTR] The attempt to establish a connection was interrupted by delivery of a signal
- 7773 that was caught; the connection shall be established asynchronously.
- 7774 [EISCONN] The specified socket is connection-mode and is already connected.
- 7775 [ENETUNREACH] No route to the network is present.
- 7776 [ENOTSOCK] The *socket* argument does not refer to a socket.
- 7777 [EPROTOTYPE] The specified address has a different type than the socket bound to the
- 7778 specified peer address.
- 7779 [ETIMEDOUT] The attempt to connect timed out before a connection was made.
- 7780 If the address family of the socket is AF_UNIX, then *connect()* shall fail if:
- 7781 [EIO] An I/O error occurred while reading from or writing to the file system.
- 7782 [ELOOP] A loop exists in symbolic links encountered during resolution of the pathname
- 7783 in *address*.
- 7784 [ENAMETOOLONG] A component of a pathname exceeded {NAME_MAX} characters, or an entire
- 7785 pathname exceeded {PATH_MAX} characters.
- 7786 [ENOENT] A component of the pathname does not name an existing file or the pathname
- 7787 is an empty string.
- 7788 [ENOTDIR] A component of the path prefix of the pathname in *address* is not a directory.
- 7789 The *connect()* function may fail if:
- 7790 [EACCES] Search permission is denied for a component of the path prefix; or write access
- 7791 to the named socket is denied.
- 7792 [EADDRINUSE] Attempt to establish a connection that uses addresses that are already in use.
- 7793 [ECONNRESET] Remote host reset the connection request.
- 7794 [EHOSTUNREACH] The destination host cannot be reached (probably because the host is down or
- 7795 a remote router cannot reach it).

connect()

- 7803 [EINVAL] The *address_len* argument is not a valid length for the address family; or
7804 invalid address family in the **sockaddr** structure.
- 7805 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
7806 resolution of the pathname in *address*.
- 7807 [ENAMETOOLONG]
7808 Pathname resolution of a symbolic link produced an intermediate result
7809 whose length exceeds {PATH_MAX}.
- 7810 [ENETDOWN] The local network interface used to reach the destination is down.
- 7811 [ENOBUFS] No buffer space is available.
- 7812 [EOPNOTSUPP] The socket is listening and cannot be connected.

EXAMPLES

7813 None.
7814

APPLICATION USAGE

7815 If *connect()* fails, the state of the socket is unspecified. Conforming applications should close the
7816 file descriptor and create a new socket before attempting to reconnect.
7817

RATIONALE

7818 None.
7819

FUTURE DIRECTIONS

7820 None.
7821

SEE ALSO

7822 *accept()*, *bind()*, *close()*, *getsockname()*, *poll()*, *select()*, *send()*, *shutdown()*, *socket()*, the Base
7823 Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**
7824

CHANGE HISTORY

7825 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
7826

7827 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
7828 [ELOOP] error condition is added.

Issue 7

7829 Austin Group Interpretation 1003.1-2001 #035 is applied, clarifying the description of connected
7830 sockets.
7831

7832 **NAME**
 7833 copysign, copysignf, copysignl — number manipulation function

7834 **SYNOPSIS**
 7835 #include <math.h>
 7836 double copysign(double x, double y);
 7837 float copysignf(float x, float y);
 7838 long double copysignl(long double x, long double y);

7839 **DESCRIPTION**
 7840 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 7841 conflict between the requirements described here and the ISO C standard is unintentional. This
 7842 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

7843 These functions shall produce a value with the magnitude of x and the sign of y . On
 7844 implementations that represent a signed zero but do not treat negative zero consistently in
 7845 arithmetic operations, these functions regard the sign of zero as positive.

7846 **RETURN VALUE**
 7847 Upon successful completion, these functions shall return a value with the magnitude of x and
 7848 the sign of y .

7849 **ERRORS**
 7850 No errors are defined.

7851 **EXAMPLES**
 7852 None.

7853 **APPLICATION USAGE**
 7854 None.

7855 **RATIONALE**
 7856 None.

7857 **FUTURE DIRECTIONS**
 7858 None.

7859 **SEE ALSO**
 7860 [signbit\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, <math.h>

7861 **CHANGE HISTORY**
 7862 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7863 **NAME**

7864 cos, cosf, cosl — cosine function

7865 **SYNOPSIS**

```
7866 #include <math.h>
7867
7867 double cos(double x);
7868 float cosf(float x);
7869 long double cosl(long double x);
```

7870 **DESCRIPTION**

7871 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 7872 conflict between the requirements described here and the ISO C standard is unintentional. This
 7873 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

7874 These functions shall compute the cosine of their argument x , measured in radians.

7875 An application wishing to check for error situations should set *errno* to zero and call
 7876 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 7877 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 7878 zero, an error has occurred.

7879 **RETURN VALUE**

7880 Upon successful completion, these functions shall return the cosine of x .

7881 MX If x is NaN, a NaN shall be returned.

7882 If x is ± 0 , the value 1.0 shall be returned.

7883 If x is $\pm\text{Inf}$, a domain error shall occur, and either a NaN (if supported), or an implementation-
 7884 defined value shall be returned.

7885 **ERRORS**

7886 These functions shall fail if:

7887 MX **Domain Error** The x argument is $\pm\text{Inf}$.

7888 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 7889 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 7890 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 7891 shall be raised.

7892 **EXAMPLES**7893 **Taking the Cosine of a 45-Degree Angle**

```
7894 #include <math.h>
7895 ...
7896 double radians = 45 * M_PI / 180;
7897 double result;
7898 ...
7899 result = cos(radians);
```

7900 **APPLICATION USAGE**

7901 These functions may lose accuracy when their argument is near an odd multiple of $\pi/2$ or is far
 7902 from 0.

7903 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
 7904 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

7905
7906
7907
7908
7909
7910
7911
7912
7913
7914
7915
7916
7917
7918
7919
7920
7921
7922
7923

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

acos(), *feclearexcept()*, *fetetestexcept()*, *isnan()*, *sin()*, *tan()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

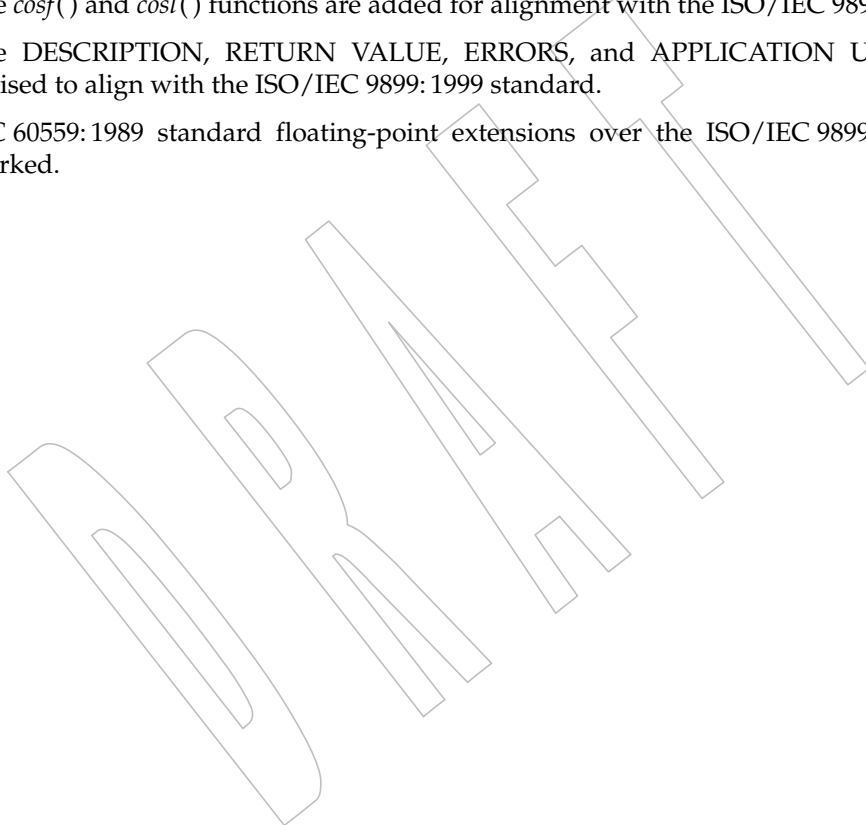
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *cosf()* and *cosl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.



7924 **NAME**
 7925 cosh, coshf, coshl — hyperbolic cosine functions

7926 **SYNOPSIS**
 7927 #include <math.h>
 7928 double cosh(double x);
 7929 float coshf(float x);
 7930 long double coshl(long double x);

7931 **DESCRIPTION**
 7932 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 7933 conflict between the requirements described here and the ISO C standard is unintentional. This
 7934 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

7935 These functions shall compute the hyperbolic cosine of their argument x .

7936 An application wishing to check for error situations should set *errno* to zero and call
 7937 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 7938 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 7939 zero, an error has occurred.

7940 **RETURN VALUE**
 7941 Upon successful completion, these functions shall return the hyperbolic cosine of x .
 7942 If the correct value would cause overflow, a range error shall occur and *cosh()*, *coshf()*, and
 7943 *coshl()* shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL,
 7944 respectively.

7945 MX If x is NaN, a NaN shall be returned.
 7946 If x is ± 0 , the value 1.0 shall be returned.
 7947 If x is $\pm\text{Inf}$, $+\text{Inf}$ shall be returned.

7948 **ERRORS**
 7949 These functions shall fail if:
 7950 Range Error The result would cause an overflow.
 7951 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 7952 then *errno* shall be set to [ERANGE]. If the integer expression
 7953 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
 7954 floating-point exception shall be raised.

7955 **EXAMPLES**
 7956 None.

7957 **APPLICATION USAGE**
 7958 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
 7959 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.
 7960 For IEEE Std 754-1985 **double**, $710.5 < |x|$ implies that *cosh*(x) has overflowed.

7961 **RATIONALE**
 7962 None.

7963
7964
7965
7966
7967
7968
7969
7970
7971
7972
7973
7974
7975
7976
7977
7978
7979

FUTURE DIRECTIONS

None.

SEE ALSO

acosh(), *feclearexcept()*, *fetestexcept()*, *isnan()*, *sinh()*, *tanh()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

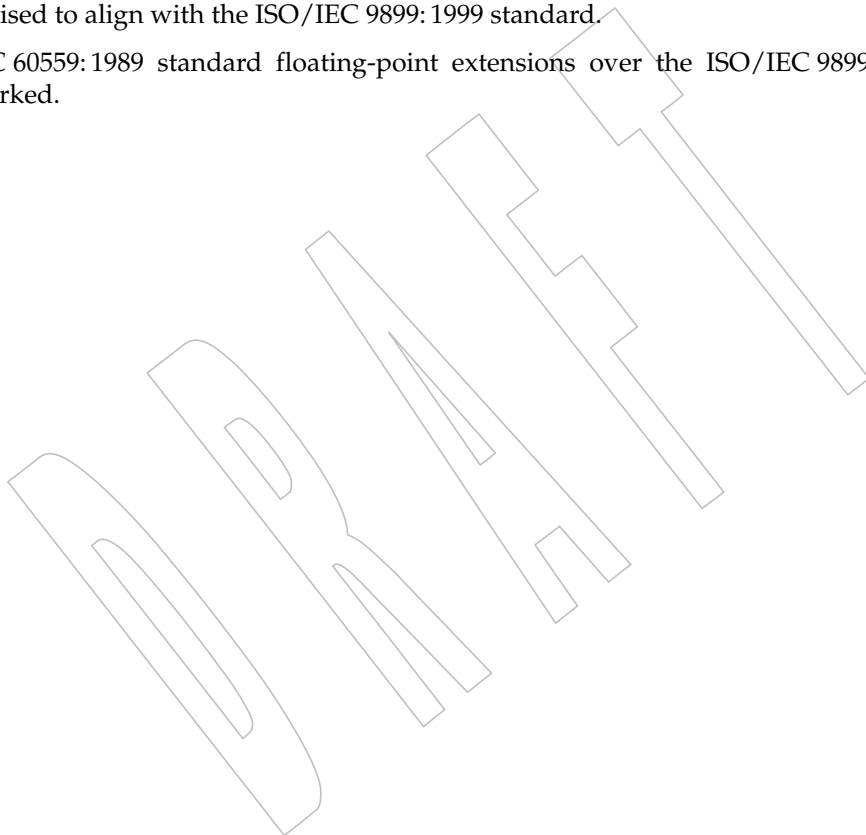
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *coshf()* and *coshl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

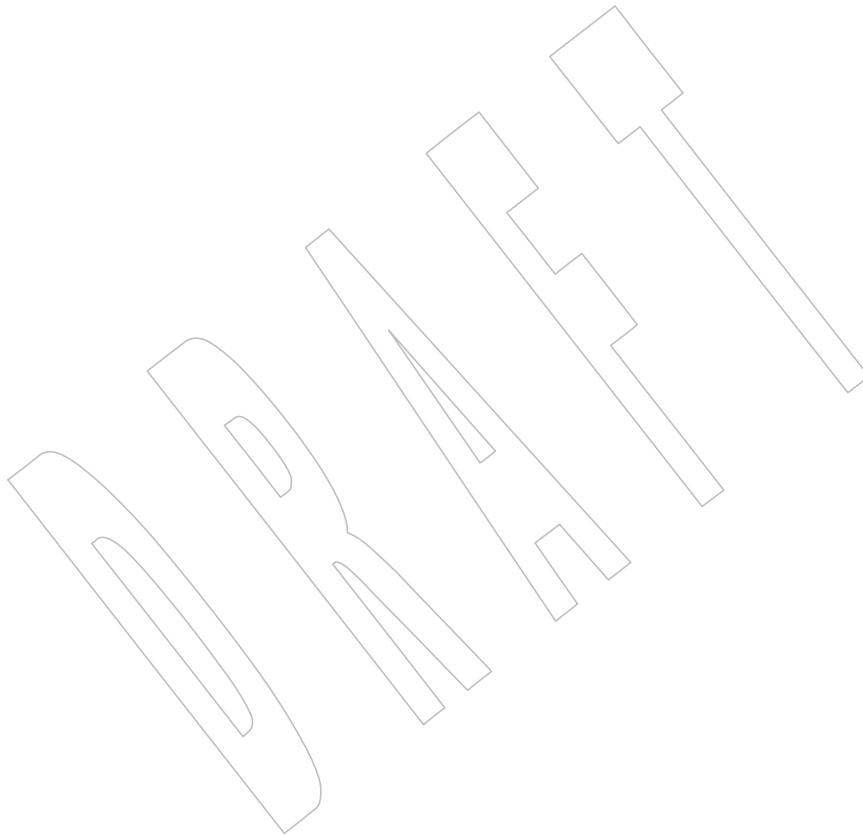
IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.



7980 **NAME**
7981 `cosl` — cosine function

7982 **SYNOPSIS**
7983 `#include <math.h>`
7984 `long double cosl(long double x);`

7985 **DESCRIPTION**
7986 Refer to `cos()`.



7987 **NAME**
 7988 cpow, cpowf, cpowl — complex power functions

7989 **SYNOPSIS**
 7990 #include <complex.h>
 7991 double complex cpow(double complex x, double complex y);
 7992 float complex cpowf(float complex x, float complex y);
 7993 long double complex cpowl(long double complex x,
 7994 long double complex y);

7995 **DESCRIPTION**
 7996 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 7997 conflict between the requirements described here and the ISO C standard is unintentional. This
 7998 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

7999 These functions shall compute the complex power function x^y , with a branch cut for the first
 8000 parameter along the negative real axis.

8001 **RETURN VALUE**
 8002 These functions shall return the complex power function value.

8003 **ERRORS**
 8004 No errors are defined.

8005 **EXAMPLES**
 8006 None.

8007 **APPLICATION USAGE**
 8008 None.

8009 **RATIONALE**
 8010 None.

8011 **FUTURE DIRECTIONS**
 8012 None.

8013 **SEE ALSO**
 8014 *cabs()*, *csqrt()*, the Base Definitions volume of IEEE Std 1003.1-200x, <complex.h>

8015 **CHANGE HISTORY**
 8016 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

8017 **NAME**
 8018 `cproj`, `cprojf`, `cprojl` — complex projection functions

8019 **SYNOPSIS**
 8020 `#include <complex.h>`
 8021 `double complex cproj(double complex z);`
 8022 `float complex cprojf(float complex z);`
 8023 `long double complex cprojl(long double complex z);`

8024 **DESCRIPTION**
 8025 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 8026 conflict between the requirements described here and the ISO C standard is unintentional. This
 8027 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

8028 These functions shall compute a projection of z onto the Riemann sphere: z projects to z , except
 8029 that all complex infinities (even those with one infinite part and one NaN part) project to
 8030 positive infinity on the real axis. If z has an infinite part, then $cproj(z)$ shall be equivalent to:

8031 `INFINITY + I * copysign(0.0, cimag(z))`

8032 **RETURN VALUE**
 8033 These functions shall return the value of the projection onto the Riemann sphere.

8034 **ERRORS**
 8035 No errors are defined.

8036 **EXAMPLES**
 8037 None.

8038 **APPLICATION USAGE**
 8039 None.

8040 **RATIONALE**
 8041 Two topologies are commonly used in complex mathematics: the complex plane with its
 8042 continuum of infinities, and the Riemann sphere with its single infinity. The complex plane is
 8043 better suited for transcendental functions, the Riemann sphere for algebraic functions. The
 8044 complex types with their multiplicity of infinities provide a useful (though imperfect) model for
 8045 the complex plane. The `cproj()` function helps model the Riemann sphere by mapping all
 8046 infinities to one, and should be used just before any operation, especially comparisons, that
 8047 might give spurious results for any of the other infinities. Note that a complex value with one
 8048 infinite part and one NaN part is regarded as an infinity, not a NaN, because if one part is
 8049 infinite, the complex value is infinite independent of the value of the other part. For the same
 8050 reason, `cabs()` returns an infinity if its argument has an infinite part and a NaN part.

8051 **FUTURE DIRECTIONS**
 8052 None.

8053 **SEE ALSO**
 8054 `carg()`, `cimag()`, `conj()`, `creal()`, the Base Definitions volume of IEEE Std 1003.1-200x,
 8055 `<complex.h>`

8056 **CHANGE HISTORY**
 8057 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

8058 **NAME**
 8059 creal, crealf, creall — complex real functions

8060 **SYNOPSIS**
 8061 #include <complex.h>
 8062 double creal(double complex z);
 8063 float crealf(float complex z);
 8064 long double creall(long double complex z);

8065 **DESCRIPTION**
 8066 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 8067 conflict between the requirements described here and the ISO C standard is unintentional. This
 8068 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

8069 These functions shall compute the real part of *z*.

8070 **RETURN VALUE**
 8071 These functions shall return the real part value.

8072 **ERRORS**
 8073 No errors are defined.

8074 **EXAMPLES**
 8075 None.

8076 **APPLICATION USAGE**
 8077 For a variable *z* of type **complex**:
 8078 `z == creal(z) + cimag(z)*I`

8079 **RATIONALE**
 8080 None.

8081 **FUTURE DIRECTIONS**
 8082 None.

8083 **SEE ALSO**
 8084 *carg()*, *cimag()*, *conj()*, *cproj()*, the Base Definitions volume of IEEE Std 1003.1-200x,
 8085 <complex.h>

8086 **CHANGE HISTORY**
 8087 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

8088 **NAME**
 8089 `creat` — create a new file or rewrite an existing one

8090 **SYNOPSIS**

8091 OH `#include <sys/stat.h>`
 8092 `#include <fcntl.h>`
 8093 `int creat(const char *path, mode_t mode);`

8094 **DESCRIPTION**

8095 The function call:
 8096 `creat(path, mode)`
 8097 shall be equivalent to:
 8098 `open(path, O_WRONLY|O_CREAT|O_TRUNC, mode)`

8099 **RETURN VALUE**

8100 Refer to *open()*.

8101 **ERRORS**

8102 Refer to *open()*.

8103 **EXAMPLES**

8104 **Creating a File**

8105 The following example creates the file `/tmp/file` with read and write permissions for the file
 8106 owner and read permission for group and others. The resulting file descriptor is assigned to the
 8107 *fd* variable.

```
8108 #include <fcntl.h>
8109 ...
8110 int fd;
8111 mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
8112 char *filename = "/tmp/file";
8113 ...
8114 fd = creat(filename, mode);
8115 ...
```

8116 **APPLICATION USAGE**

8117 None.

8118 **RATIONALE**

8119 The *creat()* function is redundant. Its services are also provided by the *open()* function. It has
 8120 been included primarily for historical purposes since many existing applications depend on it. It
 8121 is best considered a part of the C binding rather than a function that should be provided in other
 8122 languages.

8123 **FUTURE DIRECTIONS**

8124 None.

8125 **SEE ALSO**

8126 *mknod()*, *open()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<fcntl.h>`, `<sys/stat.h>`,
 8127 `<sys/types.h>`

8128
8129
8130
8131
8132
8133
8134
8135
8136

CHANGE HISTORY

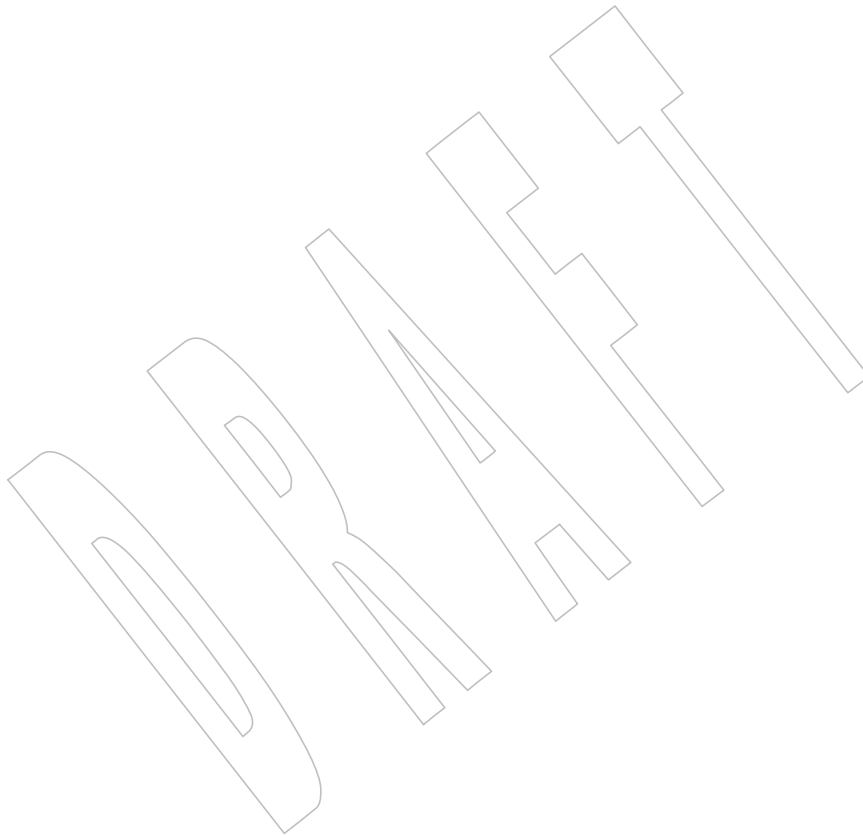
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.



8137 **NAME**8138 `crypt` — string encoding function (**CRYPT**)8139 **SYNOPSIS**

```
8140 __XSI_VISIBLE
8140 #include <unistd.h>
8141 char *crypt(const char *key, const char *salt);
```

8142 **DESCRIPTION**8143 The `crypt()` function is a string encoding function. The algorithm is implementation-defined.8144 The `key` argument points to a string to be encoded. The `salt` argument is a string chosen from the set:

```
8146 a b c d e f g h i j k l m n o p q r s t u v w x y z
8147 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
8148 0 1 2 3 4 5 6 7 8 9 . /
```

8149 The first two characters of this string may be used to perturb the encoding algorithm.

8150 The return value of `crypt()` points to static data that is overwritten by each call.8151 The `crypt()` function need not be thread-safe. A function that is not required to be thread-safe is not required to be reentrant.8153 **RETURN VALUE**8154 Upon successful completion, `crypt()` shall return a pointer to the encoded string. The first two characters of the returned value shall be those of the `salt` argument. Otherwise, it shall return a null pointer and set `errno` to indicate the error.8157 **ERRORS**8158 The `crypt()` function shall fail if:8159 `[ENOSYS]` The functionality is not supported on this implementation.8160 **EXAMPLES**8161 **Encoding Passwords**8162 The following example finds a user database entry matching a particular user name and changes the current password to a new password. The `crypt()` function generates an encoded version of each password. The first call to `crypt()` produces an encoded version of the old password; that encoded password is then compared to the password stored in the user database. The second call to `crypt()` encodes the new password before it is stored.8167 The `putpwent()` function, used in the following example, is not part of IEEE Std 1003.1-200x.

```
8168 #include <unistd.h>
8169 #include <pwd.h>
8170 #include <string.h>
8171 #include <stdio.h>
8172 ...
8173 int valid_change;
8174 int pfd; /* Integer for file descriptor returned by open(). */
8175 FILE *fpfd; /* File pointer for use in putpwent(). */
8176 struct passwd *p;
8177 char user[100];
8178 char oldpasswd[100];
```

```

8179     char newpasswd[100];
8180     char savepasswd[100];
8181     ...
8182     valid_change = 0;
8183     while ((p = getpwent()) != NULL) {
8184         /* Change entry if found. */
8185         if (strcmp(p->pw_name, user) == 0) {
8186             if (strcmp(p->pw_passwd, crypt(oldpasswd, p->pw_passwd)) == 0) {
8187                 strcpy(savepasswd, crypt(newpasswd, user));
8188                 p->pw_passwd = savepasswd;
8189                 valid_change = 1;
8190             }
8191             else {
8192                 fprintf(stderr, "Old password is not valid\n");
8193             }
8194         }
8195         /* Put passwd entry into ptmp. */
8196         putpwent(p, fpfd);
8197     }

```

APPLICATION USAGE

The values returned by this function need not be portable among XSI-conformant systems.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[encrypt\(\)](#), [setkey\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, <[unistd.h](#)>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

NAME

csin, csinf, csinl — complex sine functions

SYNOPSIS

```
#include <complex.h>

double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex sine of z .

RETURN VALUE

These functions shall return the complex sine value.

ERRORS

No errors are defined.

EXAMPLES

None.

8239 **NAME**
 8240 csinh, csinhf, csinhl — complex hyperbolic sine functions

8241 **SYNOPSIS**
 8242 #include <complex.h>
 8243 double complex csinh(double complex z);
 8244 float complex csinhf(float complex z);
 8245 long double complex csinhl(long double complex z);

8246 **DESCRIPTION**
 8247 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 8248 conflict between the requirements described here and the ISO C standard is unintentional. This
 8249 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

8250 These functions shall compute the complex hyperbolic sine of z .

8251 **RETURN VALUE**
 8252 These functions shall return the complex hyperbolic sine value.

8253 **ERRORS**
 8254 No errors are defined.

8255 **EXAMPLES**
 8256 None.

8257 **APPLICATION USAGE**
 8258 None.

8259 **RATIONALE**
 8260 None.

8261 **FUTURE DIRECTIONS**
 8262 None.

8263 **SEE ALSO**
 8264 *casinh()*, the Base Definitions volume of IEEE Std 1003.1-200x, <complex.h>

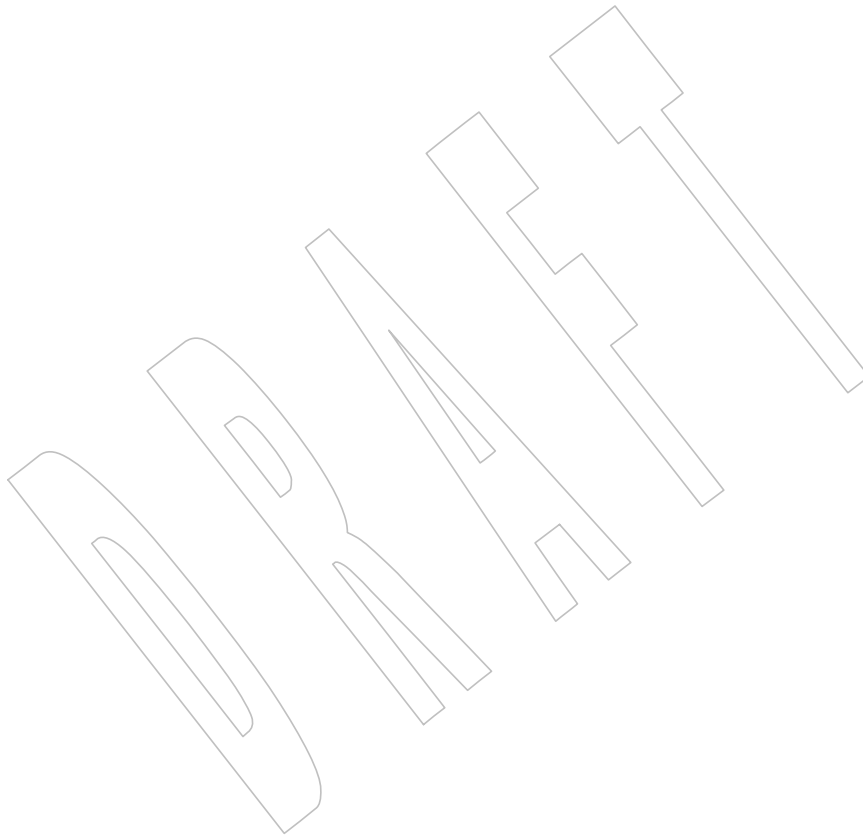
8265 **CHANGE HISTORY**
 8266 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

8267 **NAME**
8268 `csinl` — complex sine functions

8269 **SYNOPSIS**
8270 `#include <complex.h>`

8271 `long double complex csinl(long double complex z);`

8272 **DESCRIPTION**
8273 Refer to *csin()*.



8274 **NAME**
 8275 csqrt, csqrtf, csqrtl — complex square root functions

8276 **SYNOPSIS**
 8277 #include <complex.h>
 8278 double complex csqrt(double complex z);
 8279 float complex csqrtf(float complex z);
 8280 long double complex csqrtl(long double complex z);

8281 **DESCRIPTION**
 8282 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 8283 conflict between the requirements described here and the ISO C standard is unintentional. This
 8284 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

8285 These functions shall compute the complex square root of z , with a branch cut along the negative
 8286 real axis.

8287 **RETURN VALUE**
 8288 These functions shall return the complex square root value, in the range of the right half-plane
 8289 (including the imaginary axis).

8290 **ERRORS**
 8291 No errors are defined.

8292 **EXAMPLES**
 8293 None.

8294 **APPLICATION USAGE**
 8295 None.

8296 **RATIONALE**
 8297 None.

8298 **FUTURE DIRECTIONS**
 8299 None.

8300 **SEE ALSO**
 8301 *cabs()*, *cpow()*, the Base Definitions volume of IEEE Std 1003.1-200x, <complex.h>

8302 **CHANGE HISTORY**
 8303 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

8304 **NAME**
 8305 `ctan, ctanf, ctanl` — complex tangent functions

8306 **SYNOPSIS**
 8307 `#include <complex.h>`
 8308 `double complex ctan(double complex z);`
 8309 `float complex ctanf(float complex z);`
 8310 `long double complex ctanl(long double complex z);`

8311 **DESCRIPTION**
 8312 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 8313 conflict between the requirements described here and the ISO C standard is unintentional. This
 8314 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

8315 These functions shall compute the complex tangent of z .

8316 **RETURN VALUE**
 8317 These functions shall return the complex tangent value.

8318 **ERRORS**
 8319 No errors are defined.

8320 **EXAMPLES**
 8321 None.

8322 **APPLICATION USAGE**
 8323 None.

8324 **RATIONALE**
 8325 None.

8326 **FUTURE DIRECTIONS**
 8327 None.

8328 **SEE ALSO**
 8329 [*catan\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

8330 **CHANGE HISTORY**
 8331 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

8332 **NAME**
 8333 ctanh, ctanhf, ctanhl — complex hyperbolic tangent functions

8334 **SYNOPSIS**
 8335 #include <complex.h>
 8336 double complex ctanh(double complex z);
 8337 float complex ctanhf(float complex z);
 8338 long double complex ctanhl(long double complex z);

8339 **DESCRIPTION**
 8340 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 8341 conflict between the requirements described here and the ISO C standard is unintentional. This
 8342 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

8343 These functions shall compute the complex hyperbolic tangent of z .

8344 **RETURN VALUE**
 8345 These functions shall return the complex hyperbolic tangent value.

8346 **ERRORS**
 8347 No errors are defined.

8348 **EXAMPLES**
 8349 None.

8350 **APPLICATION USAGE**
 8351 None.

8352 **RATIONALE**
 8353 None.

8354 **FUTURE DIRECTIONS**
 8355 None.

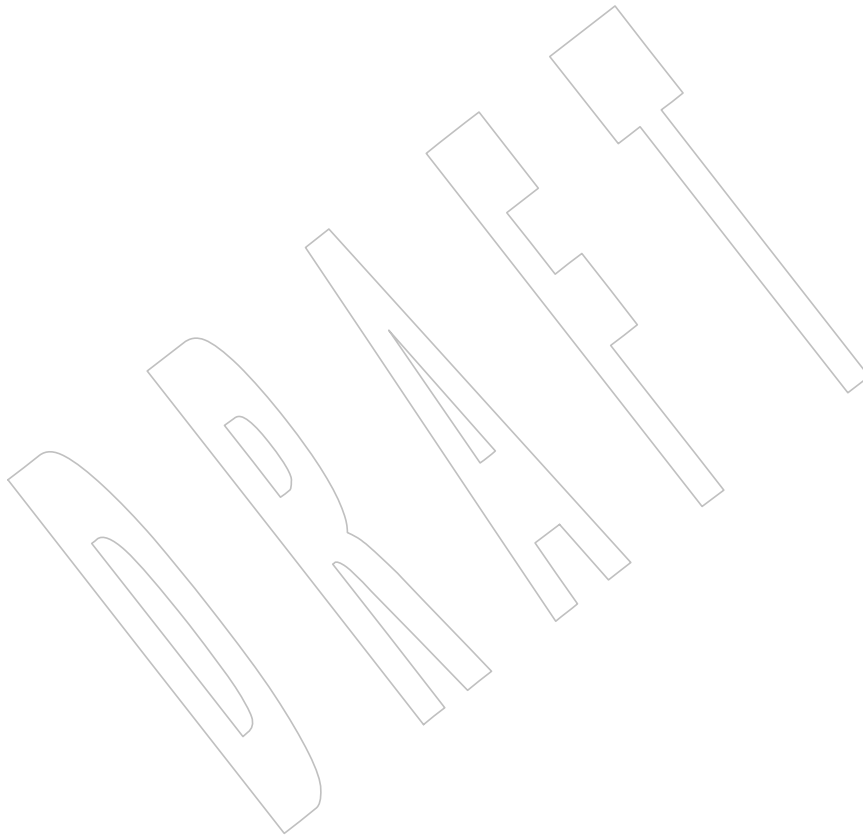
8356 **SEE ALSO**
 8357 *catanh()*, the Base Definitions volume of IEEE Std 1003.1-200x, <complex.h>

8358 **CHANGE HISTORY**
 8359 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

8360 **NAME**
8361 `ctanl` — complex tangent functions

8362 **SYNOPSIS**
8363 `#include <complex.h>`
8364 `long double complex ctanl(long double complex z);`

8365 **DESCRIPTION**
8366 Refer to *ctan()*.



8367 **NAME**8368 `ctermid` — generate a pathname for the controlling terminal8369 **SYNOPSIS**

```
8370 CX #include <stdio.h>
8371 char *ctermid(char *s);
```

8372 **DESCRIPTION**

8373 The `ctermid()` function shall generate a string that, when used as a pathname, refers to the
 8374 current controlling terminal for the current process. If `ctermid()` returns a pathname, access to the
 8375 file is not guaranteed.

8376 If the application uses any of the `_POSIX_THREAD_SAFE_FUNCTIONS` or `_POSIX_THREADS`
 8377 functions, it shall ensure that the `ctermid()` function is called with a non-NULL parameter.

8378 **RETURN VALUE**

8379 If `s` is a null pointer, the string shall be generated in an area that may be static (and therefore may
 8380 be overwritten by each call), the address of which shall be returned. Otherwise, `s` is assumed to
 8381 point to a character array of at least `L_ctermid` bytes; the string is placed in this array and the
 8382 value of `s` shall be returned. The symbolic constant `L_ctermid` is defined in `<stdio.h>`, and shall
 8383 have a value greater than 0.

8384 The `ctermid()` function shall return an empty string if the pathname that would refer to the
 8385 controlling terminal cannot be determined, or if the function is unsuccessful.

8386 **ERRORS**

8387 No errors are defined.

8388 **EXAMPLES**8389 **Determining the Controlling Terminal for the Current Process**

8390 The following example returns a pointer to a string that identifies the controlling terminal for the
 8391 current process. The pathname for the terminal is stored in the array pointed to by the `ptr`
 8392 argument, which has a size of `L_ctermid` bytes, as indicated by the `term` argument.

```
8393 #include <stdio.h>
8394 ...
8395 char term[L_ctermid];
8396 char *ptr;
8397 ptr = ctermid(term);
```

8398 **APPLICATION USAGE**

8399 The difference between `ctermid()` and `ttyname()` is that `ttyname()` must be handed a file
 8400 descriptor and return a path of the terminal associated with that file descriptor, while `ctermid()`
 8401 returns a string (such as `"/dev/tty"`) that refers to the current controlling terminal if used as a
 8402 pathname.

8403 **RATIONALE**

8404 `L_ctermid` must be defined appropriately for a given implementation and must be greater than
 8405 zero so that array declarations using it are accepted by the compiler. The value includes the
 8406 terminating null byte.

8407 Conforming applications that use threads cannot call `ctermid()` with NULL as the parameter if
 8408 either `_POSIX_THREAD_SAFE_FUNCTIONS` or `_POSIX_THREADS` is defined. If `s` is not
 8409 NULL, the `ctermid()` function generates a string that, when used as a pathname, refers to the

8410 current controlling terminal for the current process. If *s* is NULL, the return value of *ctermid()* is
8411 undefined.

8412 There is no additional burden on the programmer—changing to use a hypothetical thread-safe
8413 version of *ctermid()* along with allocating a buffer is more of a burden than merely allocating a
8414 buffer. Application code should not assume that the returned string is short, as some
8415 implementations have more than two pathname components before reaching a logical device
8416 name.

8417 **FUTURE DIRECTIONS**

8418 None.

8419 **SEE ALSO**

8420 *ttyname()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>

8421 **CHANGE HISTORY**

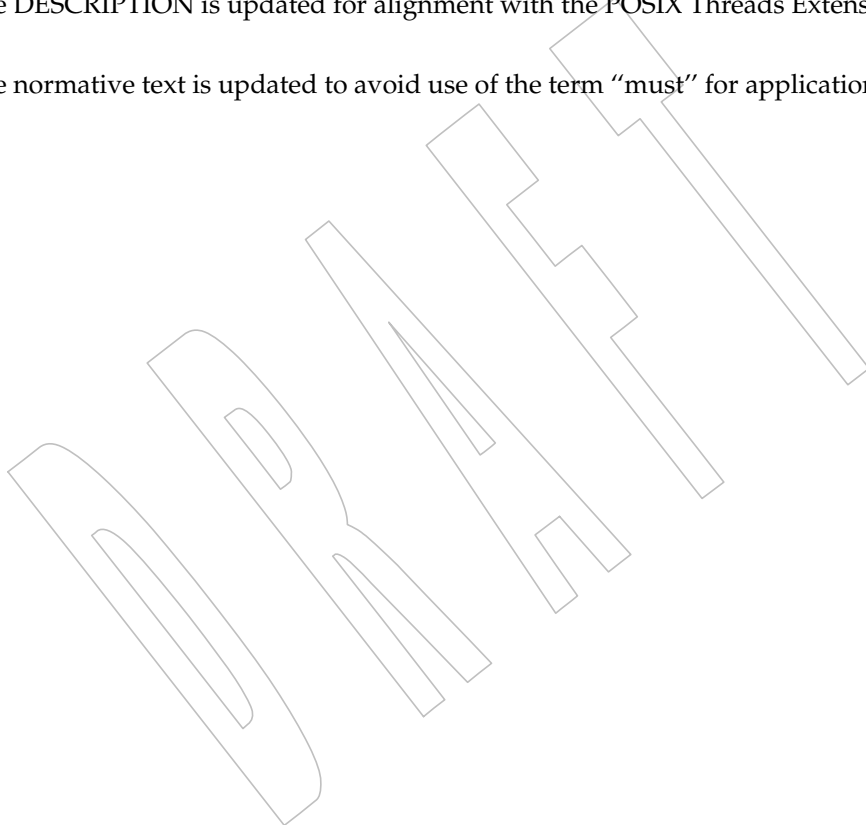
8422 First released in Issue 1. Derived from Issue 1 of the SVID.

8423 **Issue 5**

8424 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

8425 **Issue 6**

8426 The normative text is updated to avoid use of the term “must” for application requirements.



8427 **NAME**

8428 ctime, ctime_r — convert a time value to a date and time string

8429 **SYNOPSIS**

```
8430 OB #include <time.h>
8431 char *ctime(const time_t *clock);
8432 OB CX char *ctime_r(const time_t *clock, char *buf);
```

8433 **DESCRIPTION**

8434 CX For *ctime()*: The functionality described on this reference page is aligned with the ISO C
 8435 standard. Any conflict between the requirements described here and the ISO C standard is
 8436 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

8437 The *ctime()* function shall convert the time pointed to by *clock*, representing time in seconds
 8438 since the Epoch, to local time in the form of a string. It shall be equivalent to:

```
8439 asctime(localtime(clock))
```

8440 CX The *asctime()*, *ctime()*, *gmtime()*, and *localtime()* functions shall return values in one of two static
 8441 objects: a broken-down time structure and an array of **char**. Execution of any of the functions
 8442 may overwrite the information returned in either of these objects by any of the other functions.

8443 The *ctime()* function need not be thread-safe. A function that is not required to be thread-safe is
 8444 not required to be reentrant.

8445 The *ctime_r()* function shall convert the calendar time pointed to by *clock* to local time in exactly
 8446 the same form as *ctime()* and put the string into the array pointed to by *buf* (which shall be at
 8447 least 26 bytes in size) and return *buf*.

8448 Unlike *ctime()*, the thread-safe version *ctime_r()* is not required to set *tzname*.

8449 **RETURN VALUE**

8450 The *ctime()* function shall return the pointer returned by *asctime()* with that broken-down time
 8451 as an argument.

8452 CX Upon successful completion, *ctime_r()* shall return a pointer to the string pointed to by *buf*.
 8453 When an error is encountered, a null pointer shall be returned.

8454 **ERRORS**

8455 No errors are defined.

8456 **EXAMPLES**

8457 None.

8458 **APPLICATION USAGE**

8459 These functions are included only for compatibility with older implementations. They have
 8460 undefined behavior if the resulting string would be too long, so the use of these functions
 8461 should be discouraged. On implementations that do not detect output string length overflow, it
 8462 is possible to overflow the output buffers in such a way as to cause applications to fail, or
 8463 possible system security violations. Also, these functions do not support localized date and time
 8464 formats. To avoid these problems, applications should use *strftime()* to generate strings from
 8465 broken-down times.

8466 Values for the broken-down time structure can be obtained by calling *gmtime()* or *localtime()*.

8467 The *ctime_r()* function is thread-safe and shall return values in a user-supplied buffer instead of
 8468 possibly using a static data area that may be overwritten by each call.

8469 Attempts to use *ctime()* or *ctime_r()* for times before the Epoch or for times beyond the year 9999
8470 produce undefined results. Refer to *asctime()* (on page 130).

RATIONALE

8471 The standards developers decided to mark the *ctime()* and *ctime_r()* functions obsolescent even
8472 though they are in the ISO C standard due to the possibility of buffer overflow. The ISO C
8473 standard also provides the *strptime()* function which can be used to avoid these problems.
8474

FUTURE DIRECTIONS

8475 These functions may be removed in a future version.
8476

SEE ALSO

8477 *asctime()*, *clock()*, *difftime()*, *gmtime()*, *localtime()*, *mktime()*, *strptime()*, *strptime()*, *time()*, *utime()*,
8478 the Base Definitions volume of IEEE Std 1003.1-200x, <**time.h**>
8479

CHANGE HISTORY

8480 First released in Issue 1. Derived from Issue 1 of the SVID.
8481

Issue 5

8482 Normative text previously in the APPLICATION USAGE section is moved to the
8483 DESCRIPTION.
8484

8485 The *ctime_r()* function is included for alignment with the POSIX Threads Extension.

8486 A note indicating that the *ctime()* function need not be reentrant is added to the DESCRIPTION.

Issue 6

8487 Extensions beyond the ISO C standard are marked.
8488

8489 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

8490 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
8491 its avoidance of possibly using a static data area.

Issue 7

8492 SD5-XSH-ERN-25 is applied, updating the APPLICATION USAGE.
8493

8494 Austin Group Interpretation 1003.1-2001 #053 is applied, marking these functions obsolescent.

8495 The *ctime_r()* function is moved from the Thread-Safe Functions option to the Base.

8496 **NAME**
8497 daylight — daylight savings time flag

8498 **SYNOPSIS**

```
8499 XSI #include <time.h>  
8500     extern int daylight;
```

8501 **DESCRIPTION**

8502 Refer to [tzset\(\)](#).

8503 NAME

8504 dbm_clearerr, dbm_close, dbm_delete, dbm_error, dbm_fetch, dbm_firstkey, dbm_nextkey,
8505 dbm_open, dbm_store — database functions

8506 SYNOPSIS

```
8507 XSI #include <ndbm.h>
8508
8508 int dbm_clearerr(DBM *db);
8509 void dbm_close(DBM *db);
8510 int dbm_delete(DBM *db, datum key);
8511 int dbm_error(DBM *db);
8512 datum dbm_fetch(DBM *db, datum key);
8513 datum dbm_firstkey(DBM *db);
8514 datum dbm_nextkey(DBM *db);
8515 DBM *dbm_open(const char *file, int open_flags, mode_t file_mode);
8516 int dbm_store(DBM *db, datum key, datum content, int store_mode);
```

8517 DESCRIPTION

8518 These functions create, access, and modify a database.

8519 A **datum** consists of at least two members, *dptr* and *dsize*. The *dptr* member points to an object
8520 that is *dsize* bytes in length. Arbitrary binary data, as well as character strings, may be stored in
8521 the object pointed to by *dptr*.

8522 A database shall be stored in one or two files. When one file is used, the name of the database
8523 file shall be formed by appending the suffix **.db** to the *file* argument given to *dbm_open()*. When
8524 two files are used, the names of the database files shall be formed by appending the suffixes **.dir**
8525 and **.pag** respectively to the *file* argument.

8526 The *dbm_open()* function shall open a database. The *file* argument to the function is the
8527 pathname of the database. The *open_flags* argument has the same meaning as the *flags* argument
8528 of *open()* except that a database opened for write-only access opens the files for read and write
8529 access and the behavior of the **O_APPEND** flag is unspecified. The *file_mode* argument has the
8530 same meaning as the third argument of *open()*.

8531 The *dbm_open()* function need not accept pathnames longer than **{PATH_MAX}-4** bytes
8532 (including the terminating null), or pathnames with a last component longer than
8533 **{NAME_MAX}-4** bytes (excluding the terminating null).

8534 The *dbm_close()* function shall close a database. The application shall ensure that argument *db* is
8535 a pointer to a **dbm** structure that has been returned from a call to *dbm_open()*.

8536 These database functions shall support an internal block size large enough to support
8537 key/content pairs of at least 1 023 bytes.

8538 The *dbm_fetch()* function shall read a record from a database. The argument *db* is a pointer to a
8539 database structure that has been returned from a call to *dbm_open()*. The argument *key* is a
8540 **datum** that has been initialized by the application to the value of the key that matches the key of
8541 the record the program is fetching.

8542 The *dbm_store()* function shall write a record to a database. The argument *db* is a pointer to a
8543 database structure that has been returned from a call to *dbm_open()*. The argument *key* is a
8544 **datum** that has been initialized by the application to the value of the key that identifies (for
8545 subsequent reading, writing, or deleting) the record the application is writing. The argument
8546 *content* is a **datum** that has been initialized by the application to the value of the record the
8547 program is writing. The argument *store_mode* controls whether *dbm_store()* replaces any pre-

8548 existing record that has the same key that is specified by the *key* argument. The application shall
 8549 set *store_mode* to either DBM_INSERT or DBM_REPLACE. If the database contains a record that
 8550 matches the *key* argument and *store_mode* is DBM_REPLACE, the existing record shall be
 8551 replaced with the new record. If the database contains a record that matches the *key* argument
 8552 and *store_mode* is DBM_INSERT, the existing record shall be left unchanged and the new record
 8553 ignored. If the database does not contain a record that matches the *key* argument and *store_mode*
 8554 is either DBM_INSERT or DBM_REPLACE, the new record shall be inserted in the database.

8555 If the sum of a key/content pair exceeds the internal block size, the result is unspecified.
 8556 Moreover, the application shall ensure that all key/content pairs that hash together fit on a
 8557 single block. The *dbm_store()* function shall return an error in the event that a disk block fills
 8558 with inseparable data.

8559 The *dbm_delete()* function shall delete a record and its key from the database. The argument *db* is
 8560 a pointer to a database structure that has been returned from a call to *dbm_open()*. The argument
 8561 *key* is a **datum** that has been initialized by the application to the value of the key that identifies
 8562 the record the program is deleting.

8563 The *dbm_firstkey()* function shall return the first key in the database. The argument *db* is a
 8564 pointer to a database structure that has been returned from a call to *dbm_open()*.

8565 The *dbm_nextkey()* function shall return the next key in the database. The argument *db* is a
 8566 pointer to a database structure that has been returned from a call to *dbm_open()*. The application
 8567 shall ensure that the *dbm_firstkey()* function is called before calling *dbm_nextkey()*. Subsequent
 8568 calls to *dbm_nextkey()* return the next key until all of the keys in the database have been
 8569 returned.

8570 The *dbm_error()* function shall return the error condition of the database. The argument *db* is a
 8571 pointer to a database structure that has been returned from a call to *dbm_open()*.

8572 The *dbm_clearerr()* function shall clear the error condition of the database. The argument *db* is a
 8573 pointer to a database structure that has been returned from a call to *dbm_open()*.

8574 The *dptr* pointers returned by these functions may point into static storage that may be changed
 8575 by subsequent calls.

8576 These functions need not be thread-safe. A function that is not required to be thread-safe is not
 8577 required to be reentrant.

8578 RETURN VALUE

8579 The *dbm_store()* and *dbm_delete()* functions shall return 0 when they succeed and a negative
 8580 value when they fail.

8581 The *dbm_store()* function shall return 1 if it is called with a *flags* value of DBM_INSERT and the
 8582 function finds an existing record with the same key.

8583 The *dbm_error()* function shall return 0 if the error condition is not set and return a non-zero
 8584 value if the error condition is set.

8585 The return value of *dbm_clearerr()* is unspecified.

8586 The *dbm_firstkey()* and *dbm_nextkey()* functions shall return a key **datum**. When the end of the
 8587 database is reached, the *dptr* member of the key is a null pointer. If an error is detected, the *dptr*
 8588 member of the key shall be a null pointer and the error condition of the database shall be set.

8589 The *dbm_fetch()* function shall return a content **datum**. If no record in the database matches the
 8590 key or if an error condition has been detected in the database, the *dptr* member of the content
 8591 shall be a null pointer.

8592 The *dbm_open()* function shall return a pointer to a database structure. If an error is detected
 8593 during the operation, *dbm_open()* shall return a (DBM *)0.

8594 ERRORS

8595 No errors are defined.

8596 EXAMPLES

8597 None.

8598 APPLICATION USAGE

8599 The following code can be used to traverse the database:

```
8600 for(key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))
```

8601 The *dbm_** functions provided in this library should not be confused in any way with those of a
8602 general-purpose database management system. These functions do not provide for multiple
8603 search keys per entry, they do not protect against multi-user access (in other words they do not
8604 lock records or files), and they do not provide the many other useful database functions that are
8605 found in more robust database management systems. Creating and updating databases by use of
8606 these functions is relatively slow because of data copies that occur upon hash collisions. These
8607 functions are useful for applications requiring fast lookup of relatively static information that is
8608 to be indexed by a single key.

8609 Note that a strictly conforming application is extremely limited by these functions: since there is
8610 no way to determine that the keys in use do not all hash to the same value (although that would
8611 be rare), a strictly conforming application cannot be guaranteed that it can store more than one
8612 block's worth of data in the database. As long as a key collision does not occur, additional data
8613 may be stored, but because there is no way to determine whether an error is due to a key
8614 collision or some other error condition (*dbm_error()* being effectively a Boolean), once an error is
8615 detected, the application is effectively limited to guessing what the error might be if it wishes to
8616 continue using these functions.

8617 The *dbm_delete()* function need not physically reclaim file space, although it does make it
8618 available for reuse by the database.

8619 After calling *dbm_store()* or *dbm_delete()* during a pass through the keys by *dbm_firstkey()* and
8620 *dbm_nextkey()*, the application should reset the database by calling *dbm_firstkey()* before again
8621 calling *dbm_nextkey()*. The contents of these files are unspecified and may not be portable.

8622 Applications should take care that database pathname arguments specified to *dbm_open()* are
8623 not prefixes of unrelated files. This might be done, for example, by placing databases in a
8624 separate directory.

8625 Since some implementations use three characters for a suffix and others use four characters for a
8626 suffix, applications should ensure that the maximum portable pathname length passed to
8627 *dbm_open()* is no greater than {PATH_MAX}-4 bytes, with the last component of the pathname
8628 no greater than {NAME_MAX}-4 bytes.

8629 RATIONALE

8630 Previously the standard required the database to be stored in two files, one file being a directory
8631 containing a bitmap of keys and having **.dir** as its suffix. The second file containing all data and
8632 having **.pag** as its suffix. This has been changed not to specify the use of the files and to allow
8633 newer implementations of the Berkeley DB interface using a single file that have evolved while
8634 remaining compatible with the application programming interface. The standard developers
8635 considered removing the specific suffixes altogether but decided to retain them so as not to
8636 pollute the application file name space more than necessary and to allow for portable backups of
8637 the database.

8638 FUTURE DIRECTIONS

8639 None.

8640
8641
8642
8643
8644
8645
8646
8647
8648
8649
8650
8651
8652
8653

SEE ALSO

open(), the Base Definitions volume of IEEE Std 1003.1-200x, <ndbm.h>

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

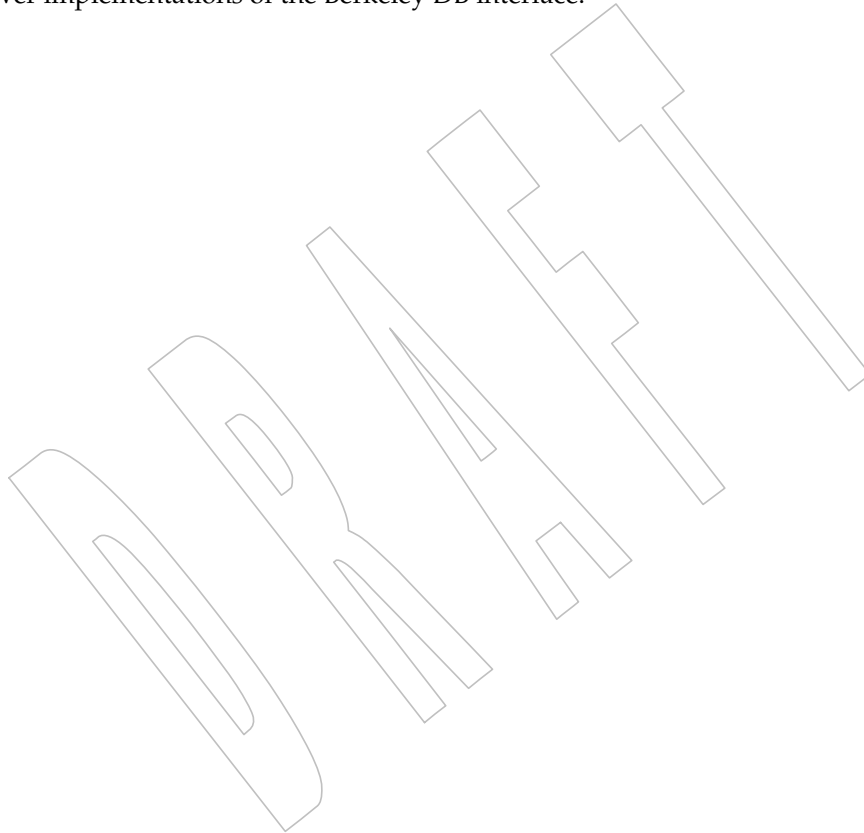
A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

Issue 6

The normative text is updated to avoid use of the term “must” for application requirements.

Issue 7

Austin Group Interpretation 1003.1-2001 #042 is applied so that the DESCRIPTION permits newer implementations of the Berkeley DB interface.



difftime()**NAME**

difftime — compute the difference between two calendar time values

SYNOPSIS

```
#include <time.h>

double difftime(time_t time1, time_t time0);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *difftime()* function shall compute the differ

8682 **NAME**

8683 dirfd — extract the file descriptor used by a DIR stream

8684 **SYNOPSIS**8685 #include <dirent.h>
8686 int dirfd(DIR *dirp);8687 **DESCRIPTION**8688 The *dirfd()* function shall return a file descriptor referring to the same directory as the *dirp*
8689 argument. This file descriptor shall be closed by a call to *closedir()*. The behavior of future calls
8690 to *readdir()* and *readdir_r()* is undefined if the application attempts to alter the file position
8691 indicator using the returned file descriptor. The behavior of future calls to *closedir()*, *readdir()*,
8692 and *readdir_r()* is undefined if the application attempts to close the file descriptor.8693 **RETURN VALUE**8694 Upon successful completion, the *dirfd()* function shall return an integer which contains a file
8695 descriptor for the stream pointed to by *dirp*. Otherwise, it shall return -1 and may set *errno* to
8696 indicate the error.8697 **ERRORS**8698 The *dirfd()* function may fail if:

- 8699 [EINVAL] The
- dirp*
- argument does not refer to a valid directory stream.
-
- 8700 [ENOTSUP] The implementation does not support the association of a file descriptor with
-
- 8701 a directory.

8702 **EXAMPLES**

8703 None.

8704 **APPLICATION USAGE**8705 The *dirfd()* function is intended to be a mechanism by which an application may obtain a file
8706 descriptor to use for the *fhdir()* function.8707 **RATIONALE**8708 This interface was introduced because the Base Definitions volume of IEEE Std 1003.1-200x does
8709 not make public the **DIR** data structure. Applications tend to use the *fhdir()* function on the file
8710 descriptor returned by this interface, and this has proven useful for security reasons; in
8711 particular, it is a better technique than others where directory names might change.8712 The description uses the term “a file descriptor” rather than “the file descriptor”. The
8713 implication intended is that an implementation that does not use an *fd* for *diropen()* could still
8714 *open()* the directory to implement the *dirfd()* function. Such a descriptor must be closed later
8715 during a call to *closedir()*.8716 An implementation that does not support file descriptors referring to directories may fail with
8717 [ENOTSUP].8718 If it is necessary to allocate an *fd* to be returned by *dirfd()*, it should be done at the time of a call
8719 to *opendir()*.8720 **FUTURE DIRECTIONS**

8721 None.

dirfd()

8722

SEE ALSO

8723

closedir(), *fchdir()*, *fdopendir()*, *fileno()*, *open()*, *opendir()*, *readdir()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**

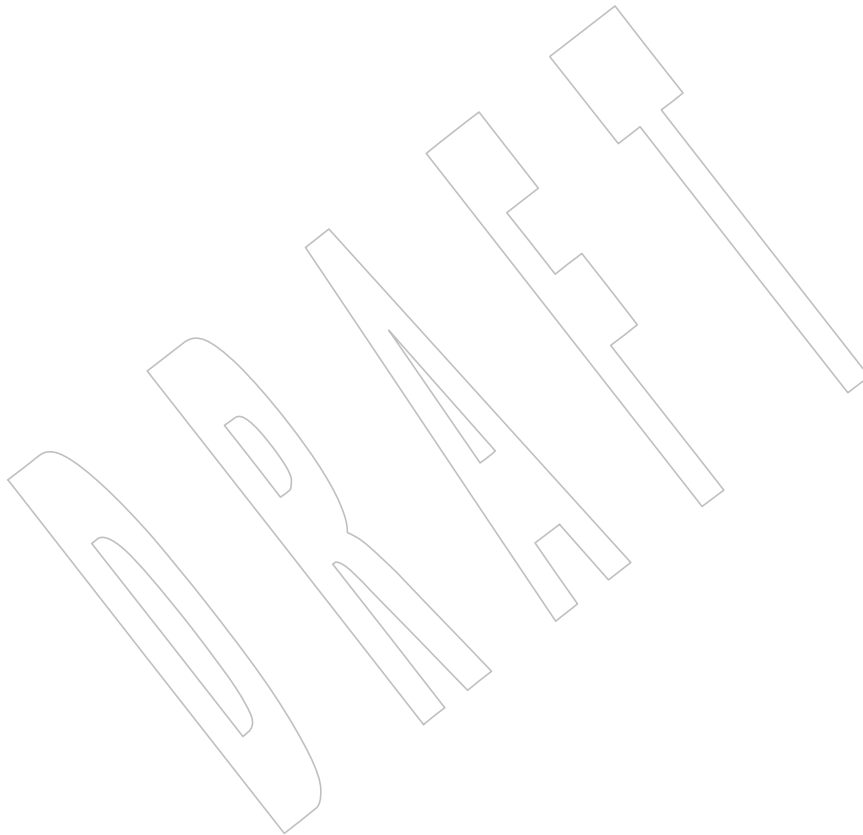
8724

8725

CHANGE HISTORY

8726

First released in Issue 7.



8727 **NAME**8728 `dirname` — report the parent directory name of a file pathname8729 **SYNOPSIS**

```
8730 XSI #include <libgen.h>
8731 char *dirname(char *path);
```

8732 **DESCRIPTION**

8733 The `dirname()` function shall take a pointer to a character string that contains a pathname, and
 8734 return a pointer to a string that is a pathname of the parent directory of that file. Trailing '/'
 8735 characters in the path are not counted as part of the path.

8736 If *path* does not contain a '/', then `dirname()` shall return a pointer to the string ".". If *path* is a
 8737 null pointer or points to an empty string, `dirname()` shall return a pointer to the string ".".

8738 The `dirname()` function need not be thread-safe. A function that is not required to be thread-safe
 8739 is not required to be reentrant.

8740 **RETURN VALUE**

8741 The `dirname()` function shall return a pointer to a string that is the parent directory of *path*. If
 8742 *path* is a null pointer or points to an empty string, a pointer to a string "." is returned.

8743 The `dirname()` function may modify the string pointed to by *path*, and may return a pointer to
 8744 static storage that may then be overwritten by subsequent calls to `dirname()`.

8745 **ERRORS**

8746 No errors are defined.

8747 **EXAMPLES**

8748 The following code fragment reads a pathname, changes the current working directory to the
 8749 parent directory, and opens the file.

```
8750 char path[PATH_MAX], *pathcopy;
8751 int fd;
8752 fgets(path, PATH_MAX, stdin);
8753 pathcopy = strdup(path);
8754 chdir(dirname(pathcopy));
8755 fd = open(basename(path), O_RDONLY);
```

8756 **Sample Input and Output Strings for `dirname()`**

8757 In the following table, the input string is the value pointed to by *path*, and the output string is
 8758 the return value of the `dirname()` function.

Input String	Output String
"/usr/lib"	"/usr"
"/usr/"	"/"
"usr"	."
"/"	"/"
."	."
".."	."

8766 **Changing the Current Directory to the Parent Directory**

8767 The following program fragment reads a pathname, changes the current working directory to
 8768 the parent directory, and opens the file.

```

8769 #include <unistd.h>
8770 #include <limits.h>
8771 #include <stdio.h>
8772 #include <fcntl.h>
8773 #include <string.h>
8774 #include <libgen.h>
8775 ...
8776 char path[PATH_MAX], *pathcopy;
8777 int fd;
8778 ...
8779 fgets(path, PATH_MAX, stdin);
8780 pathcopy = strdup(path);
8781 chdir(dirname(pathcopy));
8782 fd = open(basename(path), O_RDONLY);

```

8783 **APPLICATION USAGE**

8784 The *dirname()* and *basename()* functions together yield a complete pathname. The expression
 8785 *dirname(path)* obtains the pathname of the directory where *basename(path)* is found.

8786 Since the meaning of the leading `"/"` is implementation-defined, *dirname("//foo)* may return
 8787 either `"/"` or `'/'` (but nothing else).

8788 **RATIONALE**

8789 None.

8790 **FUTURE DIRECTIONS**

8791 None.

8792 **SEE ALSO**

8793 *basename()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<libgen.h>**

8794 **CHANGE HISTORY**

8795 First released in Issue 4, Version 2.

8796 **Issue 5**

8797 Moved from X/OPEN UNIX extension to BASE.

8798 Normative text previously in the APPLICATION USAGE section is moved to the
 8799 DESCRIPTION.

8800 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

8801 **NAME**

8802 div — compute the quotient and remainder of an integer division

8803 **SYNOPSIS**

8804 #include <stdlib.h>

8805 div_t div(int numer, int denom);

8806 **DESCRIPTION**8807 CX The functionality described on this reference page is aligned with the ISO C standard. Any
8808 conflict between the requirements described here and the ISO C standard is unintentional. This
8809 volume of IEEE Std 1003.1-200x defers to the ISO C standard.8810 The *div()* function shall compute the quotient and remainder of the division of the numerator
8811 *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the integer
8812 of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be
8813 represented, the behavior is undefined; otherwise, *quot*denom+rem* shall equal *numer*.8814 **RETURN VALUE**8815 The *div()* function shall return a structure of type **div_t**, comprising both the quotient and the
8816 remainder. The structure includes the following members, in any order:8817 int quot; /* quotient */
8818 int rem; /* remainder */8819 **ERRORS**

8820 No errors are defined.

8821 **EXAMPLES**

8822 None.

8823 **APPLICATION USAGE**

8824 None.

8825 **RATIONALE**

8826 None.

8827 **FUTURE DIRECTIONS**

8828 None.

8829 **SEE ALSO**8830 *ldiv()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdlib.h>8831 **CHANGE HISTORY**

8832 First released in Issue 4. Derived from the ISO C standard.

8833 **NAME**8834 `dlclose` — close a `dlopen()` object8835 **SYNOPSIS**8836

```
#include <dlfcn.h>
8837 int dlclosel(void *handle);
```

8838 **DESCRIPTION**8839 The `dlclose()` function shall inform the system that the object referenced by a `handle` returned
8840 from a previous `dlopen()` invocation is no longer needed by the application.8841 The use of `dlclose()` reflects a statement of intent on the part of the process, but does not create
8842 any requirement upon the implementation, such as removal of the code or symbols referenced
8843 by `handle`. Once an object has been closed using `dlclose()` an application should assume that its
8844 symbols are no longer available to `dlsym()`. All objects loaded automatically as a result of
8845 invoking `dlopen()` on the referenced object shall also be closed if this is the last reference to it.8846 Although a `dlclose()` operation is not required to remove structures from an address space,
8847 neither is an implementation prohibited from doing so. The only restriction on such a removal is
8848 that no object shall be removed to which references have been relocated, until or unless all such
8849 references are removed. For instance, an object that had been loaded with a `dlopen()` operation
8850 specifying the `RTLD_GLOBAL` flag might provide a target for dynamic relocations performed in
8851 the processing of other objects—in such environments, an application may assume that no
8852 relocation, once made, shall be undone or remade unless the object requiring the relocation has
8853 itself been removed.8854 **RETURN VALUE**8855 If the referenced object was successfully closed, `dlclose()` shall return 0. If the object could not be
8856 closed, or if `handle` does not refer to an open object, `dlclose()` shall return a non-zero value. More
8857 detailed diagnostic information shall be available through `dLError()`.8858 **ERRORS**

8859 No errors are defined.

8860 **EXAMPLES**8861 The following example illustrates use of `dlopen()` and `dlclose()`:8862

```
...
8863 /* Open a dynamic library and then close it ... */
8864 #include <dlfcn.h>
8865 void *mylib;
8866 int eret;
8867 mylib = dlopen("mylib.so", RTLD_LOCAL | RTLD_LAZY);
8868 ...
8869 eret = dlclosel(mylib);
8870 ...
```

8871 **APPLICATION USAGE**8872 A conforming application should employ a `handle` returned from a `dlopen()` invocation only
8873 within a given scope bracketed by the `dlopen()` and `dlclose()` operations. Implementations are
8874 free to use reference counting or other techniques such that multiple calls to `dlopen()` referencing
8875 the same object may return the same object for `handle`. Implementations are also free to reuse a
8876 `handle`. For these reasons, the value of a `handle` must be treated as an opaque object by the
8877 application, used only in calls to `dlsym()` and `dlclose()`.

8878

RATIONALE

8879

None.

8880

FUTURE DIRECTIONS

8881

None.

8882

SEE ALSO

8883

derror(), *dlopen()*, *dlsym()*, the Base Definitions volume of IEEE Std 1003.1-200x, <dlfcn.h>

8884

CHANGE HISTORY

8885

First released in Issue 5.

8886

Issue 6

8887

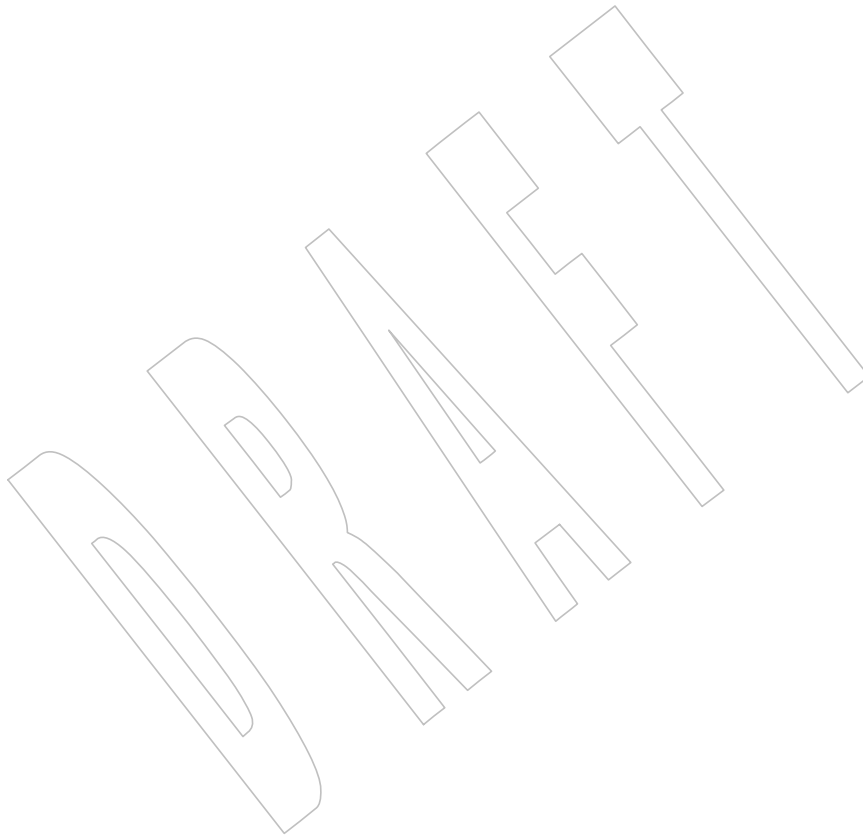
The DESCRIPTION is updated to say that the referenced object is closed “if this is the last reference to it”.

8888

8889

Issue 7

8890

The *dlopen()* function is moved from the XSI option to Base.

8891 **NAME**8892 `dlerror` — get diagnostic information8893 **SYNOPSIS**8894 `#include <dlfcn.h>`8895 `char *dlerror(void);`8896 **DESCRIPTION**

8897 The `dlerror()` function shall return a null-terminated character string (with no trailing
 8898 <newline>) that describes the last error that occurred during dynamic linking processing. If no
 8899 dynamic linking errors have occurred since the last invocation of `dlerror()`, `dlerror()` shall return
 8900 NULL. Thus, invoking `dlerror()` a second time, immediately following a prior invocation, shall
 8901 result in NULL being returned.

8902 The `dlerror()` function need not be thread-safe. A function that is not required to be thread-safe is
 8903 not required to be reentrant.

8904 **RETURN VALUE**

8905 If successful, `dlerror()` shall return a null-terminated character string; otherwise, NULL shall be
 8906 returned.

8907 **ERRORS**

8908 No errors are defined.

8909 **EXAMPLES**

8910 The following example prints out the last dynamic linking error:

```
8911 ...
8912 #include <dlfcn.h>
8913 char *errstr;
8914 errstr = dlerror();
8915 if (errstr != NULL)
8916     printf ("A dynamic linking error occurred: (%s)\n", errstr);
8917 ...
```

8918 **APPLICATION USAGE**

8919 The messages returned by `dlerror()` may reside in a static buffer that is overwritten on each call
 8920 to `dlerror()`. Application code should not write to this buffer. Programs wishing to preserve an
 8921 error message should make their own copies of that message. Depending on the application
 8922 environment with respect to asynchronous execution events, such as signals or other
 8923 asynchronous computation sharing the address space, conforming applications should use a
 8924 critical section to retrieve the error pointer and buffer.

8925 **RATIONALE**

8926 None.

8927 **FUTURE DIRECTIONS**

8928 None.

8929 **SEE ALSO**8930 `dlclose()`, `dlopen()`, `dlsym()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<dlfcn.h>`

8931

CHANGE HISTORY

8932

First released in Issue 5.

8933

Issue 6

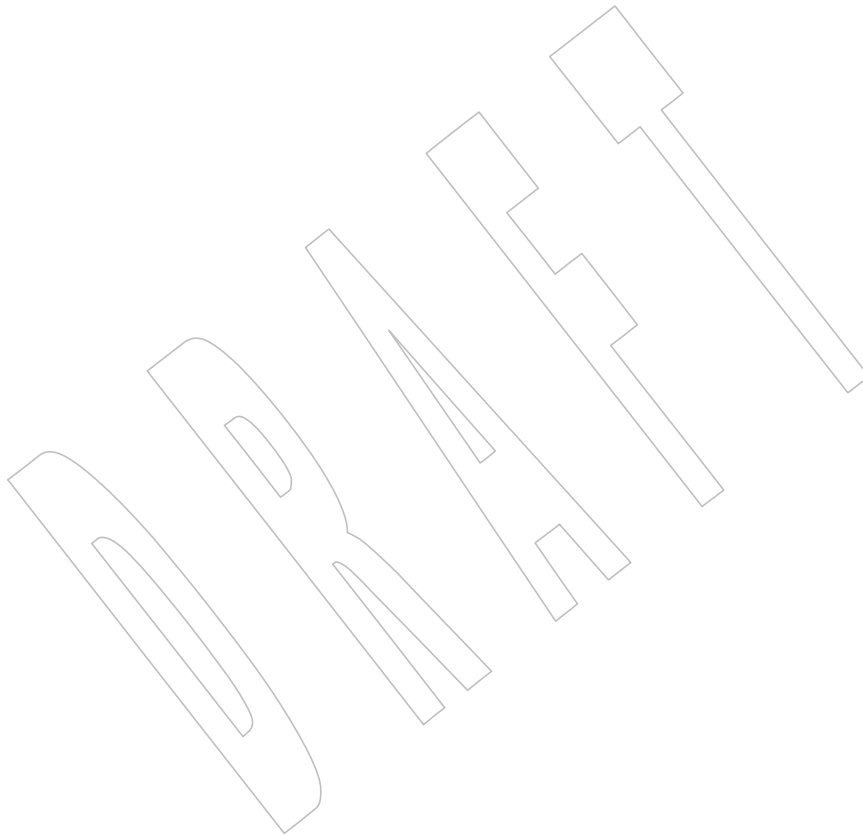
8934

In the DESCRIPTION the note about reentrancy and thread-safety is added.

8935

Issue 7

8936

The *derror()* function is moved from the XSI option to the Base.

8937 **NAME**8938 `dlopen` — gain access to an executable object file8939 **SYNOPSIS**8940 `#include <dlfcn.h>`8941 `void *dlopen(const char *file, int mode);`8942 **DESCRIPTION**

8943 The `dlopen()` function shall make an executable object file specified by `file` available to the calling
 8944 program. The class of files eligible for this operation and the manner of their construction are
 8945 implementation-defined, though typically such files are executable objects such as shared
 8946 libraries, relocatable files, or programs. Note that some implementations permit the construction
 8947 of dependencies between such objects that are embedded within files. In such cases, a `dlopen()`
 8948 operation shall load such dependencies in addition to the object referenced by `file`.
 8949 Implementations may also impose specific constraints on the construction of programs that can
 8950 employ `dlopen()` and its related services.

8951 A successful `dlopen()` shall return a *handle* which the caller may use on subsequent calls to
 8952 `dlsym()` and `dlclose()`. The value of this *handle* should not be interpreted in any way by the caller.

8953 The `file` argument is used to construct a pathname to the object file. If `file` contains a slash
 8954 character, the `file` argument is used as the pathname for the file. Otherwise, `file` is used in an
 8955 implementation-defined manner to yield a pathname.

8956 If the value of `file` is 0, `dlopen()` shall provide a *handle* on a global symbol object. This object shall
 8957 provide access to the symbols from an ordered set of objects consisting of the original program
 8958 image file, together with any objects loaded at program start-up as specified by that process
 8959 image file (for example, shared libraries), and the set of objects loaded using a `dlopen()` operation
 8960 together with the `RTLD_GLOBAL` flag. As the latter set of objects can change during execution,
 8961 the set identified by *handle* can also change dynamically.

8962 Only a single copy of an object file is brought into the address space, even if `dlopen()` is invoked
 8963 multiple times in reference to the file, and even if different pathnames are used to reference the
 8964 file.

8965 The `mode` parameter describes how `dlopen()` shall operate upon `file` with respect to the processing
 8966 of relocations and the scope of visibility of the symbols provided within `file`. When an object is
 8967 brought into the address space of a process, it may contain references to symbols whose
 8968 addresses are not known until the object is loaded. These references shall be relocated before the
 8969 symbols can be accessed. The `mode` parameter governs when these relocations take place and
 8970 may have the following values:

8971 **RTLD_LAZY** Relocations shall be performed at an implementation-defined time,
 8972 ranging from the time of the `dlopen()` call until the first reference to a
 8973 given symbol occurs. Specifying `RTLD_LAZY` should improve
 8974 performance on implementations supporting dynamic symbol binding as
 8975 a process may not reference all of the functions in any given object. And,
 8976 for systems supporting dynamic symbol resolution for normal process
 8977 execution, this behavior mimics the normal handling of process
 8978 execution.

8979 **RTLD_NOW** All necessary relocations shall be performed when the object is first
 8980 loaded. This may waste some processing if relocations are performed for
 8981 functions that are never referenced. This behavior may be useful for
 8982 applications that need to know as soon as an object is loaded that all
 8983 symbols referenced during execution are available.

8984 Any object loaded by *dlopen()* that requires relocations against global symbols can reference the
 8985 symbols in the original process image file, any objects loaded at program start-up, from the
 8986 object itself as well as any other object included in the same *dlopen()* invocation, and any objects
 8987 that were loaded in any *dlopen()* invocation and which specified the RTLD_GLOBAL flag. To
 8988 determine the scope of visibility for the symbols loaded with a *dlopen()* invocation, the *mode*
 8989 parameter should be a bitwise-inclusive OR with one of the following values:

8990	RTLD_GLOBAL	The object's symbols shall be made available for the relocation processing of any other object. In addition, symbol lookup using <i>dlopen(0, mode)</i> and an associated <i>dlsym()</i> allows objects loaded with this <i>mode</i> to be searched.
8991		
8992		
8993	RTLD_LOCAL	The object's symbols shall not be made available for the relocation processing of any other object.
8994		

8995 If neither RTLD_GLOBAL nor RTLD_LOCAL are specified, then the default behavior is
 8996 unspecified.

8997 If a *file* is specified in multiple *dlopen()* invocations, *mode* is interpreted at each invocation. Note,
 8998 however, that once RTLD_NOW has been specified all relocations shall have been completed
 8999 rendering further RTLD_NOW operations redundant and any further RTLD_LAZY operations
 9000 irrelevant. Similarly, note that once RTLD_GLOBAL has been specified the object shall maintain
 9001 the RTLD_GLOBAL status regardless of any previous or future specification of RTLD_LOCAL,
 9002 as long as the object remains in the address space (see *dlclose()*).

9003 Symbols introduced into a program through calls to *dlopen()* may be used in relocation activities.
 9004 Symbols so introduced may duplicate symbols already defined by the program or previous
 9005 *dlopen()* operations. To resolve the ambiguities such a situation might present, the resolution of a
 9006 symbol reference to symbol definition is based on a symbol resolution order. Two such
 9007 resolution orders are defined: *load* or *dependency* ordering. Load order establishes an ordering
 9008 among symbol definitions, such that the definition first loaded (including definitions from the
 9009 image file and any dependent objects loaded with it) has priority over objects added later (via
 9010 *dlopen()*). Load ordering is used in relocation processing. Dependency ordering uses a breadth-
 9011 first order starting with a given object, then all of its dependencies, then any dependents of
 9012 those, iterating until all dependencies are satisfied. With the exception of the global symbol
 9013 object obtained via a *dlopen()* operation on a *file* of 0, dependency ordering is used by the
 9014 *dlsym()* function. Load ordering is used in *dlsym()* operations upon the global symbol object.

9015 When an object is first made accessible via *dlopen()* it and its dependent objects are added in
 9016 dependency order. Once all the objects are added, relocations are performed using load order.
 9017 Note that if an object or its dependencies had been previously loaded, the load and dependency
 9018 orders may yield different resolutions.

9019 The symbols introduced by *dlopen()* operations and available through *dlsym()* are at a minimum
 9020 those which are exported as symbols of global scope by the object. Typically such symbols shall
 9021 be those that were specified in (for example) C source code as having *extern* linkage. The precise
 9022 manner in which an implementation constructs the set of exported symbols for a *dlopen()* object
 9023 is specified by that implementation.

9024 RETURN VALUE

9025 If *file* cannot be found, cannot be opened for reading, is not of an appropriate object format for
 9026 processing by *dlopen()*, or if an error occurs during the process of loading *file* or relocating its
 9027 symbolic references, *dlopen()* shall return NULL. More detailed diagnostic information shall be
 9028 available through *dlerror()*.

9029 ERRORS

9030 No errors are defined.

dlopen()9031
9032
9033
9034
9035
9036
9037
9038
9039
9040
9041
9042
9043
9044
9045
9046
9047
9048**EXAMPLES**

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO*dlclose()*, *dLError()*, *dlsym()*, the Base Definitions volume of IEEE Std 1003.1-200x, <dlfcn.h>**CHANGE HISTORY**

First released in Issue 5.

Issue 6

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/21 is applied, changing the default behavior in the DESCRIPTION when neither RTLD_GLOBAL nor RTLD_LOCAL are specified from implementation-defined to unspecified.

Issue 7

The *dlopen()* function is moved from the XSI option to the Base.

9049 **NAME**9050 dlsym — obtain the address of a symbol from a *dlopen()* object9051 **SYNOPSIS**

9052 #include <dlfcn.h>

9053 void *dlsym(void *restrict handle, const char *restrict name);

9054 **DESCRIPTION**

9055 The *dlsym()* function shall obtain the address of a symbol defined within an object made
 9056 accessible through a *dlopen()* call. The *handle* argument is the value returned from a call to
 9057 *dlopen()* (and which has not since been released via a call to *dlclose()*), and *name* is the symbol's
 9058 name as a character string.

9059 The *dlsym()* function shall search for the named symbol in all objects loaded automatically as a
 9060 result of loading the object referenced by *handle* (see *dlopen()*). Load ordering is used in *dlsym()*
 9061 operations upon the global symbol object. The symbol resolution algorithm used shall be
 9062 dependency order as described in *dlopen()*.

9063 The RTLD_DEFAULT and RTLD_NEXT flags are reserved for future use.

9064 **RETURN VALUE**

9065 If *handle* does not refer to a valid object opened by *dlopen()*, or if the named symbol cannot be
 9066 found within any of the objects associated with *handle*, *dlsym()* shall return NULL. More
 9067 detailed diagnostic information shall be available through *dlderror()*.

9068 **ERRORS**

9069 No errors are defined.

9070 **EXAMPLES**

9071 The following example shows how *dlopen()* and *dlsym()* can be used to access either function or
 9072 data objects. For simplicity, error checking has been omitted.

```
9073 void    *handle;
9074 int     *iptr, (*fptr)(int);

9075 /* open the needed object */
9076 handle = dlopen("/usr/home/me/libfoo.so", RTLD_LOCAL | RTLD_LAZY);

9077 /* find the address of function and data objects */
9078 *(void **)(&fptr) = dlsym(handle, "my_function");
9079 iptr = (int *)dlsym(handle, "my_object");

9080 /* invoke function, passing value of integer as a parameter */
9081 (*fptr)(*iptr);
```

9082 **APPLICATION USAGE**

9083 Special purpose values for *handle* are reserved for future use. These values and their meanings
 9084 are:

9085 RTLD_DEFAULT The symbol lookup happens in the normal global scope; that is, a search for a
 9086 symbol using this handle would find the same definition as a direct use of this
 9087 symbol in the program code.

9088 RTLD_NEXT Specifies the next object after this one that defines *name*. *This one* refers to the
 9089 object containing the invocation of *dlsym()*. The *next* object is the one found
 9090 upon the application of a load order symbol resolution algorithm (see
 9091 *dlopen()*). The next object is either one of global scope (because it was
 9092 introduced as part of the original process image or because it was added with

a *dlopen()* operation including the `RTLD_GLOBAL` flag), or is an object that was included in the same *dlopen()* operation that loaded this one.

The `RTLD_NEXT` flag is useful to navigate an intentionally created hierarchy of multiply-defined symbols created through *interposition*. For example, if a program wished to create an implementation of *malloc()* that embedded some statistics gathering about memory allocations, such an implementation could use the real *malloc()* definition to perform the memory allocation—and itself only embed the necessary logic to implement the statistics gathering function.

RATIONALE

The ISO C standard does not require that pointers to functions can be cast back and forth to pointers to data. However, POSIX-conforming implementations are required to support this, as noted in [Section 2.12.3](#) (on page 83). The result of converting a pointer to a function into a pointer to another data type (except `void *`) is still undefined, however.

Note that compilers conforming to the ISO C standard are required to generate a warning if a conversion from a `void *` pointer to a function pointer is attempted as in:

```
fptr = (int (*)(int))dlsym(handle, "my_function");
```

FUTURE DIRECTIONS

None.

SEE ALSO

[*dlclose\(\)*](#), [*dlerror\(\)*](#), [*dlopen\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, [`<dlfcn.h>`](#)

CHANGE HISTORY

First released in Issue 5.

Issue 6

The `restrict` keyword is added to the *dlsym()* prototype for alignment with the ISO/IEC 9899:1999 standard.

The `RTLD_DEFAULT` and `RTLD_NEXT` flags are reserved for future use.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/14 is applied, correcting an example, and adding text to the RATIONALE describing issues related to conversion of pointers to functions and back again.

Issue 7

The *dlsym()* function is moved from the XSI option to the Base.

9124 **NAME**
9125 `dprintf` — print formatted output

9126 **SYNOPSIS**

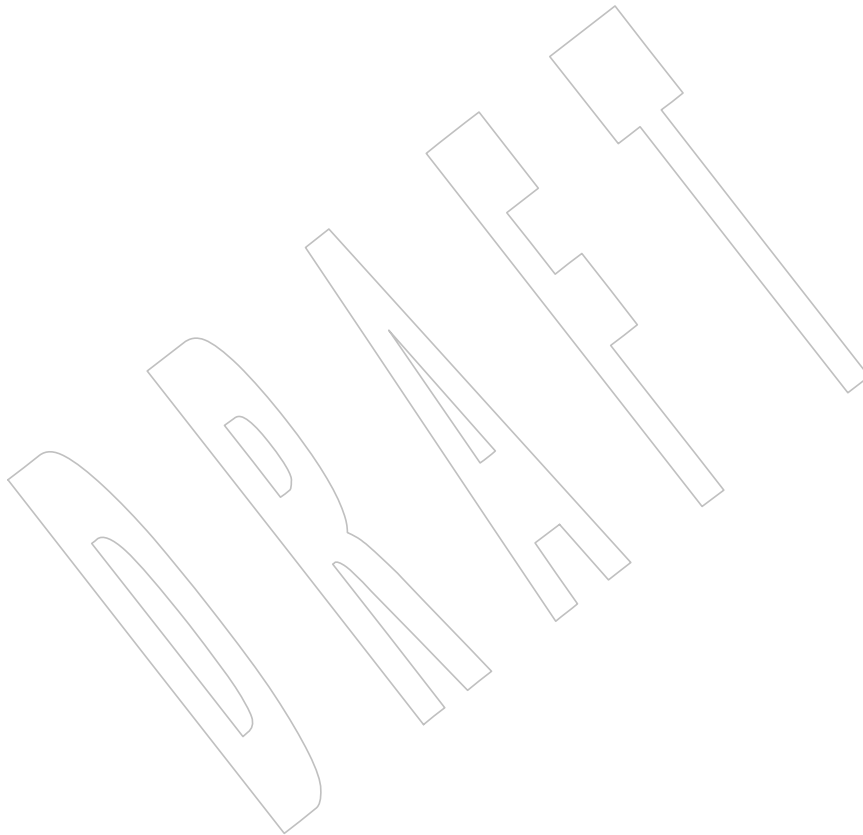
9127 CX `#include <stdio.h>`
9128 `int dprintf(int files, const char *format, ...);`

9129 **DESCRIPTION**

9130 Refer to *fprintf()*.

NAME

drand48, erand48, jrand48, lcong48, lrand48, mrand48, nrand48, seed48, srand48 — generate



9174 provide storage for the successive X_i values in the array specified as an argument when the
 9175 functions are invoked. That is why these routines do not have to be initialized; the calling
 9176 program merely has to place the desired initial value of X_i into the array and pass it as an
 9177 argument. By using different arguments, *erand48()*, *nrnd48()*, and *jrnd48()* allow separate
 9178 modules of a large program to generate several *independent* streams of pseudo-random numbers;
 9179 that is, the sequence of numbers in each stream shall *not* depend upon how many times the
 9180 routines are called to generate numbers for the other streams.

9181 The initializer function *srand48()* sets the high-order 32 bits of X_i to the low-order 32 bits
 9182 contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

9183 The initializer function *seed48()* sets the value of X_i to the 48-bit value specified in the argument
 9184 array. The low-order 16 bits of X_i are set to the low-order 16 bits of *seed16v*[0]. The mid-order 16
 9185 bits of X_i are set to the low-order 16 bits of *seed16v*[1]. The high-order 16 bits of X_i are set to the
 9186 low-order 16 bits of *seed16v*[2]. In addition, the previous value of X_i is copied into a 48-bit
 9187 internal buffer, used only by *seed48()*, and a pointer to this buffer is the value returned by
 9188 *seed48()*. This returned pointer, which can just be ignored if not needed, is useful if a program is
 9189 to be restarted from a given point at some future time—use the pointer to get at and store the
 9190 last X_i value, and then use this value to reinitialize via *seed48()* when the program is restarted.

9191 The initializer function *lcong48()* allows the user to specify the initial X_i , the multiplier value a ,
 9192 and the addend value c . Argument array elements *param*[0-2] specify X_i , *param*[3-5] specify the
 9193 multiplier a , and *param*[6] specifies the 16-bit addend c . After *lcong48()* is called, a subsequent
 9194 call to either *srand48()* or *seed48()* shall restore the standard multiplier and addend values, a and
 9195 c , specified above.

9196 The *drand48()*, *lrnd48()*, and *mrnd48()* function need not be thread-safe. A function that is not
 9197 required to be thread-safe is not required to be reentrant.

9198 RETURN VALUE

9199 As described in the DESCRIPTION above.

9200 ERRORS

9201 No errors are defined.

9202 EXAMPLES

9203 None.

9204 APPLICATION USAGE

9205 None.

9206 RATIONALE

9207 None.

9208 FUTURE DIRECTIONS

9209 None.

9210 SEE ALSO

9211 *rand()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdlib.h>

9212 CHANGE HISTORY

9213 First released in Issue 1. Derived from Issue 1 of the SVID.

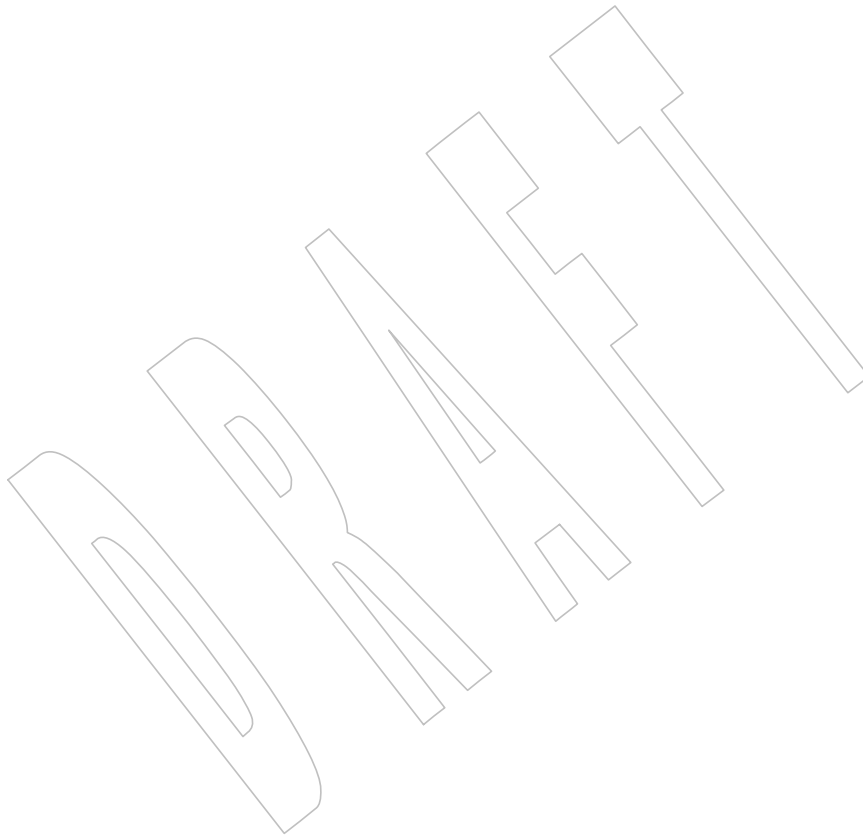
9214 Issue 5

9215 A note indicating that the *drand48()*, *lrnd48()*, and *mrnd48()* functions need not be reentrant is
 9216 added to the DESCRIPTION.

9217
9218

Issue 6

The normative text is updated to avoid use of the term “must” for application requirements.



9219 **NAME**
 9220 dup, dup2 — duplicate an open file descriptor

9221 **SYNOPSIS**
 9222 #include <unistd.h>
 9223 int dup(int *filde*s);
 9224 int dup2(int *filde*s, int *filde*s2);

9225 **DESCRIPTION**
 9226 The *dup()* and *dup2()* functions provide an alternative interface to the service provided by
 9227 *fcntl()* using the F_DUPFD command. The call:

9228 *fid* = dup(*filde*s);

9229 shall be equivalent to:

9230 *fid* = fcntl(*filde*s, F_DUPFD, 0);

9231 The call:

9232 *fid* = dup2(*filde*s, *filde*s2);

9233 shall be equivalent to:

9234 close(*filde*s2);

9235 *fid* = fcntl(*filde*s, F_DUPFD, *filde*s2);

9236 except for the following:

- 9237 • If *filde*s2 is less than 0 or greater than or equal to {OPEN_MAX}, *dup2()* shall return -1 with
 9238 *errno* set to [EBADF].
- 9239 • If *filde*s is a valid file descriptor and is equal to *filde*s2, *dup2()* shall return *filde*s2 without
 9240 closing it.
- 9241 • If *filde*s is not a valid file descriptor, *dup2()* shall return -1 and shall not close *filde*s2.
- 9242 • The value returned shall be equal to the value of *filde*s2 upon successful completion, or -1
 9243 upon failure.

9244 **RETURN VALUE**
 9245 Upon successful completion a non-negative integer, namely the file descriptor, shall be returned;
 9246 otherwise, -1 shall be returned and *errno* set to indicate the error.

9247 **ERRORS**

9248 The *dup()* function shall fail if:

9249 [EBADF] The *filde*s argument is not a valid open file descriptor.

9250 [EMFILE] All file descriptors available to the process are currently open.

9251 The *dup2()* function shall fail if:

9252 [EBADF] The *filde*s argument is not a valid open file descriptor or the argument *filde*s2 is
 9253 negative or greater than or equal to {OPEN_MAX}.

9254 [EINTR] The *dup2()* function was interrupted by a signal.

EXAMPLES**Redirecting Standard Output to a File**

The following example closes standard output for the current processes, re-assigns standard output to go to the file referenced by *pfid*, and closes the original file descriptor to clean up.

```
#include <unistd.h>
...
int pfd;
...
close(1);
dup(pfd);
close(pfd);
...
```

Redirecting Error Messages

The following example redirects messages from *stderr* to *stdout*.

```
#include <unistd.h>
...
dup2(1, 2);
...
```

APPLICATION USAGE

None.

RATIONALE

The *dup()* and *dup2()* functions are redundant. Their services are also provided by the *fcntl()* function. They have been included in this volume of IEEE Std 1003.1-200x primarily for historical reasons, since many existing applications use them.

While the brief code segment shown is very similar in behavior to *dup2()*, a conforming implementation based on other functions defined in this volume of IEEE Std 1003.1-200x is significantly more complex. Least obvious is the possible effect of a signal-catching function that could be invoked between steps and allocate or deallocate file descriptors. This could be avoided by blocking signals.

The *dup2()* function is not marked obsolescent because it presents a type-safe version of functionality provided in a type-unsafe version by *fcntl()*. It is used in the POSIX Ada binding.

The *dup2()* function is not intended for use in critical regions as a synchronization mechanism.

In the description of [EBADF], the case of *fildest* being out of range is covered by the given case of *fildest* not being valid. The descriptions for *fildest* and *fildest2* are different because the only kind of invalidity that is relevant for *fildest2* is whether it is out of range; that is, it does not matter whether *fildest2* refers to an open file when the *dup2()* call is made.

FUTURE DIRECTIONS

None.

SEE ALSO

close(), *fcntl()*, *open()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

9297 **NAME**

9298 duplocale — duplicate a locale object

9299 **SYNOPSIS**

```
9300 CX #include <locale.h>
9301 locale_t duplocale(locale_t locobj);
```

9302 **DESCRIPTION**

9303 The *duplocale()* function shall create a duplicate copy of the locale object referenced by the *locobj*
 9304 argument.

9305 **RETURN VALUE**

9306 Upon successful completion, the *duplocale()* function shall return a handle for a new locale
 9307 object. Otherwise, *duplocale()* shall return (**locale_t**)0 and set *errno* to indicate the error.

9308 **ERRORS**

9309 The *duplocale()* function shall fail if:

9310 [ENOMEM] There is not enough memory available to create the locale object or load the
 9311 locale data.

9312 The *duplocale()* function may fail if:

9313 [EINVAL] *locobj* is not a handle for a locale object.

9314 **EXAMPLES**9315 **Constructing an Altered Version of an Existing Locale Object**

9316 The following example shows a code fragment to create a slightly altered version of an existing
 9317 locale object. The function takes a locale object and a locale name and it replaces the *LC_TIME*
 9318 category data in the locale object with that from the named locale.

```
9319 #include <locale.h>
9320 ...
9321 locale_t
9322 with_changed_lc_time (locale_t obj, const char *name)
9323 {
9324     locale_t retval = duplocale (obj);
9325     if (retval != (locale_t) 0)
9326     {
9327         locale_t changed = newlocale (LC_TIME_MASK, name, retval);
9328         if (changed == (locale_t) 0)
9329             /* An error occurred. Free all allocated resources. */
9330             freelocale (retval);
9331         retval = changed;
9332     }
9333     return retval; }
9334 }
```

9335 **APPLICATION USAGE**

9336 The use of the *duplocale()* function is recommended for situations where a locale object is being
 9337 used in multiple places, and it is possible that the lifetime of the locale object might end before
 9338 all uses are finished. Another reason to duplicate a locale object is if a slightly modified form is
 9339 needed. This can be achieved by a call to *newlocale()* following the *duplocale()* call.

duplocale()

9340 As with the *newlocale()* function, handles for locale objects created by the *duplocale()* function
9341 should be released by a corresponding call to *freelocale()*.

RATIONALE

9342 None.
9343

FUTURE DIRECTIONS

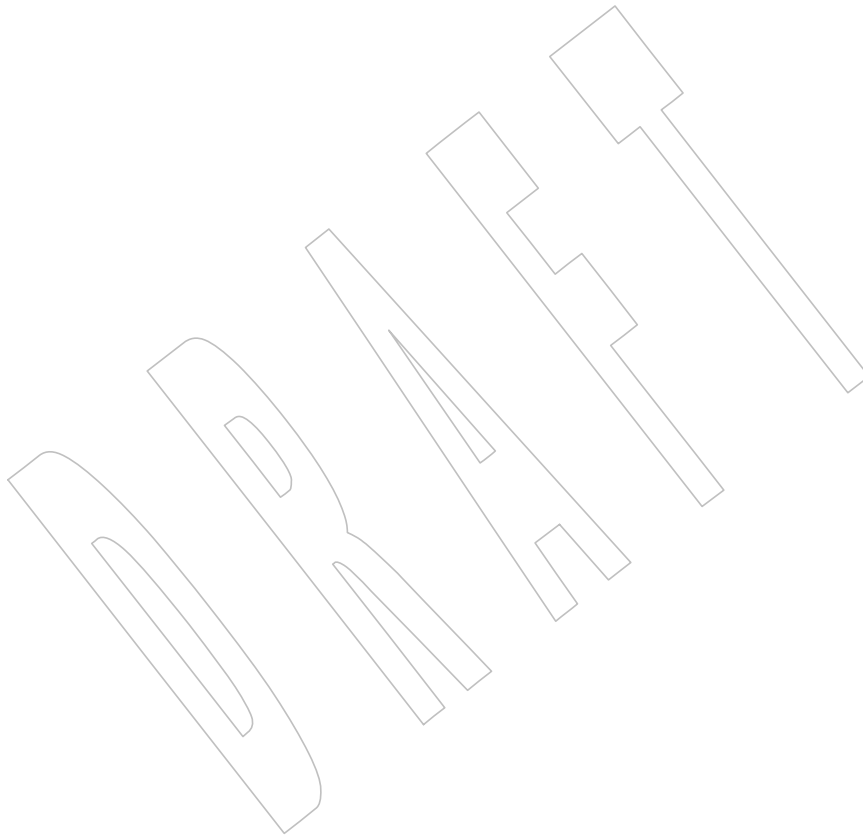
9344 None.
9345

SEE ALSO

9346 *freelocale()*, *newlocale()*, *uselocale()*, the Base Definitions volume of IEEE Std 1003.1-200x,
9347 **<locale.h>**
9348

CHANGE HISTORY

9349 First released in Issue 7.
9350



NAME

encrypt — encoding function (**CRYPT**)

SYNOPSIS

```
XSI #include <unistd.h>
void encrypt(char block[64], int edflag);
```

DESCRIPTION

The *encrypt()* function shall provide access to an implementation-defined encoding algorithm. The key generated by *setkey()* is used to encrypt the string *block* with *encrypt()*.

The *block* argument to *encrypt()* shall be an array of length 64 bytes containing only the bytes with values of 0 and 1. The array is modified in place to a similar array using the key set by *setkey()*. If *edflag* is 0, the argument is encoded. If *edflag* is 1, the argument may be decoded (see the APPLICATION USAGE section); if the argument is not decoded, *errno* shall be set to [ENOSYS].

The *encrypt()* function shall not change the setting of *errno* if successful. An application wishing to check for error situations should set *errno* to 0 before calling *encrypt()*. If *errno* is non-zero on return, an error has occurred.

The *encrypt()* function need not be thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

RETURN VALUE

The *encrypt()* function shall not return a value.

ERRORS

The *encrypt()* function shall fail if:

[ENOSYS] The functionality is not supported on this implementation.

EXAMPLES

None.

APPLICATION USAGE

Historical implementations of the *encrypt()* function used a rather primitive encoding algorithm.

In some environments, decoding might not be implemented. This is related to some Government restrictions on encryption and decryption routines. Historical practice has been to ship a different version of the encryption library without the decryption feature in the routines supplied. Thus the exported version of *encrypt()* does encoding but not decoding.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

crypt(), *setkey()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

encrypt()

9390

Issue 5

9391

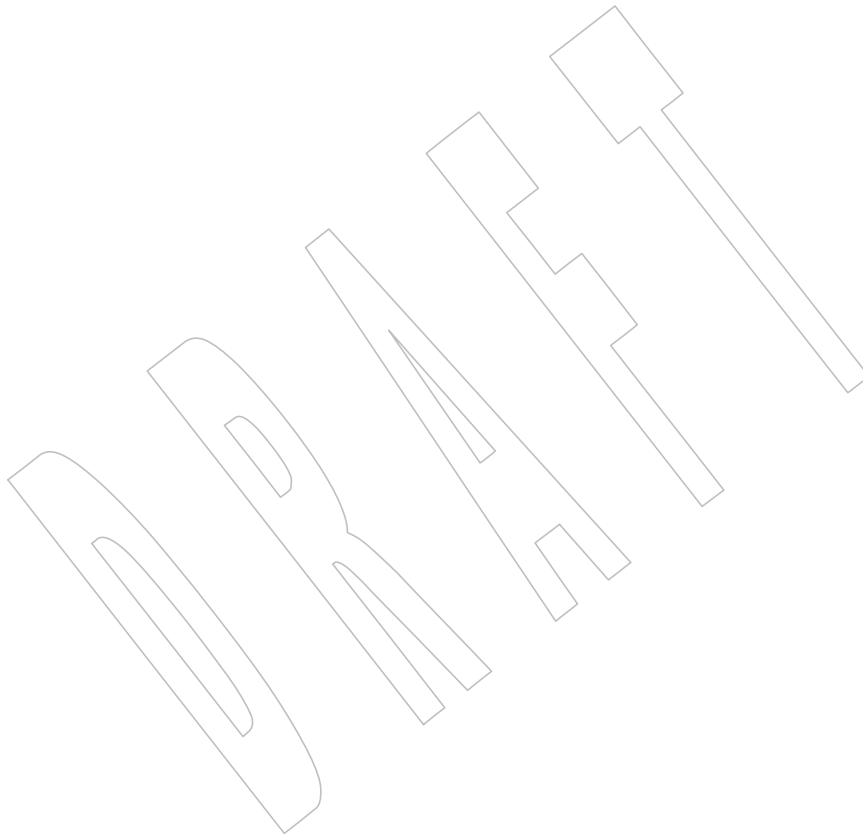
A note indicating that this function need not be reentrant is added to the DESCRIPTION.

9392

Issue 6

9393

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.



9394 **NAME**
 9395 endgrent, getgrent, setgrent — group database entry functions

9396 **SYNOPSIS**

```
9397 XSI #include <grp.h>
9398 void endgrent(void);
9399 struct group *getgrent(void);
9400 void setgrent(void);
```

9401 **DESCRIPTION**

9402 The *getgrent()* function shall return a pointer to a structure containing the broken-out fields of an
 9403 entry in the group database. When first called, *getgrent()* shall return a pointer to a **group**
 9404 structure containing the first entry in the group database. Thereafter, it shall return a pointer to a
 9405 **group** structure containing the next group structure in the group database, so successive calls
 9406 may be used to search the entire database.

9407 An implementation that provides extended security controls may impose further
 9408 implementation-defined restrictions on accessing the group database. In particular, the system
 9409 may deny the existence of some or all of the group database entries associated with groups other
 9410 than those groups associated with the caller and may omit users other than the caller from the
 9411 list of members of groups in database entries that are returned.

9412 The *setgrent()* function shall rewind the group database to allow repeated searches.

9413 The *endgrent()* function may be called to close the group database when processing is complete.

9414 These functions need not be thread-safe. A function that is not required to be thread-safe is not
 9415 required to be reentrant.

9416 **RETURN VALUE**

9417 When first called, *getgrent()* shall return a pointer to the first group structure in the group
 9418 database. Upon subsequent calls it shall return the next group structure in the group database.
 9419 The *getgrent()* function shall return a null pointer on end-of-file or an error and *errno* may be set
 9420 to indicate the error.

9421 The return value may point to a static area which is overwritten by a subsequent call to
 9422 *getgrgid()*, *getgrnam()*, or *getgrent()*.

9423 **ERRORS**

9424 The *getgrent()* function may fail if:

- | | | |
|------|----------|--|
| 9425 | [EINTR] | A signal was caught during the operation. |
| 9426 | [EIO] | An I/O error has occurred. |
| 9427 | [EMFILE] | All file descriptors available to the process are currently open. |
| 9428 | [ENFILE] | The maximum allowable number of files is currently open in the system. |

9429
9430
9431
9432
9433
9434
9435
9436
9437
9438
9439
9440
9441
9442
9443
9444
9445
9446
9447
9448
9449
9450
9451
9452
9453**EXAMPLES**

None.

APPLICATION USAGE

These functions are provided due to their historical usage. Applications should avoid dependencies on fields in the group database, whether the database is a single file, or where in the file system name space the database resides. Applications should use *getgrnam()* and *getgrgid()* whenever possible because it avoids these dependencies.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getgrgid(), *getgrnam()*, *getlogin()*, *getpwent()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<grp.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

Issue 6

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

9454 **NAME**
 9455 `endhostent`, `gethostent`, `sethostent` — network host database functions

9456 **SYNOPSIS**
 9457 `#include <netdb.h>`
 9458 `void endhostent(void);`
 9459 `struct hostent *gethostent(void);`
 9460 `void sethostent(int stayopen);`

9461 **DESCRIPTION**
 9462 These functions shall retrieve information about hosts. This information is considered to be
 9463 stored in a database that can be accessed sequentially or randomly. The implementation of this
 9464 database is unspecified.

9465 **Note:** In many cases this database is implemented by the Domain Name System, as documented in
 9466 RFC 1034, RFC 1035, and RFC 1886.

9467 The `sethostent()` function shall open a connection to the database and set the next entry for
 9468 retrieval to the first entry in the database. If the `stayopen` argument is non-zero, the connection
 9469 shall not be closed by a call to `gethostent()`, and the implementation may maintain an open file
 9470 descriptor.

9471 The `gethostent()` function shall read the next entry in the database, opening and closing a
 9472 connection to the database as necessary.

9473 Entries shall be returned in **hostent** structures.

9474 The `endhostent()` function shall close the connection to the database, releasing any open file
 9475 descriptor.

9476 These functions need not be thread-safe. A function that is not required to be thread-safe is not
 9477 required to be reentrant.

9478 **RETURN VALUE**
 9479 Upon successful completion, the `gethostent()` function shall return a pointer to a **hostent**
 9480 structure if the requested entry was found, and a null pointer if the end of the database was
 9481 reached or the requested entry was not found.

9482 **ERRORS**
 9483 No errors are defined for `endhostent()`, `gethostent()`, and `sethostent()`.

9484 **EXAMPLES**
 9485 None.

9486 **APPLICATION USAGE**
 9487 The `gethostent()` function may return pointers to static data, which may be overwritten by
 9488 subsequent calls to any of these functions.

9489 **RATIONALE**
 9490 None.

9491 **FUTURE DIRECTIONS**
 9492 None.

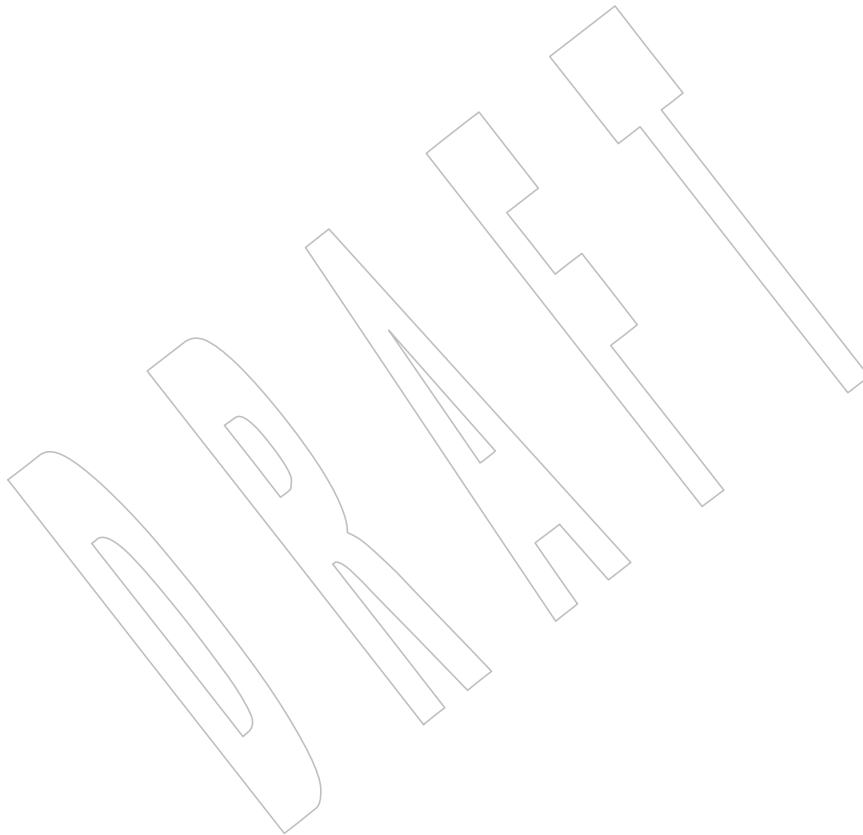
9493 **SEE ALSO**
 9494 `endservent()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<netdb.h>`

endhostent()

9495
9496

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.



9497 **NAME**
 9498 endnetent, getnetbyaddr, getnetbyname, getnetent, setnetent — network database functions

9499 **SYNOPSIS**
 9500 #include <netdb.h>
 9501 void endnetent(void);
 9502 struct netent *getnetbyaddr(uint32_t net, int type);
 9503 struct netent *getnetbyname(const char *name);
 9504 struct netent *getnetent(void);
 9505 void setnetent(int stayopen);

9506 **DESCRIPTION**
 9507 These functions shall retrieve information about networks. This information is considered to be
 9508 stored in a database that can be accessed sequentially or randomly. The implementation of this
 9509 database is unspecified.

9510 The *setnetent()* function shall open and rewind the database. If the *stayopen* argument is non-
 9511 zero, the connection to the *net* database shall not be closed after each call to *getnetent()* (either
 9512 directly, or indirectly through one of the other *getnet**() functions), and the implementation may
 9513 maintain an open file descriptor to the database.

9514 The *getnetent()* function shall read the next entry of the database, opening and closing a
 9515 connection to the database as necessary.

9516 The *getnetbyaddr()* function shall search the database from the beginning, and find the first entry
 9517 for which the address family specified by *type* matches the *n_addrtype* member and the network
 9518 number *net* matches the *n_net* member, opening and closing a connection to the database as
 9519 necessary. The *net* argument shall be the network number in host byte order.

9520 The *getnetbyname()* function shall search the database from the beginning and find the first entry
 9521 for which the network name specified by *name* matches the *n_name* member, opening and
 9522 closing a connection to the database as necessary.

9523 The *getnetbyaddr()*, *getnetbyname()*, and *getnetent()* functions shall each return a pointer to a
 9524 **netent** structure, the members of which shall contain the fields of an entry in the network
 9525 database.

9526 The *endnetent()* function shall close the database, releasing any open file descriptor.

9527 These functions need not be thread-safe. A function that is not required to be thread-safe is not
 9528 required to be reentrant.

9529 **RETURN VALUE**
 9530 Upon successful completion, *getnetbyaddr()*, *getnetbyname()*, and *getnetent()* shall return a
 9531 pointer to a **netent** structure if the requested entry was found, and a null pointer if the end of the
 9532 database was reached or the requested entry was not found. Otherwise, a null pointer shall be
 9533 returned.

9534 **ERRORS**
 9535 No errors are defined.

endnetent()

9536

EXAMPLES

9537

None.

9538

APPLICATION USAGE

9539

The *getnetbyaddr()*, *getnetbyname()*, and *getnetent()* functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

9540

9541

RATIONALE

9542

None.

9543

FUTURE DIRECTIONS

9544

None.

9545

SEE ALSO

9546

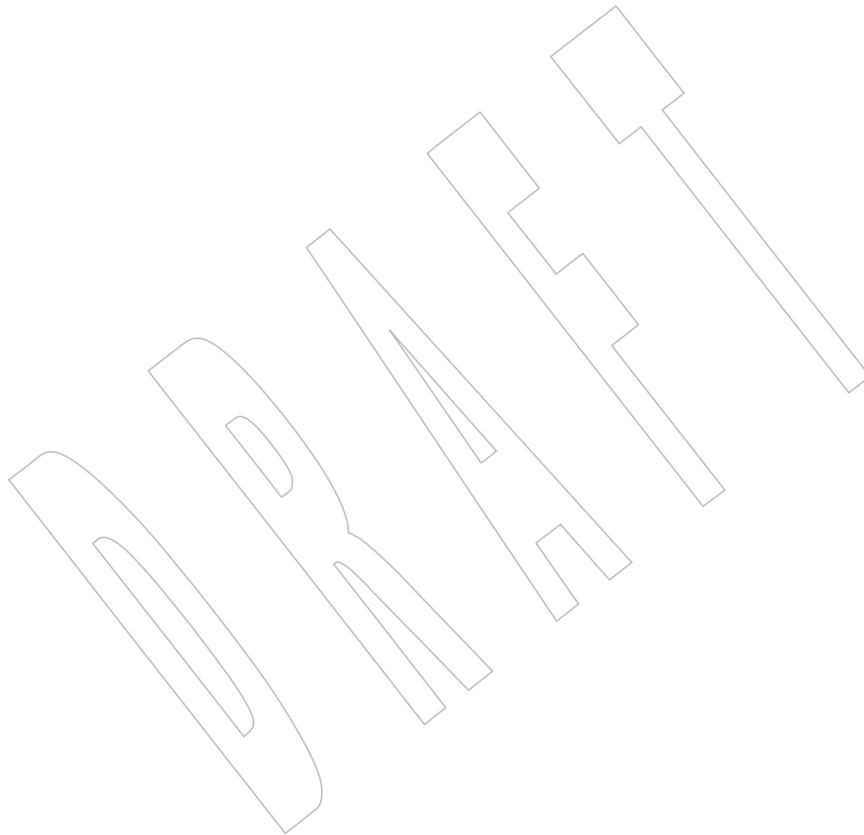
The Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**

9547

CHANGE HISTORY

9548

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.



9549 **NAME**

9550 endprotoent, getprotobyname, getprotobynumber, getprotoent, setprotoent — network protocol
9551 database functions

9552 **SYNOPSIS**

```
9553 #include <netdb.h>

9554 void endprotoent(void);
9555 struct protoent *getprotobyname(const char *name);
9556 struct protoent *getprotobynumber(int proto);
9557 struct protoent *getprotoent(void);
9558 void setprotoent(int stayopen);
```

9559 **DESCRIPTION**

9560 These functions shall retrieve information about protocols. This information is considered to be
9561 stored in a database that can be accessed sequentially or randomly. The implementation of this
9562 database is unspecified.

9563 The *setprotoent()* function shall open a connection to the database, and set the next entry to the
9564 first entry. If the *stayopen* argument is non-zero, the connection to the network protocol database
9565 shall not be closed after each call to *getprotoent()* (either directly, or indirectly through one of the
9566 other *getproto**() functions), and the implementation may maintain an open file descriptor for
9567 the database.

9568 The *getprotobyname()* function shall search the database from the beginning and find the first
9569 entry for which the protocol name specified by *name* matches the *p_name* member, opening and
9570 closing a connection to the database as necessary.

9571 The *getprotobynumber()* function shall search the database from the beginning and find the first
9572 entry for which the protocol number specified by *proto* matches the *p_proto* member, opening
9573 and closing a connection to the database as necessary.

9574 The *getprotoent()* function shall read the next entry of the database, opening and closing a
9575 connection to the database as necessary.

9576 The *getprotobyname()*, *getprotobynumber()*, and *getprotoent()* functions shall each return a pointer
9577 to a **protoent** structure, the members of which shall contain the fields of an entry in the network
9578 protocol database.

9579 The *endprotoent()* function shall close the connection to the database, releasing any open file
9580 descriptor.

9581 These functions need not be thread-safe. A function that is not required to be thread-safe is not
9582 required to be reentrant.

9583 **RETURN VALUE**

9584 Upon successful completion, *getprotobyname()*, *getprotobynumber()*, and *getprotoent()* return a
9585 pointer to a **protoent** structure if the requested entry was found, and a null pointer if the end of
9586 the database was reached or the requested entry was not found. Otherwise, a null pointer is
9587 returned.

9588 **ERRORS**

9589 No errors are defined.

endprotoent()

9590

EXAMPLES

9591

None.

9592

APPLICATION USAGE

9593

The *getprotobyname()*, *getprotobynumber()*, and *getprotoent()* functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

9594

9595

RATIONALE

9596

None.

9597

FUTURE DIRECTIONS

9598

None.

9599

SEE ALSO

9600

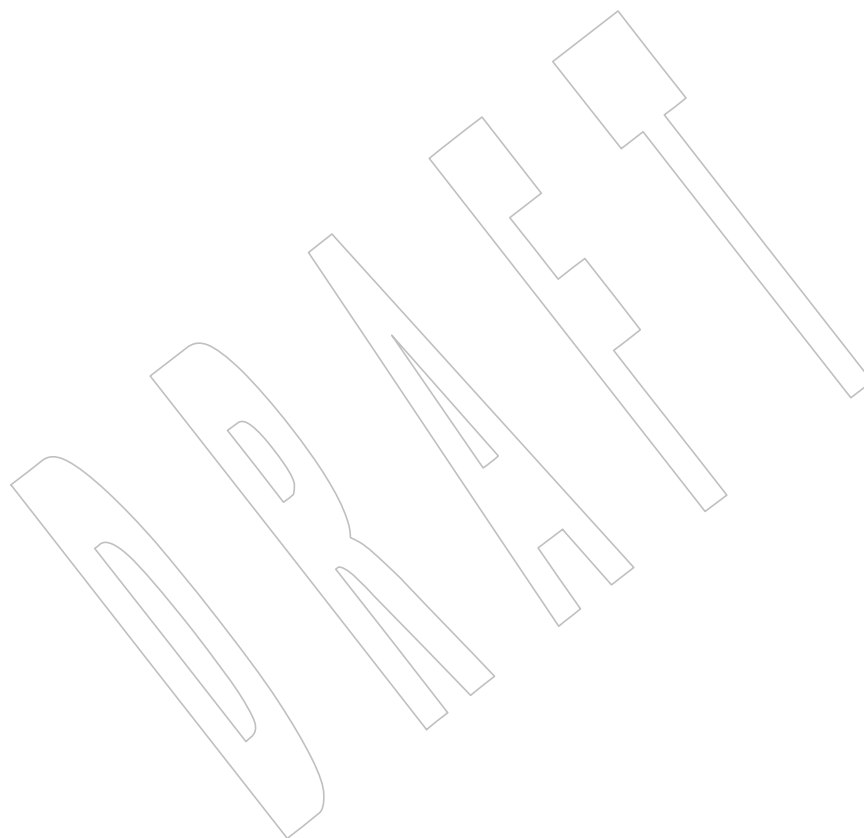
The Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**

9601

CHANGE HISTORY

9602

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.



9603 **NAME**
 9604 `endpwent`, `getpwent`, `setpwent` — user database functions

9605 **SYNOPSIS**

```
9606 XSI      #include <pwd.h>
9607
9608 void endpwent(void);
9609 struct passwd *getpwent(void);
9610 void setpwent(void);
```

9610 **DESCRIPTION**

9611 These functions shall retrieve information about users.

9612 The `getpwent()` function shall return a pointer to a structure containing the broken-out fields of
 9613 an entry in the user database. Each entry in the user database contains a **passwd** structure. When
 9614 first called, `getpwent()` shall return a pointer to a **passwd** structure containing the first entry in
 9615 the user database. Thereafter, it shall return a pointer to a **passwd** structure containing the next
 9616 entry in the user database. Successive calls can be used to search the entire user database.

9617 If an end-of-file or an error is encountered on reading, `getpwent()` shall return a null pointer.

9618 An implementation that provides extended security controls may impose further
 9619 implementation-defined restrictions on accessing the user database. In particular, the system
 9620 may deny the existence of some or all of the user database entries associated with users other
 9621 than the caller.

9622 The `setpwent()` function effectively rewinds the user database to allow repeated searches.

9623 The `endpwent()` function may be called to close the user database when processing is complete.

9624 These functions need not be thread-safe. A function that is not required to be thread-safe is not
 9625 required to be reentrant.

9626 **RETURN VALUE**

9627 The `getpwent()` function shall return a null pointer on end-of-file or error.

9628 **ERRORS**

9629 The `getpwent()`, `setpwent()`, and `endpwent()` functions may fail if:

9630 [EIO] An I/O error has occurred.

9631 In addition, `getpwent()` and `setpwent()` may fail if:

9632 [EMFILE] All file descriptors available to the process are currently open.

9633 [ENFILE] The maximum allowable number of files is currently open in the system.

9634 The return value may point to a static area which is overwritten by a subsequent call to
 9635 `getpwuid()`, `getpwnam()`, or `getpwent()`.

9636 **EXAMPLES**9637 **Searching the User Database**

9638 The following example uses the *getpwent()* function to get successive entries in the user
 9639 database, returning a pointer to a **passwd** structure that contains information about each user.
 9640 The call to *endpwent()* closes the user database and cleans up.

```
9641 #include <pwd.h>
9642 ...
9643 struct passwd *p;
9644 ...
9645 while ((p = getpwent ()) != NULL) {
9646     ...
9647 }
9648 endpwent();
9649 ...
```

9650 **APPLICATION USAGE**

9651 These functions are provided due to their historical usage. Applications should avoid
 9652 dependencies on fields in the password database, whether the database is a single file, or where
 9653 in the file system name space the database resides. Applications should use *getpwuid()*
 9654 whenever possible because it avoids these dependencies.

9655 **RATIONALE**

9656 None.

9657 **FUTURE DIRECTIONS**

9658 None.

9659 **SEE ALSO**

9660 *endgrent()*, *getlogin()*, *getpwnam()*, *getpwuid()*, the Base Definitions volume of
 9661 IEEE Std 1003.1-200x, **<pwd.h>**

9662 **CHANGE HISTORY**

9663 First released in Issue 4, Version 2.

9664 **Issue 5**

9665 Moved from X/OPEN UNIX extension to BASE.

9666 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
 9667 VALUE section.

9668 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

9669 **Issue 6**

9670 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

9671 **Issue 7**

9672 SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

9673 **NAME**

9674 endservent, getservbyname, getservbyport, getservent, setservent — network services database
 9675 functions

9676 **SYNOPSIS**

```
9677 #include <netdb.h>
9678
9679 void endservent(void);
9680 struct servent *getservbyname(const char *name, const char *proto);
9681 struct servent *getservbyport(int port, const char *proto);
9682 struct servent *getservent(void);
9683 void setservent(int stayopen);
```

9683 **DESCRIPTION**

9684 These functions shall retrieve information about network services. This information is
 9685 considered to be stored in a database that can be accessed sequentially or randomly. The
 9686 implementation of this database is unspecified.

9687 The *setservent()* function shall open a connection to the database, and set the next entry to the
 9688 first entry. If the *stayopen* argument is non-zero, the *net* database shall not be closed after each
 9689 call to the *getservent()* function (either directly, or indirectly through one of the other *getserv**(
 9690 functions), and the implementation may maintain an open file descriptor for the database.

9691 The *getservent()* function shall read the next entry of the database, opening and closing a
 9692 connection to the database as necessary.

9693 The *getservbyname()* function shall search the database from the beginning and find the first
 9694 entry for which the service name specified by *name* matches the *s_name* member and the protocol
 9695 name specified by *proto* matches the *s_proto* member, opening and closing a connection to the
 9696 database as necessary. If *proto* is a null pointer, any value of the *s_proto* member shall be
 9697 matched.

9698 The *getservbyport()* function shall search the database from the beginning and find the first entry
 9699 for which the port specified by *port* matches the *s_port* member and the protocol name specified
 9700 by *proto* matches the *s_proto* member, opening and closing a connection to the database as
 9701 necessary. If *proto* is a null pointer, any value of the *s_proto* member shall be matched. The *port*
 9702 argument shall be a value obtained by converting a *uint16_t* in network byte order to *int*.

9703 The *getservbyname()*, *getservbyport()*, and *getservent()* functions shall each return a pointer to a
 9704 **servent** structure, the members of which shall contain the fields of an entry in the network
 9705 services database.

9706 The *endservent()* function shall close the database, releasing any open file descriptor.

9707 These functions need not be thread-safe. A function that is not required to be thread-safe is not
 9708 required to be reentrant.

9709 **RETURN VALUE**

9710 Upon successful completion, *getservbyname()*, *getservbyport()*, and *getservent()* return a pointer to
 9711 a **servent** structure if the requested entry was found, and a null pointer if the end of the database
 9712 was reached or the requested entry was not found. Otherwise, a null pointer is returned.

9713 **ERRORS**

9714 No errors are defined.

endservent()*System Interfaces***EXAMPLES**

None.

APPLICATION USAGE

The *port* argument of *getserbyport()* need not be compatible with the port values of all address families.

The *getserbyname()*, *getserbyport()*, and *getservent()* functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endhostent(), *endprotoent()*, *htonl()*, *inet_addr()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

Issue 7

SD5-XBD-ERN-14 is applied.

DRAFT

9733 **NAME**

9734 endutxent, getutxent, getutxid, getutxline, pututxline, setutxent — user accounting database
 9735 functions

9736 **SYNOPSIS**

```
9737 XSI #include <utmpx.h>
9738
9738 void endutxent(void);
9739 struct utmpx *getutxent(void);
9740 struct utmpx *getutxid(const struct utmpx *id);
9741 struct utmpx *getutxline(const struct utmpx *line);
9742 struct utmpx *pututxline(const struct utmpx *utmpx);
9743 void setutxent(void);
```

9744 **DESCRIPTION**

9745 These functions shall provide access to the user accounting database.

9746 The *getutxent()* function shall read the next entry from the user accounting database. If the
 9747 database is not already open, it shall open it. If it reaches the end of the database, it shall fail.

9748 The *getutxid()* function shall search forward from the current point in the database. If the
 9749 *ut_type* value of the **utmpx** structure pointed to by *id* is *BOOT_TIME*, *OLD_TIME*, or
 9750 *NEW_TIME*, then it shall stop when it finds an entry with a matching *ut_type* value. If the
 9751 *ut_type* value is *INIT_PROCESS*, *LOGIN_PROCESS*, *USER_PROCESS*, or *DEAD_PROCESS*,
 9752 then it shall stop when it finds an entry whose type is one of these four and whose *ut_id* member
 9753 matches the *ut_id* member of the **utmpx** structure pointed to by *id*. If the end of the database is
 9754 reached without a match, *getutxid()* shall fail.

9755 The *getutxline()* function shall search forward from the current point in the database until it
 9756 finds an entry of the type *LOGIN_PROCESS* or *USER_PROCESS* which also has a *ut_line* value
 9757 matching that in the **utmpx** structure pointed to by *line*. If the end of the database is reached
 9758 without a match, *getutxline()* shall fail.

9759 The *getutxid()* or *getutxline()* function may cache data. For this reason, to use *getutxline()* to
 9760 search for multiple occurrences, the application shall zero out the static data after each success,
 9761 or *getutxline()* may return a pointer to the same **utmpx** structure.

9762 There is one exception to the rule about clearing the structure before further reads are done. The
 9763 implicit read done by *pututxline()* (if it finds that it is not already at the correct place in the user
 9764 accounting database) shall not modify the static structure returned by *getutxent()*, *getutxid()*, or
 9765 *getutxline()*, if the application has modified this structure and passed the pointer back to
 9766 *pututxline()*.

9767 For all entries that match a request, the *ut_type* member indicates the type of the entry. Other
 9768 members of the entry shall contain meaningful data based on the value of the *ut_type* member as
 9769 follows:

ut_type Member	Other Members with Meaningful Data
EMPTY	No others
BOOT_TIME	<i>ut_tv</i>
OLD_TIME	<i>ut_tv</i>
NEW_TIME	<i>ut_tv</i>
USER_PROCESS	<i>ut_id, ut_user</i> (login name of the user), <i>ut_line, ut_pid, ut_tv</i>
INIT_PROCESS	<i>ut_id, ut_pid, ut_tv</i>
LOGIN_PROCESS	<i>ut_id, ut_user</i> (implementation-defined name of the login process), <i>ut_pid, ut_tv</i>
DEAD_PROCESS	<i>ut_id, ut_pid, ut_tv</i>

An implementation that provides extended security controls may impose implementation-defined restrictions on accessing the user accounting database. In particular, the system may deny the existence of some or all of the user accounting database entries associated with users other than the caller.

If the process has appropriate privileges, the *pututxline()* function shall write out the structure into the user accounting database. It shall use *getutxid()* to search for a record that satisfies the request. If this search succeeds, then the entry shall be replaced. Otherwise, a new entry shall be made at the end of the user accounting database.

The *endutxent()* function shall close the user accounting database.

The *setutxent()* function shall reset the input to the beginning of the database. This should be done before each search for a new entry if it is desired that the entire database be examined.

These functions need not be thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

RETURN VALUE

Upon successful completion, *getutxent()*, *getutxid()*, and *getutxline()* shall return a pointer to a **utmpx** structure containing a copy of the requested entry in the user accounting database. Otherwise, a null pointer shall be returned.

The return value may point to a static area which is overwritten by a subsequent call to *getutxid()* or *getutxline()*.

Upon successful completion, *pututxline()* shall return a pointer to a **utmpx** structure containing a copy of the entry added to the user accounting database. Otherwise, a null pointer shall be returned.

The *endutxent()* and *setutxent()* functions shall not return a value.

ERRORS

No errors are defined for the *endutxent()*, *getutxent()*, *getutxid()*, *getutxline()*, and *setutxent()* functions.

The *pututxline()* function may fail if:

[EPERM] The process does not have appropriate privileges.

9808
9809
9810
9811
9812
9813
9814
9815
9816
9817
9818
9819
9820
9821
9822
9823
9824
9825
9826

EXAMPLES

None.

APPLICATION USAGE

The sizes of the arrays in the structure can be found using the *sizeof* operator.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, `<utmpx.h>`

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

Issue 6

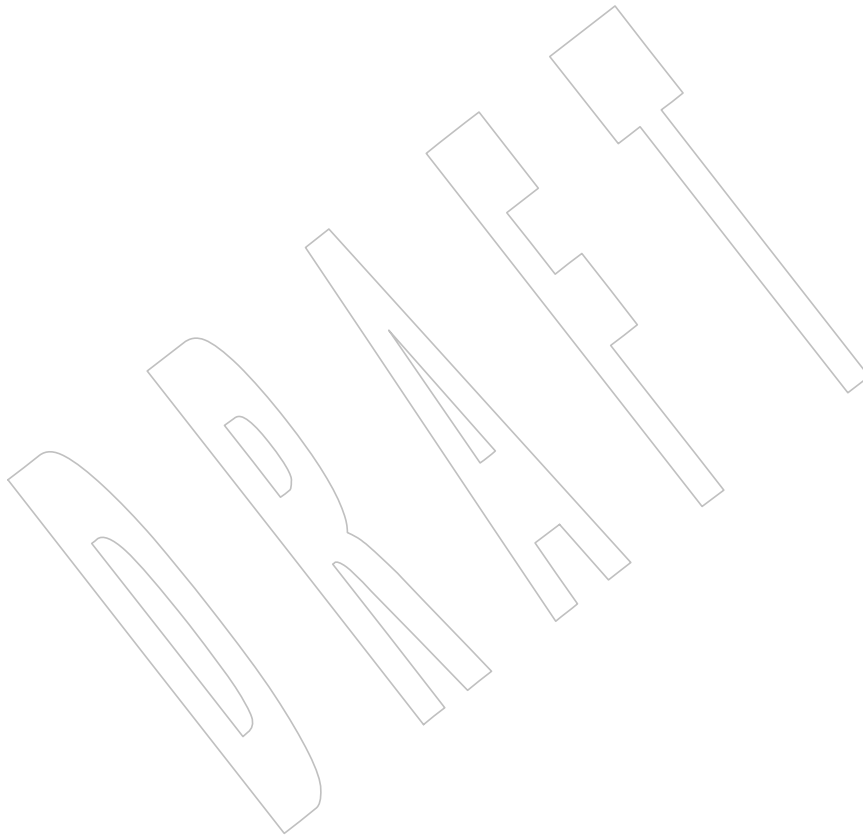
In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

DRAFT

9827 **NAME**
9828 environ — array of character pointers to the environment strings

9829 **SYNOPSIS**
9830 extern char **environ;

9831 **DESCRIPTION**
9832 Refer to the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables
9833 and *exec*.



9834 **NAME**
9835 erand48 — generate uniformly distributed pseudo-random numbers

9836 **SYNOPSIS**

```
9837 XSI #include <stdlib.h>  
9838 double erand48(unsigned short xsubi[3]);
```

9839 **DESCRIPTION**

9840 Refer to *drand48()*.

NAME

erf, erff, erfl — error functions

SYNOPSIS

```
#include <math.h>

double erf(double x);
float erff(float x);
long double erfl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the error function of their argument x , defined as:

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the value of the error function.

MX If x is NaN, a NaN shall be returned.

If x is ± 0 , ± 0 s

9882
9883
9884
9885
9886
9887
9888
9889
9890
9891
9892
9893
9894
9895
9896
9897
9898
9899
9900
9901
9902
9903
9904
9905
9906
9907
9908

}

APPLICATION USAGE

Underflow occurs when $|x| < \text{DBL_MIN} * (\text{sqrt}(\pi)/2)$.

On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

erfc(), *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *erf()* function is no longer marked as an extension.

The *erfc()* function is split out onto its own reference page.

The *erff()* and *erfl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/22 is applied, adding the example to the EXAMPLES section.

9909 **NAME**
 9910 `erfc`, `erfcf`, `erfcl` — complementary error functions

9911 **SYNOPSIS**
 9912 `#include <math.h>`
 9913 `double erfc(double x);`
 9914 `float erfcf(float x);`
 9915 `long double erfcl(long double x);`

9916 **DESCRIPTION**
 9917 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 9918 conflict between the requirements described here and the ISO C standard is unintentional. This
 9919 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

9920 These functions shall compute the complementary error function $1.0 - \text{erf}(x)$.

9921 An application wishing to check for error situations should set `errno` to zero and call
 9922 `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `errno` is non-zero or
 9923 `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-
 9924 zero, an error has occurred.

9925 **RETURN VALUE**
 9926 Upon successful completion, these functions shall return the value of the complementary error
 9927 function.

9928 If the correct value would cause underflow and is not representable, a range error may occur
 9929 MX and either 0.0 (if representable), or an implementation-defined value shall be returned.

9930 MX If x is NaN, a NaN shall be returned.

9931 If x is ± 0 , +1 shall be returned.

9932 If x is $-\text{Inf}$, +2 shall be returned.

9933 If x is $+\text{Inf}$, +0 shall be returned.

9934 If the correct value would cause underflow and is representable, a range error may occur and
 9935 the correct value shall be returned.

9936 **ERRORS**
 9937 These functions may fail if:

9938 Range Error The result underflows.

9939 If the integer expression `(math_errhandling & MATH_ERRNO)` is non-zero,
 9940 then `errno` shall be set to [ERANGE]. If the integer expression
 9941 `(math_errhandling & MATH_ERREXCEPT)` is non-zero, then the underflow
 9942 floating-point exception shall be raised.

9943 **EXAMPLES**
 9944 None.

9945 **APPLICATION USAGE**
 9946 The `erfc()` function is provided because of the extreme loss of relative accuracy if `erf(x)` is called
 9947 for large x and the result subtracted from 1.0.

9948 Note for IEEE Std 754-1985 **double**, $26.55 < x$ implies `erfc(x)` has underflowed.

9949 On error, the expressions `(math_errhandling & MATH_ERRNO)` and `(math_errhandling &`
 9950 `MATH_ERREXCEPT)` are independent of each other, but at least one of them must be non-zero.

9951
9952
9953
9954
9955
9956
9957
9958
9959
9960
9961
9962
9963
9964
9965
9966
9967
9968
9969

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

erf(), *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

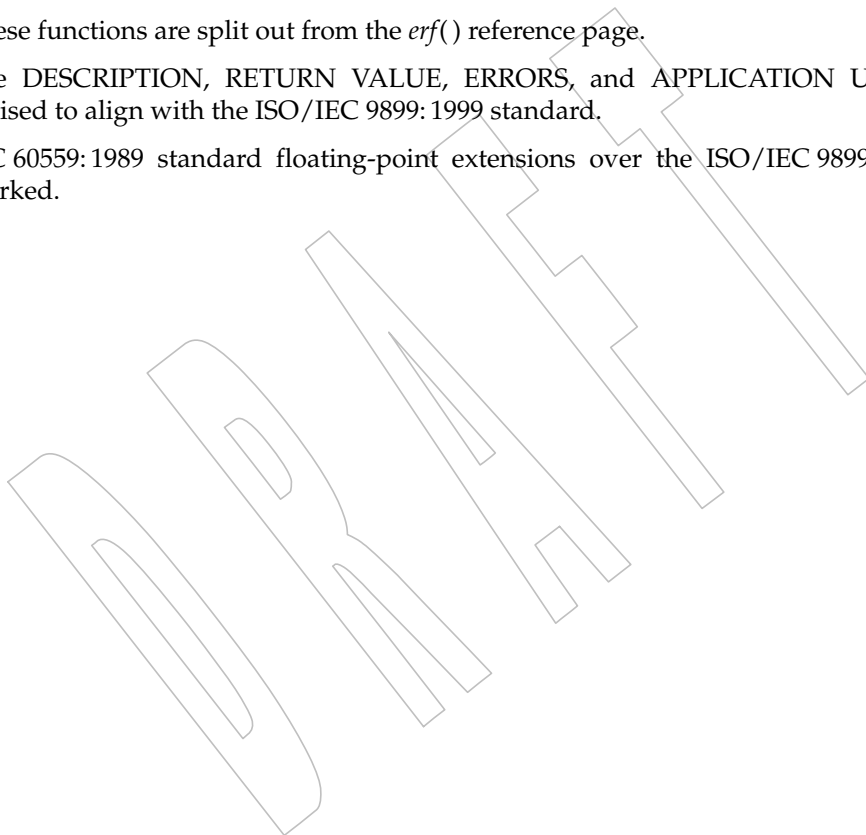
Issue 6

The *erfc()* function is no longer marked as an extension.

These functions are split out from the *erf()* reference page.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

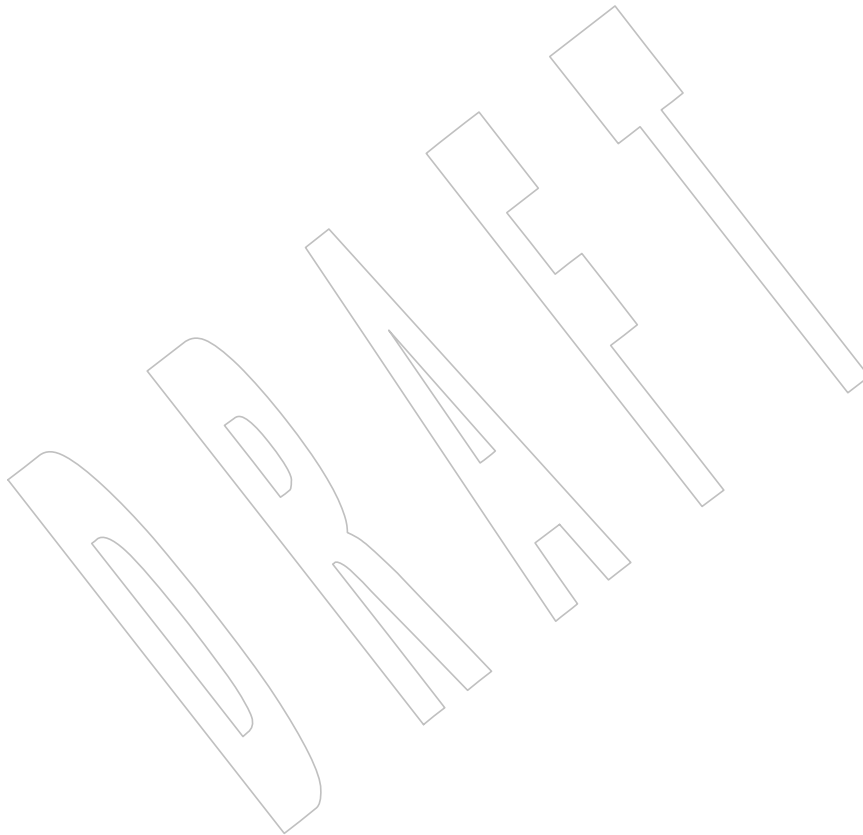
IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.



9970 **NAME**
9971 `erff, erfl` — error functions

9972 **SYNOPSIS**
9973 `#include <math.h>`
9974 `float erff(float x);`
9975 `long double erfl(long double x);`

9976 **DESCRIPTION**
9977 Refer to *erf()*.



9978 **NAME**9979 `errno` — error return value9980 **SYNOPSIS**9981 `#include <errno.h>`9982 **DESCRIPTION**9983 The lvalue *errno* is used by many functions to return error values.

9984 Many functions provide an error number in *errno*, which has type `int` and is defined in
 9985 `<errno.h>`. The value of *errno* shall be defined only after a call to a function for which it is
 9986 explicitly stated to be set and until it is changed by the next function call or if the application
 9987 assigns it a value. The value of *errno* should only be examined when it is indicated to be valid by
 9988 a function's return value. Applications shall obtain the definition of *errno* by the inclusion of
 9989 `<errno.h>`. No function in this volume of IEEE Std 1003.1-200x shall set *errno* to 0. The setting of
 9990 *errno* after a successful call to a function is unspecified unless the description of that function
 9991 specifies that *errno* shall not be modified.

9992 It is unspecified whether *errno* is a macro or an identifier declared with external linkage. If a
 9993 macro definition is suppressed in order to access an actual object, or a program defines an
 9994 identifier with the name *errno*, the behavior is undefined.

9995 The symbolic values stored in *errno* are documented in the ERRORS sections on all relevant
 9996 pages.

9997 **RETURN VALUE**

9998 None.

9999 **ERRORS**

10000 None.

10001 **EXAMPLES**

10002 None.

10003 **APPLICATION USAGE**

10004 Previously both POSIX and X/Open documents were more restrictive than the ISO C standard
 10005 in that they required *errno* to be defined as an external variable, whereas the ISO C standard
 10006 required only that *errno* be defined as a modifiable lvalue with type `int`.

10007 An application that needs to examine the value of *errno* to determine the error should set it to 0
 10008 before a function call, then inspect it before a subsequent function call.

10009 **RATIONALE**

10010 None.

10011 **FUTURE DIRECTIONS**

10012 None.

10013 **SEE ALSO**10014 [Section 2.3](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<errno.h>`10015 **CHANGE HISTORY**

10016 First released in Issue 1. Derived from Issue 1 of the SVID.

10017 **Issue 5**

10018 The following sentence is deleted from the DESCRIPTION: "The value of *errno* is 0 at program
 10019 start-up, but is never set to 0 by any XSI function". The DESCRIPTION also no longer states that
 10020 conforming implementations may support the declaration:

10021 `extern int errno;`

10022
10023
10024
10025
10026
10027
10028
10029
10030**Issue 6**

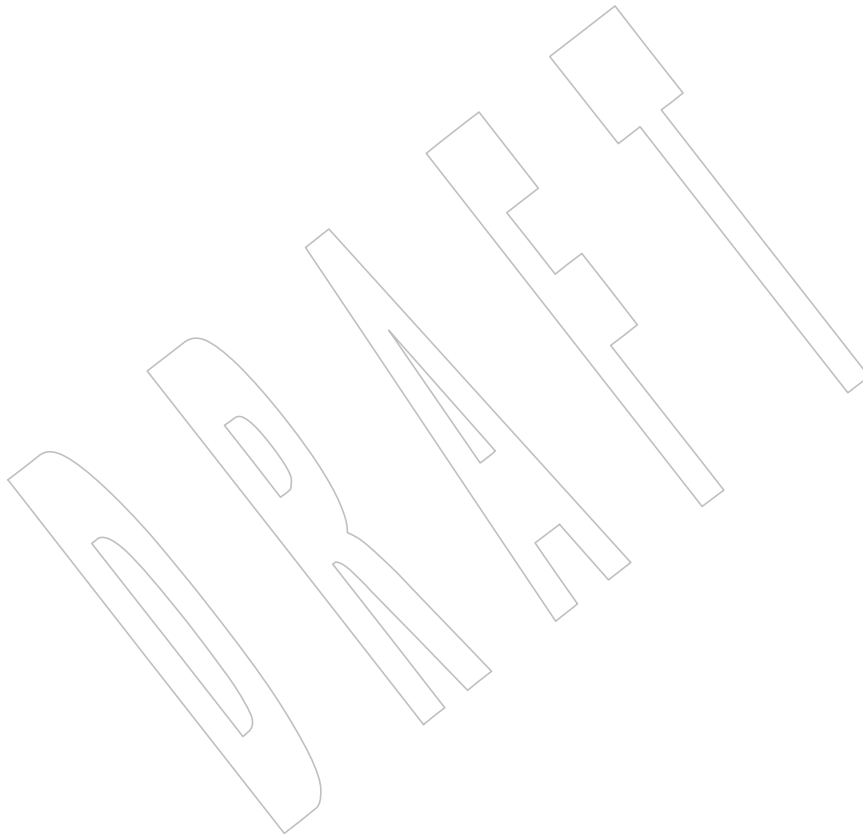
Obsolescent text regarding defining *errno* as:

```
extern int errno
```

is removed.

Text regarding no function setting *errno* to zero to indicate an error is changed to no function shall set *errno* to zero. This is for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/23 is applied, adding text to the DESCRIPTION stating that the setting of *errno* after a successful call to a function is unspecified unless the description of the function requires that it will not be modified.



10031 **NAME**

10032 environ, execl, execv, execl, execve, execlp, execvp, fexecve — execute a file

10033 **SYNOPSIS**

```

10034 #include <unistd.h>
10035
10036 extern char **environ;
10037 int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
10038 int execv(const char *path, char *const argv[]);
10039 int execlp(const char *path, const char *arg0, ... /*,
10040           (char *)0, char *const envp[] */);
10041 int execve(const char *path, char *const argv[], char *const envp[]);
10042 int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
10043 int execvp(const char *file, char *const argv[]);
10044 int fexecve(int fd, char *const argv[], char *const envp[]);

```

10044 **DESCRIPTION**

10045 The *exec* family of functions shall replace the current process image with a new process image.
 10046 The new image shall be constructed from a regular, executable file called the *new process image*
 10047 *file*. There shall be no return from a successful *exec*, because the calling process image is overlaid
 10048 by the new process image.

10049 The *fexecve()* function shall be equivalent to the *execve()* function except that the file to be
 10050 executed is determined by the file descriptor *fd* instead of a pathname. The file offset of *fd* is
 10051 ignored.

10052 When a C-language program is executed as a result of a call to one of the *exec* family of
 10053 functions, it shall be entered as a C-language function call as follows:

```
10054 int main (int argc, char *argv[]);
```

10055 where *argc* is the argument count and *argv* is an array of character pointers to the arguments
 10056 themselves. In addition, the following variable:

```
10057 extern char **environ;
```

10058 is initialized as a pointer to an array of character pointers to the environment strings. The *argv*
 10059 and *environ* arrays are each terminated by a null pointer. The null pointer terminating the *argv*
 10060 array is not counted in *argc*.

10061 Conforming multi-threaded applications shall not use the *environ* variable to access or modify
 10062 any environment variable while any other thread is concurrently modifying any environment
 10063 variable. A call to any function dependent on any environment variable shall be considered a
 10064 use of the *environ* variable to access that environment variable.

10065 The arguments specified by a program with one of the *exec* functions shall be passed on to the
 10066 new process image in the corresponding *main()* arguments.

10067 The argument *path* points to a pathname that identifies the new process image file.

10068 The argument *file* is used to construct a pathname that identifies the new process image file. If
 10069 the *file* argument contains a slash character, the *file* argument shall be used as the pathname for
 10070 this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed
 10071 as the environment variable *PATH* (see the Base Definitions volume of IEEE Std 1003.1-200x,
 10072 Chapter 8, Environment Variables). If this environment variable is not present, the results of the
 10073 search are implementation-defined.

10074 There are two distinct ways in which the contents of the process image file may cause the
 10075 execution to fail, distinguished by the setting of *errno* to either [ENOEXEC] or [EINVAL] (see the

10076 ERRORS section). In the cases where the other members of the *exec* family of functions would
 10077 fail and set *errno* to [ENOEXEC], the *execlp()* and *execvp()* functions shall execute a command
 10078 interpreter and the environment of the executed command shall be as if the process invoked the
 10079 *sh* utility using *execl()* as follows:

```
10080 execl(<shell path>, arg0, file, arg1, ..., (char *)0);
```

10081 where *<shell path>* is an unspecified pathname for the *sh* utility, *file* is the process image file, and
 10082 for *execvp()*, where *arg0*, *arg1*, and so on correspond to the values passed to *execvp()* in *argv*[0],
 10083 *argv*[1], and so on.

10084 The arguments represented by *arg0*,... are pointers to null-terminated character strings. These
 10085 strings shall constitute the argument list available to the new process image. The list is
 10086 terminated by a null pointer. The argument *arg0* should point to a filename that is associated
 10087 with the process being started by one of the *exec* functions.

10088 The argument *argv* is an array of character pointers to null-terminated strings. The application
 10089 shall ensure that the last member of this array is a null pointer. These strings shall constitute the
 10090 argument list available to the new process image. The value in *argv*[0] should point to a filename
 10091 that is associated with the process being started by one of the *exec* functions.

10092 The argument *envp* is an array of character pointers to null-terminated strings. These strings
 10093 shall constitute the environment for the new process image. The *envp* array is terminated by a
 10094 null pointer.

10095 For those forms not containing an *envp* pointer (*execl()*, *execv()*, *execlp()*, and *execvp()*), the
 10096 environment for the new process image shall be taken from the external variable *environ* in the
 10097 calling process.

10098 The number of bytes available for the new process' combined argument and environment lists is
 10099 {ARG_MAX}. It is implementation-defined whether null terminators, pointers, and/or any
 10100 alignment bytes are included in this total.

10101 File descriptors open in the calling process image shall remain open in the new process image,
 10102 except for those whose close-on-exec flag FD_CLOEXEC is set. For those file descriptors that
 10103 remain open, all attributes of the open file description remain unchanged. For any file descriptor
 10104 that is closed for this reason, file locks are removed as a result of the close as described in *close()*.
 10105 Locks that are not removed by closing of file descriptors remain unchanged.

10106 If file descriptors 0, 1, and 2 would otherwise be closed after a successful call to one of the *exec*
 10107 family of functions, and the new process image file has the set-user-ID or set-group-ID file mode
 10108 bits set, and the ST_NOSUID bit is not set for the file system containing the new process image
 10109 file, implementations may open an unspecified file for each of these file descriptors in the new
 10110 process image.

10111 Directory streams open in the calling process image shall be closed in the new process image.

10112 The state of the floating-point environment in the initial thread of the new process image shall
 10113 be set to the default.

10114 The state of conversion descriptors and message catalog descriptors in the new process image is
 10115 undefined.

10116 For the new process image, the equivalent of:

```
10117 setlocale(LC_ALL, "C")
```

10118 shall be executed at start-up.

10119 Signals set to the default action (SIG_DFL) in the calling process image shall be set to the default
 10120 action in the new process image. Except for SIGCHLD, signals set to be ignored (SIG_IGN) by
 10121 the calling process image shall be set to be ignored by the new process image. Signals set to be

- 10122 caught by the calling process image shall be set to the default action in the new process image
10123 (see <signal.h>).
- 10124 If the SIGCHLD signal is set to be ignored by the calling process image, it is unspecified whether
10125 the SIGCHLD signal is set to be ignored or to the default action in the new process image.
- 10126 XSI After a successful call to any of the *exec* functions, alternate signal stacks are not preserved and
10127 the SA_ONSTACK flag shall be cleared for all signals.
- 10128 After a successful call to any of the *exec* functions, any functions previously registered by the
10129 *atexit()* or *pthread_atfork()* functions are no longer registered.
- 10130 XSI If the ST_NOSUID bit is set for the file system containing the new process image file, then the
10131 effective user ID, effective group ID, saved set-user-ID, and saved set-group-ID are unchanged
10132 in the new process image. Otherwise, if the set-user-ID mode bit of the new process image file is
10133 set, the effective user ID of the new process image shall be set to the user ID of the new process
10134 image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the
10135 effective group ID of the new process image shall be set to the group ID of the new process
10136 image file. The real user ID, real group ID, and supplementary group IDs of the new process
10137 image shall remain the same as those of the calling process image. The effective user ID and
10138 effective group ID of the new process image shall be saved (as the saved set-user-ID and the
10139 saved set-group-ID) for use by *setuid()*.
- 10140 XSI Any shared memory segments attached to the calling process image shall not be attached to the
10141 new process image.
- 10142 Any named semaphores open in the calling process shall be closed as if by appropriate calls to
10143 *sem_close()*.
- 10144 TYM Any blocks of typed memory that were mapped in the calling process are unmapped, as if
10145 *munmap()* was implicitly called to unmap them.
- 10146 ML Memory locks established by the calling process via calls to *mlockall()* or *mlock()* shall be
10147 removed. If locked pages in the address space of the calling process are also mapped into the
10148 address spaces of other processes and are locked by those processes, the locks established by the
10149 other processes shall be unaffected by the call by this process to the *exec* function. If the *exec*
10150 function fails, the effect on memory locks is unspecified.
- 10151 Memory mappings created in the process are unmapped before the address space is rebuilt for
10152 the new process image.
- 10153 When the calling process image does not use the SCHED_FIFO, SCHED_RR, or
10154 SCHED_SPORADIC scheduling policies, the scheduling policy and parameters of the new
10155 process image and the initial thread in that new process image are implementation-defined.
- 10156 PS When the calling process image uses the SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC
10157 scheduling policies, the process policy and scheduling parameter settings shall not be changed
10158 by a call to an *exec* function. The initial thread in the new process image shall inherit the process
10159 scheduling policy and parameters. It shall have the default system contention scope, but shall
10160 inherit its allocation domain from the calling process image.
- 10161 Per-process timers created by the calling process shall be deleted before replacing the current
10162 process image with the new process image.
- 10163 MSG All open message queue descriptors in the calling process shall be closed, as described in
10164 *mq_close()*.
- 10165 Any outstanding asynchronous I/O operations may be canceled. Those asynchronous I/O
10166 operations that are not canceled shall complete as if the *exec* function had not yet occurred, but
10167 any associated signal notifications shall be suppressed. It is unspecified whether the *exec*
10168 function itself blocks awaiting such I/O completion. In no event, however, shall the new process

10169		image created by the <i>exec</i> function be affected by the presence of outstanding asynchronous I/O operations at the time the <i>exec</i> function is called. Whether any I/O is canceled, and which I/O may be canceled upon <i>exec</i> , is implementation-defined.
10170		
10171		
10172	CPT	The new process image shall inherit the CPU-time clock of the calling process image. This inheritance means that the process CPU-time clock of the process being <i>exec</i> -ed shall not be reinitialized or altered as a result of the <i>exec</i> function other than to reflect the time spent by the process executing the <i>exec</i> function itself.
10173		
10174		
10175		
10176	TCT	The initial value of the CPU-time clock of the initial thread of the new process image shall be set to zero.
10177		
10178	OB TRC	If the calling process is being traced, the new process image shall continue to be traced into the same trace stream as the original process image, but the new process image shall not inherit the mapping of trace event names to trace event type identifiers that was defined by calls to the <i>posix_trace_eventid_open()</i> or the <i>posix_trace_trid_eventid_open()</i> functions in the calling process image.
10179		
10180		
10181		
10182		
10183		If the calling process is a trace controller process, any trace streams that were created by the calling process shall be shut down as described in the <i>posix_trace_shutdown()</i> function.
10184		
10185		The thread ID of the initial thread in the new process image is unspecified.
10186		The size and location of the stack on which the initial thread in the new process image runs is unspecified.
10187		
10188		The initial thread in the new process image shall have its cancellation type set to <code>PTHREAD_CANCEL_DEFERRED</code> and its cancellation state set to <code>PTHREAD_CANCEL_ENABLED</code> .
10189		
10190		
10191		The initial thread in the new process image shall have all thread-specific data values set to <code>NULL</code> and all thread-specific data keys shall be removed by the call to <i>exec</i> without running destructors.
10192		
10193		
10194		The initial thread in the new process image shall be joinable, as if created with the <i>detachstate</i> attribute set to <code>PTHREAD_CREATE_JOINABLE</code> .
10195		
10196		The new process shall inherit at least the following attributes from the calling process image:
10197	XSI	• Nice value (see <i>nice()</i>)
10198	XSI	• <i>semadj</i> values (see <i>semop()</i>)
10199		• Process ID
10200		• Parent process ID
10201		• Process group ID
10202		• Session membership
10203		• Real user ID
10204		• Real group ID
10205		• Supplementary group IDs
10206		• Time left until an alarm clock signal (see <i>alarm()</i>)
10207		• Current working directory
10208		• Root directory
10209		• File mode creation mask (see <i>umask()</i>)

10210	XSI	<ul style="list-style-type: none"> • File size limit (see <i>ulimit()</i>)
10211		<ul style="list-style-type: none"> • Process signal mask (see <i>sigprocmask()</i>)
10212		<ul style="list-style-type: none"> • Pending signal (see <i>sigpending()</i>)
10213		<ul style="list-style-type: none"> • <i>tms_utime</i>, <i>tms_stime</i>, <i>tms_cutime</i>, and <i>tms_cstime</i> (see <i>times()</i>)
10214	XSI	<ul style="list-style-type: none"> • Resource limits
10215		<ul style="list-style-type: none"> • Controlling terminal
10216	XSI	<ul style="list-style-type: none"> • Interval timers
10217		The initial thread of the new process shall inherit at least the following attributes from the
10218		calling thread:
10219		<ul style="list-style-type: none"> • Signal mask (see <i>sigprocmask()</i> and <i>pthread_sigmask()</i>)
10220		<ul style="list-style-type: none"> • Pending signals (see <i>sigpending()</i>)
10221		All other process attributes defined in this volume of IEEE Std 1003.1-200x shall be inherited in
10222		the new process image from the old process image. All other thread attributes defined in this
10223		volume of IEEE Std 1003.1-200x shall be inherited in the initial thread in the new process image
10224		from the calling thread in the old process image. The inheritance of process or thread attributes
10225		not defined by this volume of IEEE Std 1003.1-200x is implementation-defined.
10226		A call to any <i>exec</i> function from a process with more than one thread shall result in all threads
10227		being terminated and the new executable image being loaded and executed. No destructor
10228		functions or cleanup handlers shall be called.
10229		Upon successful completion, the <i>exec</i> functions shall mark for update the <i>st_atime</i> field of the file.
10230		If an <i>exec</i> function failed but was able to locate the process image file, whether the <i>st_atime</i> field
10231		is marked for update is unspecified. Should the <i>exec</i> function succeed, the process image file
10232		shall be considered to have been opened with <i>open()</i> . The corresponding <i>close()</i> shall be
10233		considered to occur at a time after this open, but before process termination or successful
10234		completion of a subsequent call to one of the <i>exec</i> functions, <i>posix_spawn()</i> , or <i>posix_spawnnp()</i> .
10235		The <i>argv[]</i> and <i>envp[]</i> arrays of pointers and the strings to which those arrays point shall not be
10236		modified by a call to one of the <i>exec</i> functions, except as a consequence of replacing the process
10237		image.
10238	XSI	The saved resource limits in the new process image are set to be a copy of the process'
10239		corresponding hard and soft limits.

RETURN VALUE

10241 If one of the *exec* functions returns to the calling process image, an error has occurred; the return
 10242 value shall be -1 , and *errno* shall be set to indicate the error.

ERRORS

10243 The *exec* functions shall fail if:

10245	[E2BIG]	The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit of {ARG_MAX} bytes.
10246		
10247		
10248	[EACCES]	Search permission is denied for a directory listed in the new process image file's path prefix, or the new process image file denies execution permission, or the new process image file is not a regular file and the implementation does not support execution of files of its type.
10249		
10250		
10251		
10252	[EINVAL]	The new process image file has the appropriate permission and has a recognized executable binary format, but the system does not support execution of a file with this format.
10253		
10254		

- 10255 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* or *file*
10256 argument.
- 10257 [ENAMETOOLONG]
10258 The length of the *path* or *file* arguments exceeds {PATH_MAX} or a pathname
10259 component is longer than {NAME_MAX}.
- 10260 [ENOENT] A component of *path* or *file* does not name an existing file or *path* or *file* is an
10261 empty string.
- 10262 [ENOTDIR] A component of the new process image file's path prefix is not a directory.
- 10263 The *exec* functions, except for *execlp()* and *execvp()*, shall fail if:
- 10264 [ENOEXEC] The new process image file has the appropriate access permission but has an
10265 unrecognized format.
- 10266 The *fexecve()* function shall fail if:
- 10267 [EBADF] The *fd* argument is not a valid file descriptor open for executing.
- 10268 The *exec* functions may fail if:
- 10269 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
10270 resolution of the *path* or *file* argument.
- 10271 [ENAMETOOLONG]
10272 As a result of encountering a symbolic link in resolution of the *path* argument,
10273 the length of the substituted pathname string exceeded {PATH_MAX}.
- 10274 [ENOMEM] The new process image requires more memory than is allowed by the
10275 hardware or system-imposed memory management constraints.
- 10276 [ETXTBSY] The new process image file is a pure procedure (shared text) file that is
10277 currently open for writing by some process.

10278 EXAMPLES

10279 Using *execl()*

10280 The following example executes the *ls* command, specifying the pathname of the executable
10281 (*/bin/ls*) and using arguments supplied directly to the command to produce single-column
10282 output.

```
10283 #include <unistd.h>
10284 int ret;
10285 ...
10286 ret = execl ("/bin/ls", "ls", "-l", (char *)0);
```

10287 Using *execle()*

10288 The following example is similar to [Using *execl\(\)*](#) (on page 312). In addition, it specifies the
10289 environment for the new process image using the *env* argument.

```
10290 #include <unistd.h>
10291 int ret;
10292 char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
10293 ...
10294 ret = execle ("/bin/ls", "ls", "-l", (char *)0, env);
```

10295

Using execlp()

10296 The following example searches for the location of the *ls* command among the directories
10297 specified by the *PATH* environment variable.

```
10298 #include <unistd.h>
10299 int ret;
10300 ...
10301 ret = execlp ("ls", "ls", "-l", (char *)0);
```

10302

Using execv()

10303 The following example passes arguments to the *ls* command in the *cmd* array.

```
10304 #include <unistd.h>
10305 int ret;
10306 char *cmd[] = { "ls", "-l", (char *)0 };
10307 ...
10308 ret = execv ("/bin/ls", cmd);
```

10309

Using execve()

10310 The following example passes arguments to the *ls* command in the *cmd* array, and specifies the
10311 environment for the new process image using the *env* argument.

```
10312 #include <unistd.h>
10313 int ret;
10314 char *cmd[] = { "ls", "-l", (char *)0 };
10315 char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
10316 ...
10317 ret = execve ("/bin/ls", cmd, env);
```

10318

Using execvp()

10319 The following example searches for the location of the *ls* command among the directories
10320 specified by the *PATH* environment variable, and passes arguments to the *ls* command in the
10321 *cmd* array.

```
10322 #include <unistd.h>
10323 int ret;
10324 char *cmd[] = { "ls", "-l", (char *)0 };
10325 ...
10326 ret = execvp ("ls", cmd);
```

10327

APPLICATION USAGE

10328 As the state of conversion descriptors and message catalog descriptors in the new process image
10329 is undefined, conforming applications should not rely on their use and should close them prior
10330 to calling one of the *exec* functions.

10331 Applications that require other than the default POSIX locale should call *setlocale()* with the
10332 appropriate parameters to establish the locale of the new process.

10333 The *environ* array should not be accessed directly by the application.

10334 The new process might be invoked in a non-conforming environment if the *envp* array does not
10335 contain implementation-defined variables required by the implementation to provide a
10336 conforming environment. See the *_CS_V7_ENV* entry in **<unistd.h>** and *confstr()* for details.

10337 Applications should not depend on file descriptors 0, 1, and 2 being closed after an *exec*. A
10338 future version may allow these file descriptors to be automatically opened for any process.

10339 If an application wants to perform a checksum test of the file being executed before executing it,
10340 the file will need to be opened with read permission to perform the checksum test.

10341 Since execute permission is checked by *fexecve()*, the file description *fd* need not have been
10342 opened with the O_EXEC flag. However, if the file to be executed denies read and write
10343 permission for the process preparing to do the *exec*, the only way to provide the *fd* to *fexecve()*
10344 will be to use the O_EXEC flag when opening *fd*. In this case, the application will not be able to
10345 perform a checksum test since it will not be able to read the contents of the file.

10346 Note that when a file descriptor is opened with O_RDONLY, O_RDWR, or O_WRONLY mode,
10347 the file descriptor can be used to read, read and write, or write the file, respectively, even if the
10348 mode of the file changes after the file was opened. Using the O_EXEC open mode is different;
10349 *fexecve()* will ignore the mode that was used when the file descriptor was opened and the *exec*
10350 will fail if the mode of the file associated with *fd* does not grant execute permission to the calling
10351 process at the time *fexecve()* is called.

10352 RATIONALE

10353 Early proposals required that the value of *argc* passed to *main()* be “one or greater”. This was
10354 driven by the same requirement in drafts of the ISO C standard. In fact, historical
10355 implementations have passed a value of zero when no arguments are supplied to the caller of
10356 the *exec* functions. This requirement was removed from the ISO C standard and subsequently
10357 removed from this volume of IEEE Std 1003.1-200x as well. The wording, in particular the use of
10358 the word *should*, requires a Strictly Conforming POSIX Application to pass at least one argument
10359 to the *exec* function, thus guaranteeing that *argc* be one or greater when invoked by such an
10360 application. In fact, this is good practice, since many existing applications reference *argv[0]*
10361 without first checking the value of *argc*.

10362 The requirement on a Strictly Conforming POSIX Application also states that the value passed as
10363 the first argument be a filename associated with the process being started. Although some
10364 existing applications pass a pathname rather than a filename in some circumstances, a filename
10365 is more generally useful, since the common usage of *argv[0]* is in printing diagnostics. In some
10366 cases the filename passed is not the actual filename of the file; for example, many
10367 implementations of the *login* utility use a convention of prefixing a hyphen ('-') to the actual
10368 filename, which indicates to the command interpreter being invoked that it is a “login shell”.

10369 Historically there have been two ways that implementations can *exec* shell scripts.

10370 One common historical implementation is that the *execl()*, *execv()*, *execle()*, and *execve()*
10371 functions return an [ENOEXEC] error for any file not recognizable as executable, including a
10372 shell script. When the *execlp()* and *execvp()* functions encounter such a file, they assume the file
10373 to be a shell script and invoke a known command interpreter to interpret such files. This is now
10374 required by IEEE Std 1003.1-200x. These implementations of *execvp()* and *execlp()* only give the
10375 [ENOEXEC] error in the rare case of a problem with the command interpreter’s executable file.
10376 Because of these implementations, the [ENOEXEC] error is not mentioned for *execlp()* or
10377 *execvp()*, although implementations can still give it.

10378 Another way that some historical implementations handle shell scripts is by recognizing the first
10379 two bytes of the file as the character string “#!” and using the remainder of the first line of the
10380 file as the name of the command interpreter to execute.

10381 One potential source of confusion noted by the standard developers is over how the contents of
10382 a process image file affect the behavior of the *exec* family of functions. The following is a
10383 description of the actions taken:

- 10384 1. If the process image file is a valid executable (in a format that is executable and valid and
10385 having appropriate permission) for this system, then the system executes the file.
- 10386 2. If the process image file has appropriate permission and is in a format that is executable
10387 but not valid for this system (such as a recognized binary for another architecture), then
10388 this is an error and *errno* is set to [EINVAL] (see later RATIONALE on [EINVAL]).
- 10389 3. If the process image file has appropriate permission but is not otherwise recognized:
 - 10390 a. If this is a call to *execlp()* or *execvp()*, then they invoke a command interpreter
10391 assuming that the process image file is a shell script.
 - 10392 b. If this is not a call to *execlp()* or *execvp()*, then an error occurs and *errno* is set to
10393 [ENOEXEC].

10394 Applications that do not require to access their arguments may use the form:

```
10395 main(void)
```

10396 as specified in the ISO C standard. However, the implementation will always provide the two
10397 arguments *argc* and *argv*, even if they are not used.

10398 Some implementations provide a third argument to *main()* called *envp*. This is defined as a
10399 pointer to the environment. The ISO C standard specifies invoking *main()* with two arguments,
10400 so implementations must support applications written this way. Since this volume of
10401 IEEE Std 1003.1-200x defines the global variable *environ*, which is also provided by historical
10402 implementations and can be used anywhere that *envp* could be used, there is no functional need
10403 for the *envp* argument. Applications should use the *getenv()* function rather than accessing the
10404 environment directly via either *envp* or *environ*. Implementations are required to support the
10405 two-argument calling sequence, but this does not prohibit an implementation from supporting
10406 *envp* as an optional third argument.

10407 This volume of IEEE Std 1003.1-200x specifies that signals set to SIG_IGN remain set to
10408 SIG_IGN, and that the new process image inherits the signal mask of the thread that called *exec*
10409 in the old process image. This is consistent with historical implementations, and it permits some
10410 useful functionality, such as the *nohup* command. However, it should be noted that many
10411 existing applications wrongly assume that they start with certain signals set to the default action
10412 and/or unblocked. In particular, applications written with a simpler signal model that does not
10413 include blocking of signals, such as the one in the ISO C standard, may not behave properly if
10414 invoked with some signals blocked. Therefore, it is best not to block or ignore signals across
10415 *execs* without explicit reason to do so, and especially not to block signals across *execs* of arbitrary
10416 (not closely co-operating) programs.

10417 The *exec* functions always save the value of the effective user ID and effective group ID of the
10418 process at the completion of the *exec*, whether or not the set-user-ID or the set-group-ID bit of
10419 the process image file is set.

10420 The statement about *argv[]* and *envp[]* being constants is included to make explicit to future
10421 writers of language bindings that these objects are completely constant. Due to a limitation of
10422 the ISO C standard, it is not possible to state that idea in standard C. Specifying two levels of
10423 *const-qualification* for the *argv[]* and *envp[]* parameters for the *exec* functions may seem to be the
10424 natural choice, given that these functions do not modify either the array of pointers or the
10425 characters to which the function points, but this would disallow existing correct code. Instead,
10426 only the array of pointers is noted as constant. The table of assignment compatibility for *dst=src*
10427 derived from the ISO C standard summarizes the compatibility:

<i>dst:</i>	char *[]	const char *[]	char *const[]	const char *const[]
<i>src:</i>				
char *[]	VALID	—	VALID	—
const char *[]	—	VALID	—	VALID
char * const []	—	—	VALID	—
const char *const[]	—	—	—	VALID

Since all existing code has a source type matching the first row, the column that gives the most valid combinations is the third column. The only other possibility is the fourth column, but using it would require a cast on the *argv* or *envp* arguments. It is unfortunate that the fourth column cannot be used, because the declaration a non-expert would naturally use would be that in the second row.

The ISO C standard and this volume of IEEE Std 1003.1-200x do not conflict on the use of *environ*, but some historical implementations of *environ* may cause a conflict. As long as *environ* is treated in the same way as an entry point (for example, *fork()*), it conforms to both standards. A library can contain *fork()*, but if there is a user-provided *fork()*, that *fork()* is given precedence and no problem ensues. The situation is similar for *environ*: the definition in this volume of IEEE Std 1003.1-200x is to be used if there is no user-provided *environ* to take precedence. At least three implementations are known to exist that solve this problem.

[E2BIG] The limit {ARG_MAX} applies not just to the size of the argument list, but to the sum of that and the size of the environment list.

[EFAULT] Some historical systems return [EFAULT] rather than [ENOEXEC] when the new process image file is corrupted. They are non-conforming.

[EINVAL] This error condition was added to IEEE Std 1003.1-200x to allow an implementation to detect executable files generated for different architectures, and indicate this situation to the application. Historical implementations of shells, *execvp()*, and *execlp()* that encounter an [ENOEXEC] error will execute a shell on the assumption that the file is a shell script. This will not produce the desired effect when the file is a valid executable for a different architecture. An implementation may now choose to avoid this problem by returning [EINVAL] when a valid executable for a different architecture is encountered. Some historical implementations return [EINVAL] to indicate that the *path* argument contains a character with the high order bit set. The standard developers chose to deviate from historical practice for the following reasons:

1. The new utilization of [EINVAL] will provide some measure of utility to the user community.
2. Historical use of [EINVAL] is not acceptable in an internationalized operating environment.

[ENAMETOOLONG]

Since the file pathname may be constructed by taking elements in the *PATH* variable and putting them together with the filename, the [ENAMETOOLONG] error condition could also be reached this way.

[ETXTBSY]

System V returns this error when the executable file is currently open for writing by some process. This volume of IEEE Std 1003.1-200x neither requires nor prohibits this behavior.

Other systems (such as System V) may return [EINTR] from *exec*. This is not addressed by this volume of IEEE Std 1003.1-200x, but implementations may have a window between the call to *exec* and the time that a signal could cause one of the *exec* calls to return with [EINTR].

An explicit statement regarding the floating-point environment (as defined in the **<fenv.h>**

10476 header) was added to make it clear that the floating-point environment is set to its default when
 10477 a call to one of the *exec* functions succeeds. The requirements for inheritance or setting to the
 10478 default for other process and thread start-up functions is covered by more generic statements in
 10479 their descriptions and can be summarized as follows:

10480 *posix_spawn()* Set to default.
 10481 *fork()* Inherit.
 10482 *pthread_create()* Inherit.

10483 The purpose of the *fexecve()* function is to enable executing a file which has been verified to be
 10484 the intended file. It is possible to actively check the file by reading from the file descriptor and be
 10485 sure that the file is not exchanged for another between the reading and the execution.
 10486 Alternatively, a function like *openat()* can be used to open a file which has been found by
 10487 reading the content of a directory using *readdir()*.

10488 FUTURE DIRECTIONS

10489 None.

10490 SEE ALSO

10491 *alarm()*, *atexit()*, *chmod()*, *close()*, *confstr()*, *exit()*, *fcntl()*, *fork()*, *fstatvfs()*, *getenv()*, *getitimer()*,
 10492 *getrlimit()*, *mknod()*, *mmap()*, *nice()*, *open()*, *posix_spawn()*, *posix_trace_create()*,
 10493 *posix_trace_event()*, *posix_trace_eventid_equal()*, *pthread_atfork()*, *pthread_sigmask()*, *putenv()*,
 10494 *readdir()*, *semop()*, *setlocale()*, *shmat()*, *sigaction()*, *sigaltstack()*, *sigpending()*, *system()*, *times()*,
 10495 *ulimit()*, *umask()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

10496 CHANGE HISTORY

10497 First released in Issue 1. Derived from Issue 1 of the SVID.

10498 Issue 5

10499 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
 10500 Threads Extension.

10501 Large File Summit extensions are added.

10502 Issue 6

10503 The following new requirements on POSIX implementations derive from alignment with the
 10504 Single UNIX Specification:

- 10505 • In the DESCRIPTION, behavior is defined for when the process image file is not a valid
 10506 executable.
- 10507 • In this issue, `_POSIX_SAVED_IDS` is mandated, thus the effective user ID and effective
 10508 group ID of the new process image shall be saved (as the saved set-user-ID and the saved
 10509 set-group-ID) for use by the *setuid()* function.
- 10510 • The [ELOOP] mandatory error condition is added.
- 10511 • A second [ENAMETOOLONG] is added as an optional error condition.
- 10512 • The [ETXTBSY] optional error condition is added.

10513 The following changes were made to align with the IEEE P1003.1a draft standard:

- 10514 • The [EINVAL] mandatory error condition is added.
- 10515 • The [ELOOP] optional error condition is added.

10516 The description of CPU-time clock semantics is added for alignment with IEEE Std 1003.1d-1999.

10517 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics
 10518 for typed memory.

10519 The normative text is updated to avoid use of the term “must” for application requirements.

- 10520 The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.
- 10521 IEEE PASC Interpretation 1003.1 #132 is applied.
- 10522 The DESCRIPTION is updated to make it explicit that the floating-point environment in the new
10523 process image is set to the default.
- 10524 The DESCRIPTION and RATIONALE are updated to include clarifications of how the contents
10525 of a process image file affect the behavior of the *exec* functions.
- 10526 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/15 is applied, adding a new paragraph to
10527 the DESCRIPTION and text to the end of the APPLICATION USAGE section. This change
10528 addresses a security concern, where implementations may want to reopen file descriptors 0, 1,
10529 and 2 for programs with the set-user-id or set-group-id file mode bits calling the *exec* family of
10530 functions.
- 10531 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/24 is applied, applying changes to the
10532 DESCRIPTION, addressing which attributes are inherited by threads, and behavioral
10533 requirements for threads attributes.
- 10534 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/25 is applied, updating text in the
10535 RATIONALE from “the process signal mask be unchanged across an *exec*” to “the new process
10536 image inherits the signal mask of the thread that called *exec* in the old process image”.
- 10537 **Issue 7**
- 10538 Austin Group Interpretation 1003.1-2001 #047 is applied, adding the description of `_CS_V7_ENV`
10539 to the APPLICATION USAGE.
- 10540 The *fexecve()* function is added from The Open Group Technical Standard, 2006, Extended API
10541 Set Part 2.
- 10542 Functionality relating to the Asynchronous Input and Output, Memory Mapped Files, Threads,
10543 and Timers options is moved to the Base.

10544 **NAME**

10545 exit — terminate a process

10546 **SYNOPSIS**

10547 #include <stdlib.h>

10548 void exit(int status);

10549 **DESCRIPTION**

10550 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 10551 conflict between the requirements described here and the ISO C standard is unintentional. This
 10552 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

10553 CX The value of *status* may be 0, EXIT_SUCCESS, EXIT_FAILURE, or any other value, though only
 10554 the least significant 8 bits (that is, *status* & 0377) shall be available to a waiting parent process.

10555 The *exit()* function shall first call all functions registered by *atexit()*, in the reverse order of their
 10556 registration, except that a function is called after any previously registered functions that had
 10557 already been called at the time it was registered. Each function is called as many times as it was
 10558 registered. If, during the call to any such function, a call to the *longjmp()* function is made that
 10559 would terminate the call to the registered function, the behavior is undefined.

10560 If a function registered by a call to *atexit()* fails to return, the remaining registered functions shall
 10561 not be called and the rest of the *exit()* processing shall not be completed. If *exit()* is called more
 10562 than once, the behavior is undefined.

10563 The *exit()* function shall then flush all open streams with unwritten buffered data and close all
 10564 open streams. Finally, the process shall be terminated with the same consequences as described
 10565 in [Consequences of Process Termination](#) (on page 87).

10566 **RETURN VALUE**10567 The *exit()* function does not return.10568 **ERRORS**

10569 No errors are defined.

10570 **EXAMPLES**

10571 None.

10572 **APPLICATION USAGE**

10573 None.

10574 **RATIONALE**10575 See *_Exit()*.10576 **FUTURE DIRECTIONS**

10577 None.

10578 **SEE ALSO**

10579 [_Exit\(\)](#), [atexit\(\)](#), [exec](#), [longjmp\(\)](#), [tmpfile\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x,
 10580 [<stdlib.h>](#)

10581 **CHANGE HISTORY**10582 **Issue 7**

10583 Austin Group Interpretation 1003.1-2001 #031 is applied, separating the *_Exit()* and *_exit()*
 10584 functions from the *exit()* function.

10585 Austin Group Interpretation 1003.1-2001 #085 is applied.

10586 **NAME**
 10587 `exp, expf, expl` — exponential function

10588 **SYNOPSIS**
 10589 `#include <math.h>`
 10590 `double exp(double x);`
 10591 `float expf(float x);`
 10592 `long double expl(long double x);`

10593 **DESCRIPTION**

10594 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 10595 conflict between the requirements described here and the ISO C standard is unintentional. This
 10596 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

10597 These functions shall compute the base-*e* exponential of *x*.

10598 An application wishing to check for error situations should set *errno* to zero and call
 10599 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 10600 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 10601 zero, an error has occurred.

10602 **RETURN VALUE**

10603 Upon successful completion, these functions shall return the exponential value of *x*.

10604 If the correct value would cause overflow, a range error shall occur and *exp()*, *expf()*, and *expl()*
 10605 shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

10606 If the correct value would cause underflow, and is not representable, a range error may occur,
 10607 and either 0.0 (if supported), or an implementation-defined value shall be returned.

10608 MX If *x* is NaN, a NaN shall be returned.

10609 If *x* is ± 0 , 1 shall be returned.

10610 If *x* is $-\text{Inf}$, +0 shall be returned.

10611 If *x* is $+\text{Inf}$, *x* shall be returned.

10612 If the correct value would cause underflow, and is representable, a range error may occur and
 10613 the correct value shall be returned.

10614 **ERRORS**

10615 These functions shall fail if:

10616 Range Error The result overflows.

10617 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 10618 then *errno* shall be set to [ERANGE]. If the integer expression
 10619 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
 10620 floating-point exception shall be raised.

10621 These functions may fail if:

10622 Range Error The result underflows.

10623 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 10624 then *errno* shall be set to [ERANGE]. If the integer expression
 10625 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 10626 floating-point exception shall be raised.

10627 **EXAMPLES**10628 **Computing the Density of the Standard Normal Distribution**

10629 This function shows an implementation for the density of the standard normal distribution
 10630 using *exp()*. This example uses the constant `M_PI` which is part of the XSI option.

```
10631 #include <math.h>
10632
10633 double
10634 normal_density (double x)
10635 {
10636     return exp(-x*x/2) / sqrt (2*M_PI);
10637 }
```

10637 **APPLICATION USAGE**

10638 Note that for IEEE Std 754-1985 **double**, $709.8 < x$ implies *exp(x)* has overflowed. The value
 10639 $x < -708.4$ implies *exp(x)* has underflowed.

10640 On error, the expressions (*math_errhandling* & `MATH_ERRNO`) and (*math_errhandling* &
 10641 `MATH_ERREXCEPT`) are independent of each other, but at least one of them must be non-zero.

10642 **RATIONALE**

10643 None.

10644 **FUTURE DIRECTIONS**

10645 None.

10646 **SEE ALSO**

10647 *feclearexcept()*, *fetestexcept()*, *isnan()*, *log()*, the Base Definitions volume of IEEE Std 1003.1-200x,
 10648 Section 4.18, Treatment of Error Conditions for Mathematical Functions, `<math.h>`

10649 **CHANGE HISTORY**

10650 First released in Issue 1. Derived from Issue 1 of the SVID.

10651 **Issue 5**

10652 The **DESCRIPTION** is updated to indicate how an application should check for an error. This
 10653 text was previously published in the **APPLICATION USAGE** section.

10654 **Issue 6**

10655 The *expf()* and *expl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

10656 The **DESCRIPTION**, **RETURN VALUE**, **ERRORS**, and **APPLICATION USAGE** sections are
 10657 revised to align with the ISO/IEC 9899:1999 standard.

10658 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
 10659 marked.

10660 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/26 is applied, adding the example to the
 10661 **EXAMPLES** section.

10662 **NAME**
 10663 `exp2, exp2f, exp2l` — exponential base 2 functions

10664 **SYNOPSIS**
 10665 `#include <math.h>`
 10666 `double exp2(double x);`
 10667 `float exp2f(float x);`
 10668 `long double exp2l(long double x);`

10669 **DESCRIPTION**

10670 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 10671 conflict between the requirements described here and the ISO C standard is unintentional. This
 10672 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

10673 These functions shall compute the base-2 exponential of x .

10674 An application wishing to check for error situations should set *errno* to zero and call
 10675 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 10676 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 10677 zero, an error has occurred.

10678 **RETURN VALUE**

10679 Upon successful completion, these functions shall return 2^x .

10680 If the correct value would cause overflow, a range error shall occur and *exp2()*, *exp2f()*, and
 10681 *exp2l()* shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL,
 10682 respectively.

10683 If the correct value would cause underflow, and is not representable, a range error may occur,
 10684 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

10685 MX If x is NaN, a NaN shall be returned.

10686 If x is ± 0 , 1 shall be returned.

10687 If x is $-\text{Inf}$, +0 shall be returned.

10688 If x is $+\text{Inf}$, x shall be returned.

10689 If the correct value would cause underflow, and is representable, a range error may occur and
 10690 the correct value shall be returned.

10691 **ERRORS**

10692 These functions shall fail if:

10693 Range Error The result overflows.

10694 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 10695 then *errno* shall be set to [ERANGE]. If the integer expression
 10696 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
 10697 floating-point exception shall be raised.

10698 These functions may fail if:

10699 Range Error The result underflows.

10700 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 10701 then *errno* shall be set to [ERANGE]. If the integer expression
 10702 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 10703 floating-point exception shall be raised.

10704
10705

10706
10707
10708

10709
10710

10711
10712

10713
10714

10715
10716
10717
10718

10719
10720

EXAMPLES

None.

APPLICATION USAGE

For IEEE Std 754-1985 **double**, $1024 \leq x$ implies $\text{exp2}(x)$ has overflowed. The value $x < -1022$ implies $\text{exp}(x)$ has underflowed.

On error, the expressions $(\text{math_errhandling} \ \& \ \text{MATH_ERRNO})$ and $(\text{math_errhandling} \ \& \ \text{MATH_ERREXCEPT})$ are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exp(), *feclearexcept()*, *fetestexcept()*, *isnan()*, *log()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

DRAFT

10721 **NAME**
 10722 `expm1, expm1f, expm1l` — compute exponential functions

10723 **SYNOPSIS**
 10724 `#include <math.h>`
 10725 `double expm1(double x);`
 10726 `float expm1f(float x);`
 10727 `long double expm1l(long double x);`

10728 **DESCRIPTION**
 10729 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 10730 conflict between the requirements described here and the ISO C standard is unintentional. This
 10731 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

10732 These functions shall compute $e^x-1.0$.
 10733 An application wishing to check for error situations should set *errno* to zero and call
 10734 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 10735 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 10736 zero, an error has occurred.

10737 **RETURN VALUE**
 10738 Upon successful completion, these functions return $e^x-1.0$.
 10739 If the correct value would cause overflow, a range error shall occur and *expm1()*, *expm1f()*, and
 10740 *expm1l()* shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL,
 10741 respectively.

10742 MX If *x* is NaN, a NaN shall be returned.
 10743 If *x* is ± 0 , ± 0 shall be returned.
 10744 If *x* is $-\text{Inf}$, -1 shall be returned.
 10745 If *x* is $+\text{Inf}$, *x* shall be returned.
 10746 If *x* is subnormal, a range error may occur and *x* should be returned.

10747 **ERRORS**
 10748 These functions shall fail if:
 10749 Range Error The result overflows.
 10750 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 10751 then *errno* shall be set to [ERANGE]. If the integer expression
 10752 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
 10753 floating-point exception shall be raised.

10754 These functions may fail if:
 10755 MX Range Error The value of *x* is subnormal.
 10756 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 10757 then *errno* shall be set to [ERANGE]. If the integer expression
 10758 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 10759 floating-point exception shall be raised.

EXAMPLES

10760
10761 None.

APPLICATION USAGE

10762 The value of $\text{expm1}(x)$ may be more accurate than $\text{exp}(x)-1.0$ for small values of x .

10763 The $\text{expm1}()$ and $\text{log1p}()$ functions are useful for financial calculations of $((1+x)^n-1)/x$, namely:

10764
10765 $\text{expm1}(n * \text{log1p}(x)) / x$

10766 when x is very small (for example, when calculating small daily interest rates). These functions
10767 also simplify writing accurate inverse hyperbolic functions.

10768 For IEEE Std 754-1985 **double**, $709.8 < x$ implies $\text{expm1}(x)$ has overflowed.

10769 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
10770 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

10771
10772 None.

FUTURE DIRECTIONS

10773
10774 None.

SEE ALSO

10775 $\text{exp}()$, $\text{feclearexcept}()$, $\text{fetestexcept}()$, $\text{ilogb}()$, $\text{log1p}()$, the Base Definitions volume of
10776 IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
10777 <math.h>
10778

CHANGE HISTORY

10779 First released in Issue 4, Version 2.
10780

Issue 5

10781 Moved from X/OPEN UNIX extension to BASE.
10782

Issue 6

10783 The $\text{expm1f}()$ and $\text{expm1l}()$ functions are added for alignment with the ISO/IEC 9899:1999
10784 standard.
10785

10786 The $\text{expm1}()$ function is no longer marked as an extension.

10787 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
10788 revised to align with the ISO/IEC 9899:1999 standard.

10789 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
10790 marked.

10791 **NAME**
 10792 `fabs, fabsf, fabsl` — absolute value function

10793 **SYNOPSIS**
 10794 `#include <math.h>`
 10795 `double fabs(double x);`
 10796 `float fabsf(float x);`
 10797 `long double fabsl(long double x);`

10798 **DESCRIPTION**
 10799 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 10800 conflict between the requirements described here and the ISO C standard is unintentional. This
 10801 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

10802 These functions shall compute the absolute value of their argument x , $|x|$.

10803 **RETURN VALUE**
 10804 Upon successful completion, these functions shall return the absolute value of x .

10805 MX If x is NaN, a NaN shall be returned.
 10806 If x is ± 0 , $+0$ shall be returned.
 10807 If x is $\pm\text{Inf}$, $+\text{Inf}$ shall be returned.

10808 **ERRORS**
 10809 No errors are defined.

10810 EXAMPLES

10811 Computing the 1-Norm of a Floating-Point Vector

10812 This example shows the use of `fabs()` to compute the 1-norm of a vector defined as follows:

10813 $\text{norm1}(v) = |v[0]| + |v[1]| + \dots + |v[n-1]|$

10814 where $|x|$ denotes the absolute value of x , n denotes the vector's dimension $v[i]$ denotes the i -th
 10815 component of v ($0 \leq i < n$).

```
10816 #include <math.h>
10817 double
10818 norm1(const double v[], const int n)
10819 {
10820     int    i;
10821     double n1_v; /* 1-norm of v */
10822
10823     n1_v = 0;
10824     for (i=0; i<n; i++) {
10825         n1_v += fabs (v[i]);
10826     }
10827     return n1_v;
10828 }
```

10828 **APPLICATION USAGE**
 10829 None.

10830
10831
10832
10833
10834
10835
10836
10837
10838
10839
10840
10841
10842
10843
10844
10845
10846
10847
10848

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isnan(), the Base Definitions volume of IEEE Std 1003.1-200x, <math.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

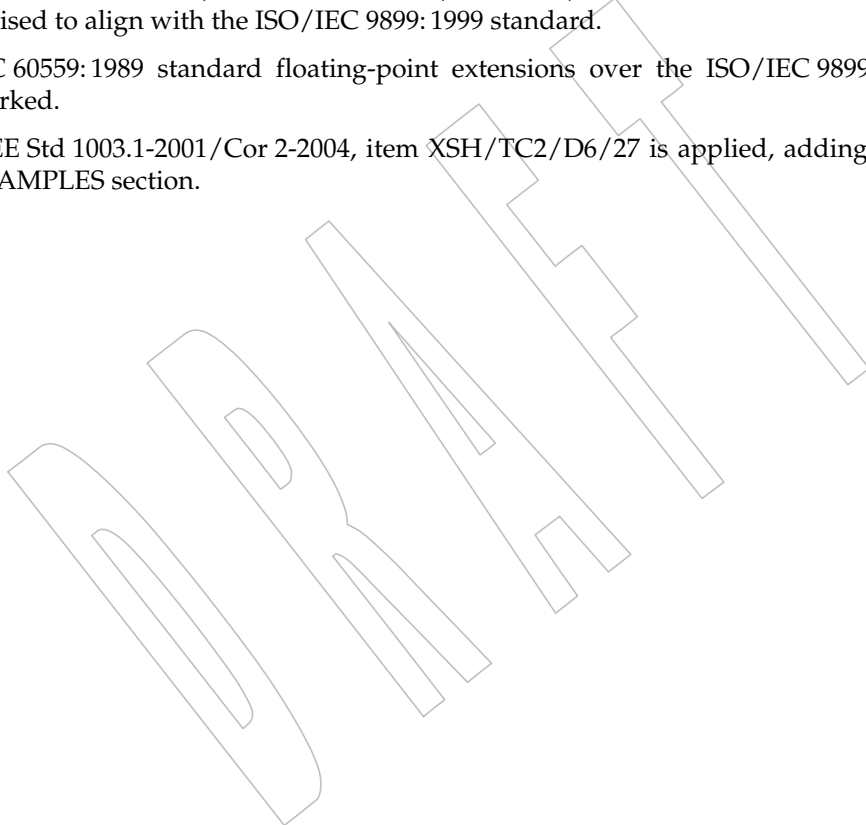
Issue 6

The *fabsf()* and *fabsl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

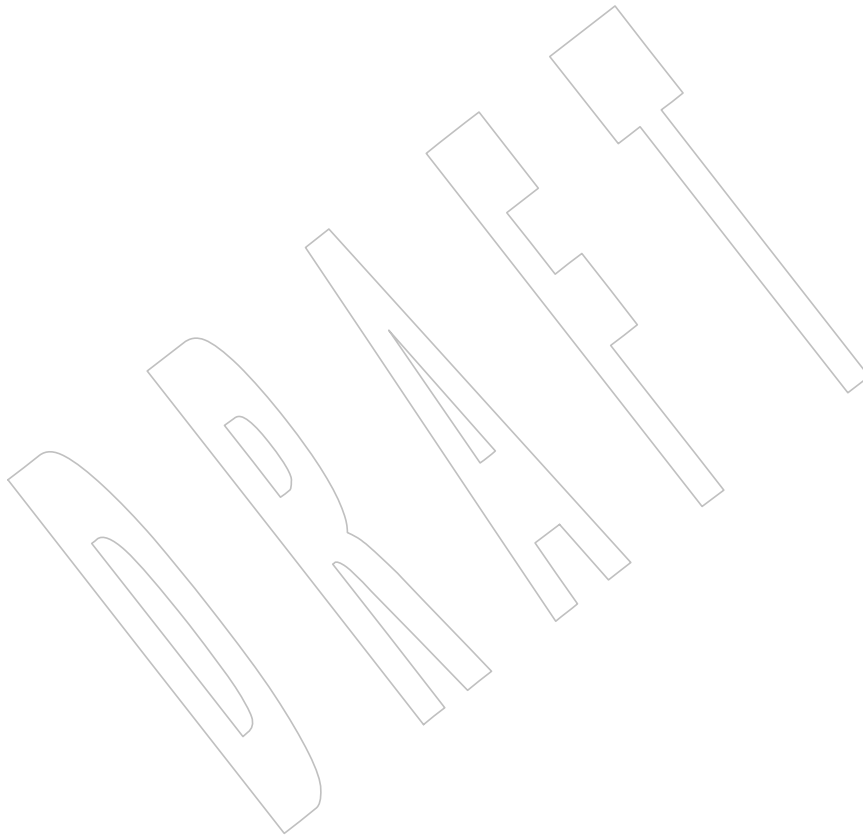
IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/27 is applied, adding the example to the EXAMPLES section.



10849 **NAME**
10850 `faccessat` — determine accessibility of a file relative to directory file descriptor

10851 **SYNOPSIS**
10852 `#include <unistd.h>`
10853 `int faccessat(int fd, const char *path, int amode, int flag);`

10854 **DESCRIPTION**
10855 Refer to [access\(\)](#).



10856 **NAME**

10857 **fattach** — attach a STREAMS-based file descriptor to a file in the file system name space
 10858 **(STREAMS)**

10859 **SYNOPSIS**

```
10860 OB XSR #include <stropts.h>
10861 int fattach(int fildes, const char *path);
```

10862 **DESCRIPTION**

10863 The *fattach()* function shall attach a STREAMS-based file descriptor to a file, effectively
 10864 associating a pathname with *fildes*. The application shall ensure that the *fildes* argument is a
 10865 valid open file descriptor associated with a STREAMS file. The *path* argument points to a
 10866 pathname of an existing file. The application shall have the appropriate privileges or be the
 10867 owner of the file named by *path* and have write permission. A successful call to *fattach()* shall
 10868 cause all pathnames that name the file named by *path* to name the STREAMS file associated with
 10869 *fildes*, until the STREAMS file is detached from the file. A STREAMS file can be attached to more
 10870 than one file and can have several pathnames associated with it.

10871 The attributes of the named STREAMS file shall be initialized as follows: the permissions, user
 10872 ID, group ID, and times are set to those of the file named by *path*, the number of links is set to 1,
 10873 and the size and device identifier are set to those of the STREAMS file associated with *fildes*. If
 10874 any attributes of the named STREAMS file are subsequently changed (for example, by *chmod()*),
 10875 neither the attributes of the underlying file nor the attributes of the STREAMS file to which *fildes*
 10876 refers shall be affected.

10877 File descriptors referring to the underlying file, opened prior to an *fattach()* call, shall continue to
 10878 refer to the underlying file.

10879 **RETURN VALUE**

10880 Upon successful completion, *fattach()* shall return 0. Otherwise, -1 shall be returned and *errno*
 10881 set to indicate the error.

10882 **ERRORS**

10883 The *fattach()* function shall fail if:

- | | | |
|-------|----------------|---|
| 10884 | [EACCES] | Search permission is denied for a component of the path prefix, or the process is the owner of <i>path</i> but does not have write permissions on the file named by <i>path</i> . |
| 10885 | | |
| 10886 | | |
| 10887 | [EBADF] | The <i>fildes</i> argument is not a valid open file descriptor. |
| 10888 | [EBUSY] | The file named by <i>path</i> is currently a mount point or has a STREAMS file attached to it. |
| 10889 | | |
| 10890 | [ELOOP] | A loop exists in symbolic links encountered during resolution of the <i>path</i> argument. |
| 10891 | | |
| 10892 | [ENAMETOOLONG] | The size of <i>path</i> exceeds {PATH_MAX} or a component of <i>path</i> is longer than {NAME_MAX}. |
| 10893 | | |
| 10894 | | |
| 10895 | [ENOENT] | A component of <i>path</i> does not name an existing file or <i>path</i> is an empty string. |
| 10896 | [ENOTDIR] | A component of the path prefix is not a directory. |

- 10897 [EPERM] The effective user ID of the process is not the owner of the file named by *path*
 10898 and the process does not have appropriate privilege.
- 10899 The *fattach()* function may fail if:
- 10900 [EINVAL] The *fd* argument does not refer to a STREAMS file.
- 10901 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 10902 resolution of the *path* argument.
- 10903 [ENAMETOOLONG]
 10904 Pathname resolution of a symbolic link produced an intermediate result
 10905 whose length exceeds {PATH_MAX}.
- 10906 [EXDEV] A link to a file on another file system was attempted.

EXAMPLES**Attaching a File Descriptor to a File**

In the following example, *fd* refers to an open STREAMS file. The call to *fattach()* associates this STREAM with the file **/tmp/named-STREAM**, such that any future calls to open **/tmp/named-STREAM**, prior to breaking the attachment via a call to *fdetach()*, will instead create a new file handle referring to the STREAMS file associated with *fd*.

```
10913 #include <stropts.h>
10914 ...
10915     int fd;
10916     char *filename = "/tmp/named-STREAM";
10917     int ret;
10918     ret = fattach(fd, filename);
```

APPLICATION USAGE

The *fattach()* function behaves similarly to the traditional *mount()* function in the way a file is temporarily replaced by the root directory of the mounted file system. In the case of *fattach()*, the replaced file need not be a directory and the replacing file is a STREAMS file.

RATIONALE

The file attributes of a file which has been the subject of an *fattach()* call are specifically set because of an artefact of the original implementation. The internal mechanism was the same as for the *mount()* function. Since *mount()* is typically only applied to directories, the effects when applied to a regular file are a little surprising, especially as regards the link count which rigidly remains one, even if there were several links originally and despite the fact that all original links refer to the STREAM as long as the *fattach()* remains in effect.

FUTURE DIRECTIONS

The *fattach()* function may be removed in a future version.

SEE ALSO

fdetach(), *isastream()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The [EXDEV] error is added to the list of optional errors in the ERRORS section.

10939

Issue 6

10940

This function is marked as part of the XSI STREAMS Option Group.

10941

The normative text is updated to avoid use of the term “must” for application requirements.

10942

10943

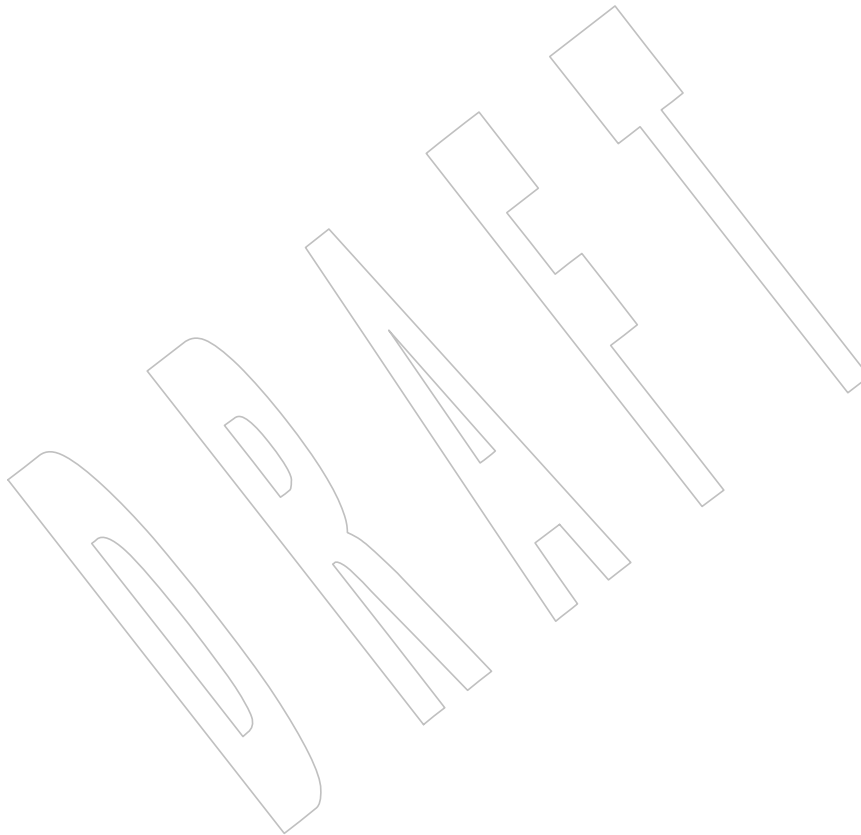
The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

10944

Issue 7

10945

The *fattach()* function is marked obsolescent.



10946 **NAME**
 10947 `fchdir` — change working directory

10948 **SYNOPSIS**
 10949 `#include <unistd.h>`
 10950 `int fchdir(int fildev);`

10951 **DESCRIPTION**
 10952 The `fchdir()` function shall be equivalent to `chdir()` except that the directory that is to be the new
 10953 current working directory is specified by the file descriptor *fildev*.

10954 A conforming application can obtain a file descriptor for a file of type directory using `open()`,
 10955 provided that the file status flags and access modes do not contain `O_WRONLY` or `O_RDWR`.

10956 **RETURN VALUE**
 10957 Upon successful completion, `fchdir()` shall return 0. Otherwise, it shall return `-1` and set `errno` to
 10958 indicate the error. On failure the current working directory shall remain unchanged.

10959 **ERRORS**
 10960 The `fchdir()` function shall fail if:

10961	[EACCES]	Search permission is denied for the directory referenced by <i>fildev</i> .
10962	[EBADF]	The <i>fildev</i> argument is not an open file descriptor.
10963	[ENOTDIR]	The open file descriptor <i>fildev</i> does not refer to a directory.

10964 The `fchdir()` may fail if:

10965	[EINTR]	A signal was caught during the execution of <code>fchdir()</code> .
10966	[EIO]	An I/O error occurred while reading from or writing to the file system.

10967 **EXAMPLES**
 10968 None.

10969 **APPLICATION USAGE**
 10970 None.

10971 **RATIONALE**
 10972 None.

10973 **FUTURE DIRECTIONS**
 10974 None.

10975 **SEE ALSO**
 10976 `chdir()`, `dirfd()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<unistd.h>`

10977 **CHANGE HISTORY**
 10978 First released in Issue 4, Version 2.

10979 **Issue 5**
 10980 Moved from X/OPEN UNIX extension to BASE.

10981 **Issue 7**
 10982 The `fchdir()` function is moved from the XSI option to the Base.

10983 **NAME**

10984 fchmod — change mode of a file

10985 **SYNOPSIS**

10986 #include <sys/stat.h>

10987 int fchmod(int *fildes*, mode_t *mode*);10988 **DESCRIPTION**10989 The *fchmod()* function shall be equivalent to *chmod()* except that the file whose permissions are
10990 changed is specified by the file descriptor *fildes*.10991 SHM If *fildes* references a shared memory object, the *fchmod()* function need only affect the S_IRUSR,
10992 S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits.10993 TYM If *fildes* references a typed memory object, the behavior of *fchmod()* is unspecified.10994 If *fildes* refers to a socket, the behavior of *fchmod()* is unspecified.10995 OB XSR If *fildes* refers to a STREAM (which is *fattach()*-ed into the file system name space) the call
10996 returns successfully, doing nothing.10997 **RETURN VALUE**10998 Upon successful completion, *fchmod()* shall return 0. Otherwise, it shall return -1 and set *errno* to
10999 indicate the error.11000 **ERRORS**11001 The *fchmod()* function shall fail if:11002 [EBADF] The *fildes* argument is not an open file descriptor.11003 [EPERM] The effective user ID does not match the owner of the file and the process does
11004 not have appropriate privilege.11005 [EROFS] The file referred to by *fildes* resides on a read-only file system.11006 The *fchmod()* function may fail if:11007 XSI [EINTR] The *fchmod()* function was interrupted by a signal.11008 XSI [EINVAL] The value of the *mode* argument is invalid.11009 [EINVAL] The *fildes* argument refers to a pipe and the implementation disallows
11010 execution of *fchmod()* on a pipe.11011 **EXAMPLES**11012 **Changing the Current Permissions for a File**11013 The following example shows how to change the permissions for a file named **/home/cnd/mod1**
11014 so that the owner and group have read/write/execute permissions, but the world only has
11015 read/write permissions.

11016 #include <sys/stat.h>

11017 #include <fcntl.h>

11018 mode_t mode;

11019 int fildes;

11020 ...

11021 fildes = open("/home/cnd/mod1", O_RDWR);

11022 fchmod(fildes, S_IRWXU | S_IRWXG | S_IROTH | S_IWOTH);

fchmod()

11023	APPLICATION USAGE
11024	None.
11025	RATIONALE
11026	None.
11027	FUTURE DIRECTIONS
11028	None.
11029	SEE ALSO
11030	<i>chmod()</i> , <i>chown()</i> , <i>creat()</i> , <i>fcntl()</i> , <i>fstatat()</i> , <i>fstatvfs()</i> , <i>mknod()</i> , <i>open()</i> , <i>read()</i> , <i>write()</i> , the Base
11031	Definitions volume of IEEE Std 1003.1-200x, <sys/stat.h>
11032	CHANGE HISTORY
11033	First released in Issue 4, Version 2.
11034	Issue 5
11035	Moved from X/OPEN UNIX extension to BASE and aligned with <i>fchmod()</i> in the POSIX
11036	Realtime Extension. Specifically, the second paragraph of the DESCRIPTION is added and a
11037	second instance of [EINVAL] is defined in the list of optional errors.
11038	Issue 6
11039	The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by stating that <i>fchmod()</i>
11040	behavior is unspecified for typed memory objects.

NAME

fchmodat — change mode of a file relative to directory file descriptor

SYNOPSIS

```
#include <sys/stat.h>

int fchmodat(int fd, const char *path, mode_t mode, int flag);
```

DESCRIPTION

Refer to [chmod\(2\)](#).

DRAFT

11048 **NAME**11049 `fchown` — change owner and group of a file11050 **SYNOPSIS**11051 `#include <unistd.h>`11052 `int fchown(int fildev, uid_t owner, gid_t group);`11053 **DESCRIPTION**11054 The `fchown()` function shall be equivalent to `chown()` except that the file whose owner and group
11055 are changed is specified by the file descriptor *fildev*.11056 **RETURN VALUE**11057 Upon successful completion, `fchown()` shall return 0. Otherwise, it shall return `-1` and set *errno* to
11058 indicate the error.11059 **ERRORS**11060 The `fchown()` function shall fail if:11061 [EBADF] The *fildev* argument is not an open file descriptor.11062 [EPERM] The effective user ID does not match the owner of the file or the process does
11063 not have appropriate privilege and `_POSIX_CHOWN_RESTRICTED` indicates
11064 that such privilege is required.11065 [EROFS] The file referred to by *fildev* resides on a read-only file system.11066 The `fchown()` function may fail if:11067 [EINVAL] The owner or group ID is not a value supported by the implementation. The
11068 OB XSR *fildev* argument refers to a pipe or socket or an `fattach()`-ed `STREAM` and the
11069 implementation disallows execution of `fchown()` on a pipe.

11070 [EIO] A physical I/O error has occurred.

11071 [EINTR] The `fchown()` function was interrupted by a signal which was caught.11072 **EXAMPLES**11073 **Changing the Current Owner of a File**11074 The following example shows how to change the owner of a file named `/home/cnd/mod1` to
11075 “jones” and the group to “cnd”.11076 The numeric value for the user ID is obtained by extracting the user ID from the user database
11077 entry associated with “jones”. Similarly, the numeric value for the group ID is obtained by
11078 extracting the group ID from the group database entry associated with “cnd”. This example
11079 assumes the calling program has appropriate privileges.11080 `#include <sys/types.h>`
11081 `#include <unistd.h>`
11082 `#include <fcntl.h>`
11083 `#include <pwd.h>`
11084 `#include <grp.h>`

11085 `struct passwd *pwd;`
11086 `struct group *grp;`
11087 `int fildev;`
11088 `...`
11089 `fildev = open("/home/cnd/mod1", O_RDWR);`

```
11090     pwd = getpwnam("jones");
11091     grp = getgrnam("cnd");
11092     fchown(fildes, pwd->pw_uid, grp->gr_gid);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

chown(), the Base Definitions volume of IEEE Std 1003.1-200x, <unistd.h>

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The following changes were made to align with the IEEE P1003.1a draft standard:

- Clarification is added that a call to *fchown()* may not be allowed on a pipe.

The *fchown()* function is defined as mandatory.

Issue 7

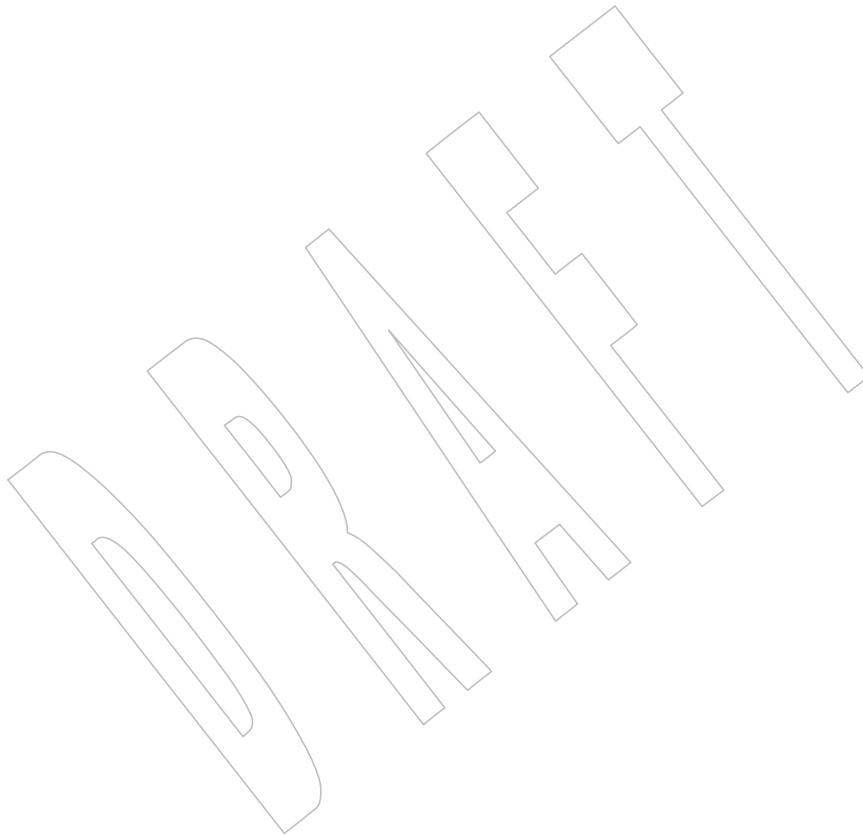
Functionality relating to XSI STREAMS is marked obsolescent.

11111 **NAME**
11112 fchownat — change owner and group of a file relative to directory file descriptor

11113 **SYNOPSIS**
11114 #include <unistd.h>

11115 int fchownat(int *fd*, const char **path*, uid_t *owner*, gid_t *group*,
11116 int *flag*);

11117 **DESCRIPTION**
11118 Refer to *chown()*.



11119 **NAME**11120 `fclose` — close a stream11121 **SYNOPSIS**11122 `#include <stdio.h>`11123 `int fclose(FILE *stream);`11124 **DESCRIPTION**

11125 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 11126 conflict between the requirements described here and the ISO C standard is unintentional. This
 11127 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11128 The `fclose()` function shall cause the stream pointed to by `stream` to be flushed and the associated
 11129 file to be closed. Any unwritten buffered data for the stream shall be written to the file; any
 11130 unread buffered data shall be discarded. Whether or not the call succeeds, the stream shall be
 11131 disassociated from the file and any buffer set by the `setbuf()` or `setvbuf()` function shall be
 11132 disassociated from the stream. If the associated buffer was automatically allocated, it shall be
 11133 deallocated.

11134 CX If the file is not already at EOF, and the file is one capable of seeking, the file offset of the
 11135 underlying open file description shall be adjusted so that the next operation on the open file
 11136 description deals with the byte after the last one read from or written to the stream being closed.

11137 The `fclose()` function shall mark for update the `st_ctime` and `st_mtime` fields of the underlying file,
 11138 if the stream was writable, and if buffered data remains that has not yet been written to the file.
 11139 The `fclose()` function shall perform the equivalent of a `close()` on the file descriptor that is
 11140 associated with the stream pointed to by `stream`.

11141 After the call to `fclose()`, any use of `stream` results in undefined behavior.

11142 **RETURN VALUE**

11143 CX Upon successful completion, `fclose()` shall return 0; otherwise, it shall return EOF and set `errno`
 11144 to indicate the error.

11145 **ERRORS**

11146 The `fclose()` function shall fail if:

11147 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying `stream` and
 11148 the thread would be delayed in the write operation.

11149 CX [EBADF] The file descriptor underlying stream is not valid.

11150 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size.

11151 XSI [EFBIG] An attempt was made to write a file that exceeds the file size limit of the
 11152 process.

11153 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
 11154 offset maximum associated with the corresponding stream.

11155 CX [EINTR] The `fclose()` function was interrupted by a signal.

11156 CX [EIO] The process is a member of a background process group attempting to write to
 11157 its controlling terminal, TOSTOP is set, the process is neither ignoring nor
 11158 blocking SIGTTOU, and the process group of the process is orphaned. This
 11159 error may also be returned under implementation-defined conditions.

fclose()

System Interfaces

11160 CX [ENOMEM] The underlying stream was created by *open_memstream()* or
11161 *open_wmemstream()* and insufficient memory is available.

11162 CX [ENOSPC] There was no free space remaining on the device containing the file or in the
11163 buffer used by the *fmemopen()* function.

11164 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
11165 any process. A SIGPIPE signal shall also be sent to the thread.

11166 The *fclose()* function may fail if:

11167 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
11168 capabilities of the device.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

close(), *fmemopen()*, *fopen()*, *getrlimit()*, *open_memstream()*, *ulimit()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EFBIG] error is added as part of the large file support extensions.
- The [ENXIO] optional error condition is added.

The DESCRIPTION is updated to note that the stream and any buffer are disassociated whether or not the call succeeds. This is for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/28 is applied, updating the [EAGAIN] error in the ERRORS section from “the process would be delayed” to “the thread would be delayed”.

Issue 7

Austin Group Interpretation 1003.1-2001 #002 is applied, clarifying the interaction of file descriptors and streams.

The [ENOSPC] error condition is updated and the [ENOMEM] error is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

11200 **NAME**

11201 fcntl — file control

11202 **SYNOPSIS**

11203 #include <fcntl.h>

11204 int fcntl(int *fil-des*, int *cmd*, ...);11205 **DESCRIPTION**11206 The *fcntl()* function shall perform the operations described below on open files. The *fil-des*
11207 argument is a file descriptor.11208 The available values for *cmd* are defined in <fcntl.h> and are as follows:

11209 **F_DUPFD** Return a new file descriptor which shall be the lowest numbered available
11210 (that is, not already open) file descriptor greater than or equal to the third
11211 argument, *arg*, taken as an integer of type **int**. The new file descriptor shall
11212 refer to the same open file description as the original file descriptor, and shall
11213 share any locks. The FD_CLOEXEC flag associated with the new file
11214 descriptor shall be cleared to keep the file open across calls to one of the *exec*
11215 functions.

11216 **F_GETFD** Get the file descriptor flags defined in <fcntl.h> that are associated with the
11217 file descriptor *fil-des*. File descriptor flags are associated with a single file
11218 descriptor and do not affect other file descriptors that refer to the same file.

11219 **F_SETFD** Set the file descriptor flags defined in <fcntl.h>, that are associated with *fil-des*,
11220 to the third argument, *arg*, taken as type **int**. If the FD_CLOEXEC flag in the
11221 third argument is 0, the file shall remain open across the *exec* functions;
11222 otherwise, the file shall be closed upon successful execution of one of the *exec*
11223 functions.

11224 **F_GETFL** Get the file status flags and file access modes, defined in <fcntl.h>, for the file
11225 description associated with *fil-des*. The file access modes can be extracted from
11226 the return value using the mask O_ACCMODE, which is defined in <fcntl.h>.
11227 File status flags and file access modes are associated with the file description
11228 and do not affect other file descriptors that refer to the same file with different
11229 open file descriptions.

11230 **F_SETFL** Set the file status flags, defined in <fcntl.h>, for the file description associated
11231 with *fil-des* from the corresponding bits in the third argument, *arg*, taken as
11232 type **int**. Bits corresponding to the file access mode and the file creation flags,
11233 as defined in <fcntl.h>, that are set in *arg* shall be ignored. If any bits in *arg*
11234 other than those mentioned here are changed by the application, the result is
11235 unspecified.

11236 **F_GETOWN** If *fil-des* refers to a socket, get the process or process group ID specified to
11237 receive SIGURG signals when out-of-band data is available. Positive values
11238 indicate a process ID; negative values, other than -1, indicate a process group
11239 ID. If *fil-des* does not refer to a socket, the results are unspecified.

11240 **F_SETOWN** If *fil-des* refers to a socket, set the process or process group ID specified to
11241 receive SIGURG signals when out-of-band data is available, using the value of
11242 the third argument, *arg*, taken as type **int**. Positive values indicate a process
11243 ID; negative values, other than -1, indicate a process group ID. If *fil-des* does
11244 not refer to a socket, the results are unspecified.

11245 The following values for *cmd* are available for advisory record locking. Record locking shall be

11246 supported for regular files, and may be supported for other files.

11247 **F_GETLK** Get the first lock which blocks the lock description pointed to by the third
11248 argument, *arg*, taken as a pointer to type **struct flock**, defined in **<fcntl.h>**.
11249 The information retrieved shall overwrite the information passed to *fcntl()* in
11250 the structure **flock**. If no lock is found that would prevent this lock from
11251 being created, then the structure shall be left unchanged except for the lock
11252 type which shall be set to **F_UNLCK**.

11253 **F_SETLK** Set or clear a file segment lock according to the lock description pointed to by
11254 the third argument, *arg*, taken as a pointer to type **struct flock**, defined in
11255 **<fcntl.h>**. **F_SETLK** can establish shared (or read) locks (**F_RDLCK**) or
11256 exclusive (or write) locks (**F_WRLCK**), as well as to remove either type of lock
11257 (**F_UNLCK**). **F_RDLCK**, **F_WRLCK**, and **F_UNLCK** are defined in **<fcntl.h>**.
11258 If a shared or exclusive lock cannot be set, *fcntl()* shall return immediately
11259 with a return value of **-1**.

11260 **F_SETLKW** This command shall be equivalent to **F_SETLK** except that if a shared or
11261 exclusive lock is blocked by other locks, the thread shall wait until the request
11262 can be satisfied. If a signal that is to be caught is received while *fcntl()* is
11263 waiting for a region, *fcntl()* shall be interrupted. Upon return from the signal
11264 handler, *fcntl()* shall return **-1** with *errno* set to **[EINTR]**, and the lock
11265 operation shall not be done.

11266 Additional implementation-defined values for *cmd* may be defined in **<fcntl.h>**. Their names
11267 shall start with **F_**.

11268 When a shared lock is set on a segment of a file, other processes shall be able to set shared locks
11269 on that segment or a portion of it. A shared lock prevents any other process from setting an
11270 exclusive lock on any portion of the protected area. A request for a shared lock shall fail if the
11271 file descriptor was not opened with read access.

11272 An exclusive lock shall prevent any other process from setting a shared lock or an exclusive lock
11273 on any portion of the protected area. A request for an exclusive lock shall fail if the file
11274 descriptor was not opened with write access.

11275 The structure **flock** describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*),
11276 size (*l_len*), and process ID (*l_pid*) of the segment of the file to be affected.

11277 The value of *l_whence* is **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**, to indicate that the relative
11278 offset *l_start* bytes shall be measured from the start of the file, current position, or end of the file,
11279 respectively. The value of *l_len* is the number of consecutive bytes to be locked. The value of *l_len*
11280 may be negative (where the definition of **off_t** permits negative values of *l_len*). The *l_pid* field
11281 is only used with **F_GETLK** to return the process ID of the process holding a blocking lock. After
11282 a successful **F_GETLK** request, when a blocking lock is found, the values returned in the **flock**
11283 structure shall be as follows:

11284 *l_type* Type of blocking lock found.

11285 *l_whence* **SEEK_SET**.

11286 *l_start* Start of the blocking lock.

11287 *l_len* Length of the blocking lock.

11288 *l_pid* Process ID of the process that holds the blocking lock.

11289 If the command is **F_SETLKW** and the process must wait for another process to release a lock,
11290 then the range of bytes to be locked shall be determined before the *fcntl()* function blocks. If the
11291 file size or file descriptor seek offset change while *fcntl()* is blocked, this shall not affect the
11292 range of bytes locked.

11293 If *l_len* is positive, the area affected shall start at *l_start* and end at *l_start+l_len-1*. If *l_len* is
 11294 negative, the area affected shall start at *l_start+l_len* and end at *l_start-1*. Locks may start and
 11295 extend beyond the current end of a file, but shall not extend before the beginning of the file. A
 11296 lock shall be set to extend to the largest possible value of the file offset for that file by setting
 11297 *l_len* to 0. If such a lock also has *l_start* set to 0 and *l_whence* is set to `SEEK_SET`, the whole file
 11298 shall be locked.

11299 There shall be at most one type of lock set for each byte in the file. Before a successful return
 11300 from an `F_SETLK` or an `F_SETLKW` request when the calling process has previously existing
 11301 locks on bytes in the region specified by the request, the previous lock type for each byte in the
 11302 specified region shall be replaced by the new lock type. As specified above under the
 11303 descriptions of shared locks and exclusive locks, an `F_SETLK` or an `F_SETLKW` request
 11304 (respectively) shall fail or block when another process has existing locks on bytes in the specified
 11305 region and the type of any of those locks conflicts with the type specified in the request.

11306 All locks associated with a file for a given process shall be removed when a file descriptor for
 11307 that file is closed by that process or the process holding that file descriptor terminates. Locks are
 11308 not inherited by a child process.

11309 A potential for deadlock occurs if a process controlling a locked region is put to sleep by
 11310 attempting to lock the locked region of another process. If the system detects that sleeping until
 11311 a locked region is unlocked would cause a deadlock, `fcntl()` shall fail with an `[EDEADLK]` error.

11312 An unlock (`F_UNLCK`) request in which *l_len* is non-zero and the offset of the last byte of the
 11313 requested segment is the maximum value for an object of type `off_t`, when the process has an
 11314 existing lock in which *l_len* is 0 and which includes the last byte of the requested segment, shall
 11315 be treated as a request to unlock from the start of the requested segment with an *l_len* equal to 0.
 11316 Otherwise, an unlock (`F_UNLCK`) request shall attempt to unlock only the requested segment.

11317 SHM When the file descriptor *fildev* refers to a shared memory object, the behavior of `fcntl()` shall be
 11318 the same as for a regular file except the effect of the following values for the argument *cmd* shall
 11319 be unspecified: `F_SETFL`, `F_GETLK`, `F_SETLK`, and `F_SETLKW`.

11320 TYM If *fildev* refers to a typed memory object, the result of the `fcntl()` function is unspecified.

11321 RETURN VALUE

11322 Upon successful completion, the value returned shall depend on *cmd* as follows:

11323	<code>F_DUPFD</code>	A new file descriptor.
11324	<code>F_GETFD</code>	Value of flags defined in <code><fcntl.h></code> . The return value shall not be negative.
11325	<code>F_SETFD</code>	Value other than <code>-1</code> .
11326	<code>F_GETFL</code>	Value of file status flags and access modes. The return value is not negative.
11327	<code>F_SETFL</code>	Value other than <code>-1</code> .
11328	<code>F_GETLK</code>	Value other than <code>-1</code> .
11329	<code>F_SETLK</code>	Value other than <code>-1</code> .
11330	<code>F_SETLKW</code>	Value other than <code>-1</code> .
11331	<code>F_GETOWN</code>	Value of the socket owner process or process group; this will not be <code>-1</code> .
11332	<code>F_SETOWN</code>	Value other than <code>-1</code> .
11333		Otherwise, <code>-1</code> shall be returned and <i>errno</i> set to indicate the error.

11334 **ERRORS**11335 The *fcntl()* function shall fail if:

11336 [EACCES] or [EAGAIN]

11337 The *cmd* argument is F_SETLK; the type of lock (*l_type*) is a shared (F_RDLCK)
11338 or exclusive (F_WRLCK) lock and the segment of a file to be locked is already
11339 exclusive-locked by another process, or the type is an exclusive lock and some
11340 portion of the segment of a file to be locked is already shared-locked or
11341 exclusive-locked by another process.

11342 [EBADF]

11343 The *fildev* argument is not a valid open file descriptor, or the argument *cmd* is
11344 F_SETLK or F_SETLKW, the type of lock, *l_type*, is a shared lock (F_RDLCK),
11345 and *fildev* is not a valid file descriptor open for reading, or the type of lock,
11346 *l_type*, is an exclusive lock (F_WRLCK), and *fildev* is not a valid file descriptor
open for writing.

11347 [EINTR]

11348 The *cmd* argument is F_SETLKW and the function was interrupted by a signal.

11349 [EINVAL]

11350 The *cmd* argument is invalid, or the *cmd* argument is F_DUPFD and *arg* is
11351 negative or greater than or equal to {OPEN_MAX}, or the *cmd* argument is
F_GETLK, F_SETLK, or F_SETLKW and the data pointed to by *arg* is not
valid, or *fildev* refers to a file that does not support locking.

11352 [EMFILE]

11353 The argument *cmd* is F_DUPFD and all file descriptors available to the process
11354 are currently open, or no file descriptors greater than or equal to *arg* are
available.

11355 [ENOLCK]

11356 The argument *cmd* is F_SETLK or F_SETLKW and satisfying the lock or unlock
11357 request would result in the number of locked regions in the system exceeding
a system-imposed limit.

11358 [EOVERFLOW]

11359 One of the values to be returned cannot be represented correctly.

11360 [EOVERFLOW]

11361 The *cmd* argument is F_GETLK, F_SETLK, or F_SETLKW and the smallest or,
if *l_len* is non-zero, the largest offset of any byte in the requested segment
cannot be represented correctly in an object of type *off_t*.11362 The *fcntl()* function may fail if:

11363 [EDEADLK]

11364 The *cmd* argument is F_SETLKW, the lock is blocked by a lock from another
11365 process, and putting the calling process to sleep to wait for that lock to become
free would cause a deadlock.11366 **EXAMPLES**11367 **Locking and Unlocking a File**11368 The following example demonstrates how to place a lock on bytes 100 to 109 of a file and then
11369 later remove it. F_SETLK is used to perform a non-blocking lock request so that the process does
11370 not have to wait if an incompatible lock is held by another process; instead the process can take
11371 some other action.11372 #include <stdlib.h>
11373 #include <unistd.h>
11374 #include <fcntl.h>
11375 #include <errno.h>11376 int
11377 main(int argc, char *argv[])
11378 {
11379 int fd;

```

11380     struct flock fl;
11381     fd = open("testfile", O_RDWR);
11382     if (fd == -1)
11383         /* Handle error */;
11384     /* Make a non-blocking request to place a write lock
11385        on bytes 100-109 of testfile */
11386     fl.l_type = F_WRLCK;
11387     fl.l_whence = SEEK_SET;
11388     fl.l_start = 100;
11389     fl.l_len = 10;
11390     if (fcntl(fd, F_SETLK, &fl) == -1) {
11391         if (errno == EACCES || errno == EAGAIN) {
11392             printf("Already locked by another process\n");
11393             /* We can't get the lock at the moment */
11394         } else {
11395             /* Handle unexpected error */;
11396         }
11397     } else { /* Lock was granted... */
11398         /* Perform I/O on bytes 100 to 109 of file */
11399         /* Unlock the locked bytes */
11400         fl.l_type = F_UNLCK;
11401         fl.l_whence = SEEK_SET;
11402         fl.l_start = 100;
11403         fl.l_len = 10;
11404         if (fcntl(fd, F_SETLK, &fl) == -1)
11405             /* Handle error */;
11406     }
11407     exit(EXIT_SUCCESS);
11408 } /* main */

```

Setting the Close-on-Exec Flag

The following example demonstrates how to set the close-on-exec flag for the file descriptor *fd*.

```

11411 #include <unistd.h>
11412 #include <fcntl.h>
11413 ...
11414     int flags;
11415     flags = fcntl(fd, F_GETFD);
11416     if (flags == -1)
11417         /* Handle error */;
11418     flags |= FD_CLOEXEC;
11419     if (fcntl(fd, F_SETFD, flags) == -1)
11420         /* Handle error */;

```

APPLICATION USAGE

None.

RATIONALE

- 11423
11424 The ellipsis in the SYNOPSIS is the syntax specified by the ISO C standard for a variable number
11425 of arguments. It is used because System V uses pointers for the implementation of file locking
11426 functions.
- 11427 The *arg* values to F_GETFD, F_SETFD, F_GETFL, and F_SETFL all represent flag values to allow
11428 for future growth. Applications using these functions should do a read-modify-write operation
11429 on them, rather than assuming that only the values defined by this volume of
11430 IEEE Std 1003.1-200x are valid. It is a common error to forget this, particularly in the case of
11431 F_SETFD.
- 11432 This volume of IEEE Std 1003.1-200x permits concurrent read and write access to file data using
11433 the *fcntl()* function; this is a change from the 1984 /usr/group standard and early proposals.
11434 Without concurrency controls, this feature may not be fully utilized without occasional loss of
11435 data.
- 11436 Data losses occur in several ways. One case occurs when several processes try to update the
11437 same record, without sequencing controls; several updates may occur in parallel and the last
11438 writer “wins”. Another case is a bit-tree or other internal list-based database that is undergoing
11439 reorganization. Without exclusive use to the tree segment by the updating process, other reading
11440 processes chance getting lost in the database when the index blocks are split, condensed,
11441 inserted, or deleted. While *fcntl()* is useful for many applications, it is not intended to be overly
11442 general and does not handle the bit-tree example well.
- 11443 This facility is only required for regular files because it is not appropriate for many devices such
11444 as terminals and network connections.
- 11445 Since *fcntl()* works with “any file descriptor associated with that file, however it is obtained”,
11446 the file descriptor may have been inherited through a *fork()* or *exec* operation and thus may
11447 affect a file that another process also has open.
- 11448 The use of the open file description to identify what to lock requires extra calls and presents
11449 problems if several processes are sharing an open file description, but there are too many
11450 implementations of the existing mechanism for this volume of IEEE Std 1003.1-200x to use
11451 different specifications.
- 11452 Another consequence of this model is that closing any file descriptor for a given file (whether or
11453 not it is the same open file description that created the lock) causes the locks on that file to be
11454 relinquished for that process. Equivalently, any *close* for any file/process pair relinquishes the
11455 locks owned on that file for that process. But note that while an open file description may be
11456 shared through *fork()*, locks are not inherited through *fork()*. Yet locks may be inherited through
11457 one of the *exec* functions.
- 11458 The identification of a machine in a network environment is outside the scope of this volume of
11459 IEEE Std 1003.1-200x. Thus, an *l_sysid* member, such as found in System V, is not included in the
11460 locking structure.
- 11461 Changing of lock types can result in a previously locked region being split into smaller regions.
- 11462 Mandatory locking was a major feature of the 1984 /usr/group standard.
- 11463 For advisory file record locking to be effective, all processes that have access to a file must
11464 cooperate and use the advisory mechanism before doing I/O on the file. Enforcement-mode
11465 record locking is important when it cannot be assumed that all processes are cooperating. For
11466 example, if one user uses an editor to update a file at the same time that a second user executes
11467 another process that updates the same file and if only one of the two processes is using advisory
11468 locking, the processes are not cooperating. Enforcement-mode record locking would protect
11469 against accidental collisions.
- 11470 Secondly, advisory record locking requires a process using locking to bracket each I/O operation

11471 with lock (or test) and unlock operations. With enforcement-mode file and record locking, a
 11472 process can lock the file once and unlock when all I/O operations have been completed.
 11473 Enforcement-mode record locking provides a base that can be enhanced; for example, with
 11474 sharable locks. That is, the mechanism could be enhanced to allow a process to lock a file so
 11475 other processes could read it, but none of them could write it.

11476 Mandatory locks were omitted for several reasons:

- 11477 1. Mandatory lock setting was done by multiplexing the set-group-ID bit in most
 11478 implementations; this was confusing, at best.
- 11479 2. The relationship to file truncation as supported in 4.2 BSD was not well specified.
- 11480 3. Any publicly readable file could be locked by anyone. Many historical implementations
 11481 keep the password database in a publicly readable file. A malicious user could thus
 11482 prohibit logins. Another possibility would be to hold open a long-distance telephone line.
- 11483 4. Some demand-paged historical implementations offer memory mapped files, and
 11484 enforcement cannot be done on that type of file.

11485 Since sleeping on a region is interrupted with any signal, *alarm()* may be used to provide a
 11486 timeout facility in applications requiring it. This is useful in deadlock detection. Since
 11487 implementation of full deadlock detection is not always feasible, the [EDEADLK] error was
 11488 made optional.

11489 FUTURE DIRECTIONS

11490 None.

11491 SEE ALSO

11492 *alarm()*, *close()*, *exec*, *open()*, *sigaction()*, the Base Definitions volume of IEEE Std 1003.1-200x,
 11493 *<fcntl.h>*, *<signal.h>*

11494 CHANGE HISTORY

11495 First released in Issue 1. Derived from Issue 1 of the SVID.

11496 Issue 5

11497 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
 11498 Threads Extension.

11499 Large File Summit extensions are added.

11500 Issue 6

11501 In the SYNOPSIS, the optional include of the *<sys/types.h>* header is removed.

11502 The following new requirements on POSIX implementations derive from alignment with the
 11503 Single UNIX Specification:

- 11504 • The requirement to include *<sys/types.h>* has been removed. Although *<sys/types.h>* was
 11505 required for conforming implementations of previous POSIX specifications, it was not
 11506 required for UNIX applications.
- 11507 • In the DESCRIPTION, sentences describing behavior when *l_len* is negative are now
 11508 mandated, and the description of unlock (F_UNLOCK) when *l_len* is non-negative is
 11509 mandated.
- 11510 • In the ERRORS section, the [EINVAL] error condition has the case mandated when the *cmd*
 11511 is invalid, and two [EOVERFLOW] error conditions are added.

11512 The F_GETOWN and F_SETOWN values are added for sockets.

11513 The following changes were made to align with the IEEE P1003.1a draft standard:

11514
11515

- Clarification is added that the extent of the bytes locked is determined prior to the blocking action.

11516
11517

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that *fcntl()* results are unspecified for typed memory objects.

11518

The normative text is updated to avoid use of the term “must” for application requirements.

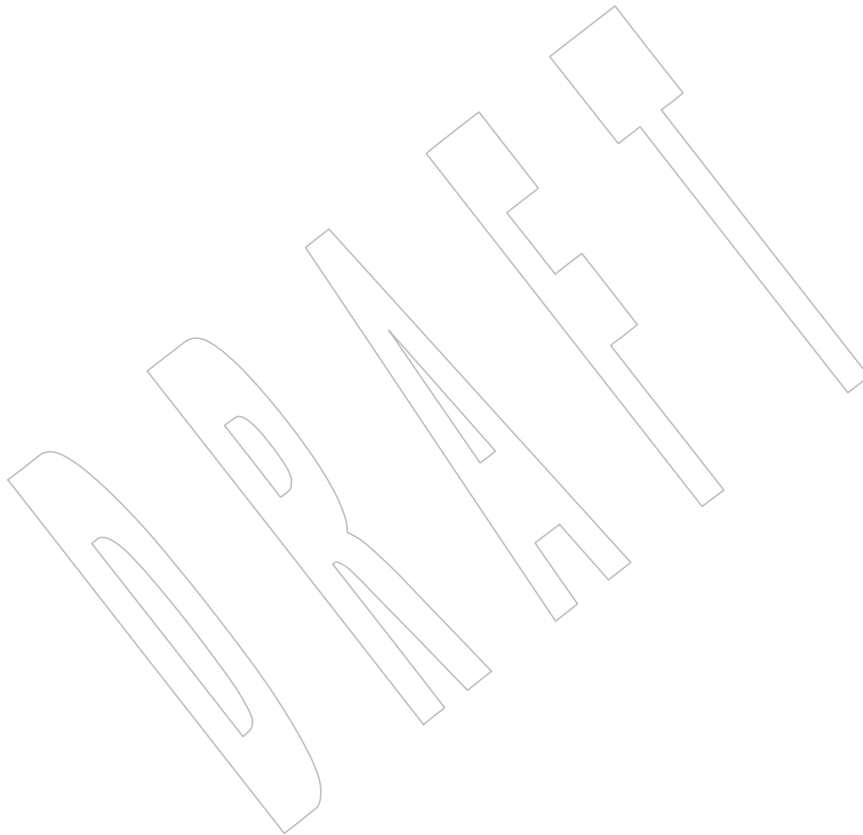
11519
11520

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/29 is applied, adding the example to the EXAMPLES section.

11521

Issue 711522
11523

The optional `<unistd.h>` header is removed from this function, since `<fcntl.h>` now defines `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` as part of the Base.



11524 **NAME**
 11525 `fdatasync` — synchronize the data of a file (**REALTIME**)

11526 **SYNOPSIS**

11527 SIO `#include <unistd.h>`
 11528 `int fdatasync(int fildes);`

11529 **DESCRIPTION**

11530 The `fdatasync()` function shall force all currently queued I/O operations associated with the file
 11531 indicated by file descriptor `fil`des to the synchronized I/O completion state.

11532 The functionality shall be equivalent to `fsync()` with the symbol `_POSIX_SYNCHRONIZED_IO`
 11533 defined, with the exception that all I/O operations shall be completed as defined for
 11534 synchronized I/O data integrity completion.

11535 **RETURN VALUE**

11536 If successful, the `fdatasync()` function shall return the value 0; otherwise, the function shall return
 11537 the value `-1` and set `errno` to indicate the error. If the `fdatasync()` function fails, outstanding I/O
 11538 operations are not guaranteed to have been completed.

11539 **ERRORS**

11540 The `fdatasync()` function shall fail if:

11541 [EBADF] The `fil`des argument is not a valid file descriptor open for writing.

11542 [EINVAL] This implementation does not support synchronized I/O for this file.

11543 In the event that any of the queued I/O operations fail, `fdatasync()` shall return the error
 11544 conditions defined for `read()` and `write()`.

11545 **EXAMPLES**

11546 None.

11547 **APPLICATION USAGE**

11548 None.

11549 **RATIONALE**

11550 None.

11551 **FUTURE DIRECTIONS**

11552 None.

11553 **SEE ALSO**

11554 `aio_fsync()`, `fcntl()`, `fsync()`, `open()`, `read()`, `write()`, the Base Definitions volume of
 11555 IEEE Std 1003.1-200x, `<unistd.h>`

11556 **CHANGE HISTORY**

11557 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

11558 **Issue 6**

11559 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 11560 implementation does not support the Synchronized Input and Output option.

11561 The `fdatasync()` function is marked as part of the Synchronized Input and Output option.

11562 **NAME**11563 `fdetach` — detach a name from a STREAMS-based file descriptor (**STREAMS**)11564 **SYNOPSIS**

```
11565 OB XSR #include <stropts.h>
11566 int fdetach(const char *path);
```

11567 **DESCRIPTION**

11568 The `fdetach()` function shall detach a STREAMS-based file from the file to which it was attached
 11569 by a previous call to `fattach()`. The `path` argument points to the pathname of the attached
 11570 STREAMS file. The process shall have appropriate privileges or be the owner of the file. A
 11571 successful call to `fdetach()` shall cause all pathnames that named the attached STREAMS file to
 11572 again name the file to which the STREAMS file was attached. All subsequent operations on `path`
 11573 shall operate on the underlying file and not on the STREAMS file.

11574 All open file descriptions established while the STREAMS file was attached to the file referenced
 11575 by `path` shall still refer to the STREAMS file after the `fdetach()` has taken effect.

11576 If there are no open file descriptors or other references to the STREAMS file, then a successful
 11577 call to `fdetach()` shall be equivalent to performing the last `close()` on the attached file.

11578 **RETURN VALUE**

11579 Upon successful completion, `fdetach()` shall return 0; otherwise, it shall return `-1` and set `errno` to
 11580 indicate the error.

11581 **ERRORS**

11582 The `fdetach()` function shall fail if:

- | | | |
|-------|----------------|--|
| 11583 | [EACCES] | Search permission is denied on a component of the path prefix. |
| 11584 | [EINVAL] | The <code>path</code> argument names a file that is not currently attached. |
| 11585 | [ELOOP] | A loop exists in symbolic links encountered during resolution of the <code>path</code>
11586 argument. |
| 11587 | [ENAMETOOLONG] | |
| 11588 | | The size of a pathname exceeds {PATH_MAX} or a pathname component is
11589 longer than {NAME_MAX}. |
| 11590 | [ENOENT] | A component of <code>path</code> does not name an existing file or <code>path</code> is an empty string. |
| 11591 | [ENOTDIR] | A component of the path prefix is not a directory. |
| 11592 | [EPERM] | The effective user ID is not the owner of <code>path</code> and the process does not have
11593 appropriate privileges. |

11594 The `fdetach()` function may fail if:

- | | | |
|-------|----------------|---|
| 11595 | [ELOOP] | More than {SYMLOOP_MAX} symbolic links were encountered during
11596 resolution of the <code>path</code> argument. |
| 11597 | [ENAMETOOLONG] | |
| 11598 | | Pathname resolution of a symbolic link produced an intermediate result
11599 whose length exceeds {PATH_MAX}. |

11600 **EXAMPLES**11601 **Detaching a File**

11602 The following example detaches the STREAMS-based file **/tmp/named-STREAM** from the file to
 11603 which it was attached by a previous, successful call to *fattach()*. Subsequent calls to open this
 11604 file refer to the underlying file, not to the STREAMS file.

```
11605 #include <stropts.h>
11606 ...
11607     char *filename = "/tmp/named-STREAM";
11608     int ret;
11609
11609     ret = fdetach(filename);
```

11610 **APPLICATION USAGE**

11611 None.

11612 **RATIONALE**

11613 None.

11614 **FUTURE DIRECTIONS**

11615 The *fdetach()* function may be removed in a future version.

11616 **SEE ALSO**

11617 *fattach()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**

11618 **CHANGE HISTORY**

11619 First released in Issue 4, Version 2.

11620 **Issue 5**

11621 Moved from X/OPEN UNIX extension to BASE.

11622 **Issue 6**

11623 The normative text is updated to avoid use of the term “must” for application requirements.

11624 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
 11625 [ELOOP] error condition is added.

11626 **Issue 7**

11627 The *fdetach()* function is marked obsolescent.

11628 **NAME**11629 `fdim, fdimf, fdiml` — compute positive difference between two floating-point numbers11630 **SYNOPSIS**11631 `#include <math.h>`11632 `double fdim(double x, double y);`11633 `float fdimf(float x, float y);`11634 `long double fdiml(long double x, long double y);`11635 **DESCRIPTION**11636 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11637 conflict between the requirements described here and the ISO C standard is unintentional. This
11638 volume of IEEE Std 1003.1-200x defers to the ISO C standard.11639 These functions shall determine the positive difference between their arguments. If x is greater
11640 than y , $x-y$ is returned. If x is less than or equal to y , $+0$ is returned.11641 An application wishing to check for error situations should set *errno* to zero and call
11642 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
11643 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
11644 zero, an error has occurred.11645 **RETURN VALUE**

11646 Upon successful completion, these functions shall return the positive difference value.

11647 If $x-y$ is positive and overflows, a range error shall occur and *fdim()*, *fdimf()*, and *fdiml()* shall
11648 return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.11649 XSI If $x-y$ is positive and underflows, a range error may occur, and either $(x-y)$ (if representable), or
11650 0.0 (if supported), or an implementation-defined value shall be returned.11651 MX If x or y is NaN, a NaN shall be returned.11652 **ERRORS**11653 The *fdim()* function shall fail if:

11654 Range Error The result overflows.

11655 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
11656 then *errno* shall be set to [ERANGE]. If the integer expression
11657 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
11658 floating-point exception shall be raised.11659 The *fdim()* function may fail if:

11660 Range Error The result underflows.

11661 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
11662 then *errno* shall be set to [ERANGE]. If the integer expression
11663 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
11664 floating-point exception shall be raised.

11665
11666
11667
11668
11669
11670
11671
11672
11673
11674
11675
11676
11677
11678
11679
11680
11681

EXAMPLES

None.

APPLICATION USAGE

On implementations supporting IEEE Std 754-1985, $x-y$ cannot underflow, and hence the 0.0 return value is shaded as an extension for implementations supporting the XSI option rather than an MX extension.

On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

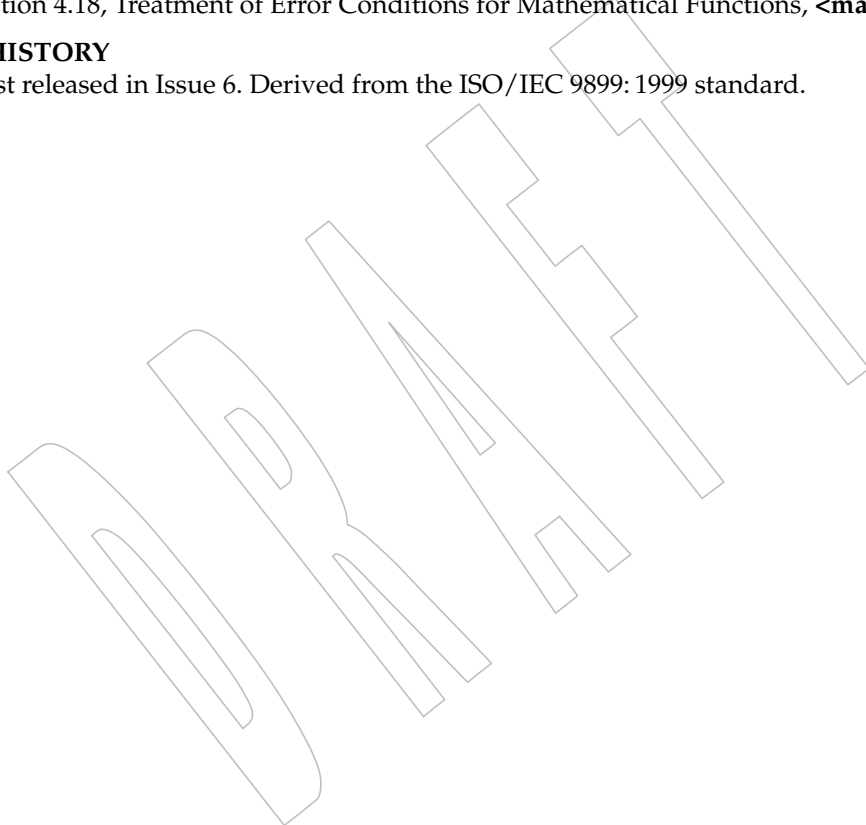
None.

SEE ALSO

feclearexcept(), *fetestexcept()*, *fmax()*, *fmin()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.



11682 **NAME**
 11683 `fdopen` — associate a stream with a file descriptor

11684 **SYNOPSIS**

11685 CX `#include <stdio.h>`
 11686 `FILE *fdopen(int fil-des, const char *mode);`

11687 **DESCRIPTION**

11688 The `fdopen()` function shall associate a stream with a file descriptor.

11689 The *mode* argument is a character string having one of the following values:

11690 *r* or *rb* Open a file for reading.
 11691 *w* or *wb* Open a file for writing.
 11692 *a* or *ab* Open a file for writing at end-of-file.
 11693 *r+* or *rb+* or *r+b* Open a file for update (reading and writing).
 11694 *w+* or *wb+* or *w+b* Open a file for update (reading and writing).
 11695 *a+* or *ab+* or *a+b* Open a file for update (reading and writing) at end-of-file.

11696 The meaning of these flags is exactly as specified in `open()`, except that modes beginning with *w*
 11697 shall not cause truncation of the file.

11698 Additional values for the *mode* argument may be supported by an implementation.

11699 The application shall ensure that the mode of the stream as expressed by the *mode* argument is
 11700 allowed by the file access mode of the open file description to which *fil-des* refers. The file
 11701 position indicator associated with the new stream is set to the position indicated by the file offset
 11702 associated with the file descriptor.

11703 The error and end-of-file indicators for the stream shall be cleared. The `fdopen()` function may
 11704 cause the *st_atime* field of the underlying file to be marked for update.

11705 SHM If *fil-des* refers to a shared memory object, the result of the `fdopen()` function is unspecified.

11706 TYM If *fil-des* refers to a typed memory object, the result of the `fdopen()` function is unspecified.

11707 The `fdopen()` function shall preserve the offset maximum previously set for the open file
 11708 description corresponding to *fil-des*.

11709 **RETURN VALUE**

11710 Upon successful completion, `fdopen()` shall return a pointer to a stream; otherwise, a null pointer
 11711 shall be returned and *errno* set to indicate the error.

11712 **ERRORS**

11713 The `fdopen()` function may fail if:

11714 [EBADF] The *fil-des* argument is not a valid file descriptor.
 11715 [EINVAL] The *mode* argument is not a valid mode.
 11716 [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.
 11717 [EMFILE] {STREAM_MAX} streams are currently open in the calling process.
 11718 [ENOMEM] Insufficient space to allocate a buffer.

EXAMPLES

None.

APPLICATION USAGE

File descriptors are obtained from calls like *open()*, *dup()*, *creat()*, or *pipe()*, which open files but do not return streams.

RATIONALE

The file descriptor may have been obtained from *open()*, *creat()*, *pipe()*, *dup()*, *fcntl()*, or *socket()*; inherited through *fork()*, *posix_spawn()*, or *exec*; or perhaps obtained by other means.

The meanings of the *mode* arguments of *fdopen()* and *fopen()* differ. With *fdopen()*, open for write (*w* or *w+*) does not truncate, and append (*a* or *a+*) cannot create for writing. The *mode* argument formats that include *a b* are allowed for consistency with the ISO C standard function *fopen()*. The *b* has no effect on the resulting stream. Although not explicitly required by this volume of IEEE Std 1003.1-200x, a good implementation of append (*a*) mode would cause the *O_APPEND* flag to be set.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.5.1 (on page 35), *fclose()*, *fmemopen()*, *fopen()*, *open()*, *open_memstream()*, *posix_spawn()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

Large File Summit extensions are added.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the use and setting of the *mode* argument are changed to include binary streams.
- In the DESCRIPTION, text is added for large file support to indicate setting of the offset maximum in the open file description.
- All errors identified in the ERRORS section are added.
- In the DESCRIPTION, text is added that the *fdopen()* function may cause *st_atime* to be updated.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Clarification is added that it is the responsibility of the application to ensure that the mode is compatible with the open file descriptor.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that *fdopen()* results are unspecified for typed memory objects.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/30 is applied, making corrections to the RATIONALE.

11760 **NAME**

11761 fdopendir, opendir — open directory associated with file descriptor

11762 **SYNOPSIS**

```
11763 #include <dirent.h>
11764 DIR *fdopendir(int fd);
11765 DIR *opendir(const char *dirname);
```

11766 **DESCRIPTION**

11767 The *fdopendir()* function shall be equivalent to the *opendir()* function except that the directory is
 11768 specified by a file descriptor rather than by a name. The file offset associated with the file
 11769 descriptor at the time of the call determines which entries are returned.

11770 Upon successful return from *fdopendir()*, the file descriptor is under the control of the system,
 11771 and if any attempt is made to close the file descriptor, or to modify the state of the associated
 11772 description other than by means of *closedir()*, *readdir()*, *readdir_r()*, or *rewinddir()*, the behavior is
 11773 implementation-defined. Upon calling *closedir()* the file descriptor shall be closed.

11774 It is unspecified whether the FD_CLOEXEC flag will be set on the file descriptor by a successful
 11775 call to *fdopendir()*.

11776 The *opendir()* function shall open a directory stream corresponding to the directory named by
 11777 the *dirname* argument. The directory stream is positioned at the first entry. If the type **DIR** is
 11778 implemented using a file descriptor, applications shall only be able to open up to a total of
 11779 {OPEN_MAX} files and directories.

11780 If the type **DIR** is implemented using a file descriptor, the descriptor shall be obtained as if the
 11781 O_DIRECTORY flag was passed to *open()*.

11782 **RETURN VALUE**

11783 Upon successful completion, these functions shall return a pointer to an object of type **DIR**.
 11784 Otherwise, these functions shall return a null pointer and set *errno* to indicate the error.

11785 **ERRORS**

11786 The *fdopendir()* function shall fail if:

- 11787 [EBADF] The *fd* argument is not a valid file descriptor open for searching.
- 11788 [ENOTDIR] The descriptor *fd* is not associated with a directory.

11789 The *opendir()* function shall fail if:

- 11790 [EACCES] Search permission is denied for the component of the path prefix of *dirname* or
 11791 read permission is denied for *dirname*.
- 11792 [ELOOP] A loop exists in symbolic links encountered during resolution of the *dirname*
 11793 argument.
- 11794 [ENAMETOOLONG]
 11795 The length of the *dirname* argument exceeds {PATH_MAX} or a pathname
 11796 component is longer than {NAME_MAX}.
- 11797 [ENOENT] A component of *dirname* does not name an existing directory or *dirname* is an
 11798 empty string.
- 11799 [ENOTDIR] A component of *dirname* is not a directory.

- 11800 The *opendir()* function may fail if:
- 11801 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
11802 resolution of the *dirname* argument.
- 11803 [EMFILE] All file descriptors available to the process are currently open.
- 11804 [ENAMETOOLONG]
11805 As a result of encountering a symbolic link in resolution of the *dirname*
11806 argument, the length of the substituted pathname string exceeded
11807 {PATH_MAX}.
- 11808 [ENFILE] Too many files are currently open in the system.

11809 EXAMPLES

11810 Open a Directory Stream

11811 The following program fragment demonstrates how the *opendir()* function is used.

```
11812 #include <sys/types.h>
11813 #include <dirent.h>
11814 #include <libgen.h>
11815 ...
11816     DIR *dir;
11817     struct dirent *dp;
11818 ...
11819     if ((dir = opendir(".")) == NULL) {
11820         perror ("Cannot open .");
11821         exit (1);
11822     }
11823     while ((dp = readdir (dir)) != NULL) {
11824         ...
```

11825 APPLICATION USAGE

11826 The *opendir()* function should be used in conjunction with *readdir()*, *closedir()*, and *rewinddir()* to
11827 examine the contents of the directory (see the EXAMPLES section in *readdir()*). This method is
11828 recommended for portability.

11829 RATIONALE

11830 The purpose of the *fdopendir()* function is to enable opening files in directories other than the
11831 current working directory without exposure to race conditions. Any part of the path of a file
11832 could be changed in parallel to a call to *opendir()*, resulting in unspecified behavior.

11833 Based on historical implementations, the rules about file descriptors apply to directory streams
11834 as well. However, this volume of IEEE Std 1003.1-200x does not mandate that the directory
11835 stream be implemented using file descriptors. The description of *closedir()* clarifies that if a file
11836 descriptor is used for the directory stream, it is mandatory that *closedir()* deallocate the file
11837 descriptor. When a file descriptor is used to implement the directory stream, it behaves as if the
11838 FD_CLOEXEC had been set for the file descriptor.

11839 The directory entries for dot and dot-dot are optional. This volume of IEEE Std 1003.1-200x does
11840 not provide a way to test *a priori* for their existence because an application that is portable must
11841 be written to look for (and usually ignore) those entries. Writing code that presumes that they
11842 are the first two entries does not always work, as many implementations permit them to be
11843 other than the first two entries, with a “normal” entry preceding them. There is negligible value
11844 in providing a way to determine what the implementation does because the code to deal with
11845 dot and dot-dot must be written in any case and because such a flag would add to the list of
11846 those flags (which has proven in itself to be objectionable) and might be abused.

11847 Since the structure and buffer allocation, if any, for directory operations are defined by the
 11848 implementation, this volume of IEEE Std 1003.1-200x imposes no portability requirements for
 11849 erroneous program constructs, erroneous data, or the use of unspecified values such as the use
 11850 or referencing of a *dirp* value or a **dirent** structure value after a directory stream has been closed
 11851 or after a *fork()* or one of the *exec* function calls.

FUTURE DIRECTIONS

11852 None.
 11853

SEE ALSO

11854 *closedir()*, *dirfd()*, *fstatat()*, *open()*, *readdir()*, *rewinddir()*, *symlink()*, the Base Definitions volume
 11855 of IEEE Std 1003.1-200x, **<dirent.h>**, **<limits.h>**, **<sys/types.h>**
 11856

CHANGE HISTORY

11857 First released in Issue 2.
 11858

Issue 6

11859 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.
 11860

11861 The following new requirements on POSIX implementations derive from alignment with the
 11862 Single UNIX Specification:

- 11863 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
 11864 required for conforming implementations of previous POSIX specifications, it was not
 11865 required for UNIX applications.
- 11866 • The [ELOOP] mandatory error condition is added.
- 11867 • A second [ENAMETOOLONG] is added as an optional error condition.

11868 The following changes were made to align with the IEEE P1003.1a draft standard:

- 11869 • The [ELOOP] optional error condition is added.
 11870

Issue 7

11871 SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

11872 The *fdopendir()* function is added from The Open Group Technical Standard, 2006, Extended API
 11873 Set Part 2.

11874 **NAME**
 11875 `feclearexcept` — clear floating-point exception

11876 **SYNOPSIS**
 11877 `#include <fenv.h>`
 11878 `int feclearexcept(int excepts);`

11879 **DESCRIPTION**
 11880 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 11881 conflict between the requirements described here and the ISO C standard is unintentional. This
 11882 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11883 The `feclearexcept()` function shall attempt to clear the supported floating-point exceptions
 11884 represented by *excepts*.

11885 **RETURN VALUE**
 11886 If the argument is zero or if all the specified exceptions were successfully cleared, `feclearexcept()`
 11887 shall return zero. Otherwise, it shall return a non-zero value.

11888 **ERRORS**
 11889 No errors are defined.

11890 **EXAMPLES**
 11891 None.

11892 **APPLICATION USAGE**
 11893 None.

11894 **RATIONALE**
 11895 None.

11896 **FUTURE DIRECTIONS**
 11897 None.

11898 **SEE ALSO**
 11899 `fegetexceptflag()`, `feraiseexcept()`, `fesetexceptflag()`, `fetestexcept()`, the Base Definitions volume of
 11900 IEEE Std 1003.1-200x, `<fenv.h>`

11901 **CHANGE HISTORY**
 11902 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.
 11903 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

11904 **NAME**
 11905 `fegetenv, fesetenv` — get and set current floating-point environment

11906 **SYNOPSIS**
 11907 `#include <fenv.h>`
 11908 `int fegetenv(fenv_t *envp);`
 11909 `int fesetenv(const fenv_t *envp);`

11910 **DESCRIPTION**
 11911 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 11912 conflict between the requirements described here and the ISO C standard is unintentional. This
 11913 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11914 The *fegetenv()* function shall attempt to store the current floating-point environment in the object
 11915 pointed to by *envp*.

11916 The *fesetenv()* function shall attempt to establish the floating-point environment represented by
 11917 the object pointed to by *envp*. The argument *envp* shall point to an object set by a call to
 11918 *fegetenv()* or *feholdexcept()*, or equal a floating-point environment macro. The *fesetenv()* function
 11919 does not raise floating-point exceptions, but only installs the state of the floating-point status
 11920 flags represented through its argument.

11921 **RETURN VALUE**
 11922 If the representation was successfully stored, *fegetenv()* shall return zero. Otherwise, it shall
 11923 return a non-zero value. If the environment was successfully established, *fesetenv()* shall return
 11924 zero. Otherwise, it shall return a non-zero value.

11925 **ERRORS**
 11926 No errors are defined.

11927 **EXAMPLES**
 11928 None.

11929 **APPLICATION USAGE**
 11930 None.

11931 **RATIONALE**
 11932 None.

11933 **FUTURE DIRECTIONS**
 11934 None.

11935 **SEE ALSO**
 11936 *feholdexcept()*, *feupdateenv()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<fenv.h>`

11937 **CHANGE HISTORY**
 11938 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.
 11939 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

11940 **NAME**

11941 fegetexceptflag, fesetexceptflag — get and set floating-point status flags

11942 **SYNOPSIS**

11943 #include <fenv.h>

11944 int fegetexceptflag(fexcept_t *flagp, int excepts);

11945 int fesetexceptflag(const fexcept_t *flagp, int excepts);

11946 **DESCRIPTION**

11947 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 11948 conflict between the requirements described here and the ISO C standard is unintentional. This
 11949 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11950 The *fegetexceptflag()* function shall attempt to store an implementation-defined representation of
 11951 the states of the floating-point status flags indicated by the argument *excepts* in the object
 11952 pointed to by the argument *flagp*.

11953 The *fesetexceptflag()* function shall attempt to set the floating-point status flags indicated by the
 11954 argument *excepts* to the states stored in the object pointed to by *flagp*. The value pointed to by
 11955 *flagp* shall have been set by a previous call to *fegetexceptflag()* whose second argument
 11956 represented at least those floating-point exceptions represented by the argument *excepts*. This
 11957 function does not raise floating-point exceptions, but only sets the state of the flags.

11958 **RETURN VALUE**

11959 If the representation was successfully stored, *fegetexceptflag()* shall return zero. Otherwise, it
 11960 shall return a non-zero value. If the *excepts* argument is zero or if all the specified exceptions
 11961 were successfully set, *fesetexceptflag()* shall return zero. Otherwise, it shall return a non-zero
 11962 value.

11963 **ERRORS**

11964 No errors are defined.

11965 **EXAMPLES**

11966 None.

11967 **APPLICATION USAGE**

11968 None.

11969 **RATIONALE**

11970 None.

11971 **FUTURE DIRECTIONS**

11972 None.

11973 **SEE ALSO**

11974 *feclearexcept()*, *feraiseexcept()*, *fetestexcept()*, the Base Definitions volume of IEEE Std 1003.1-200x,
 11975 <fenv.h>

11976 **CHANGE HISTORY**

11977 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11978 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

11979 **NAME**

11980 fegetround, fesetround — get and set current rounding direction

11981 **SYNOPSIS**

```
11982 #include <fenv.h>
11983
11983 int fegetround(void);
11984 int fesetround(int round);
```

11985 **DESCRIPTION**

11986 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 11987 conflict between the requirements described here and the ISO C standard is unintentional. This
 11988 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11989 The *fegetround()* function shall get the current rounding direction.

11990 The *fesetround()* function shall establish the rounding direction represented by its argument
 11991 *round*. If the argument is not equal to the value of a rounding direction macro, the rounding
 11992 direction is not changed.

11993 **RETURN VALUE**

11994 The *fegetround()* function shall return the value of the rounding direction macro representing the
 11995 current rounding direction or a negative value if there is no such rounding direction macro or
 11996 the current rounding direction is not determinable.

11997 The *fesetround()* function shall return a zero value if and only if the requested rounding direction
 11998 was established.

11999 **ERRORS**

12000 No errors are defined.

12001 **EXAMPLES**

12002 The following example saves, sets, and restores the rounding direction, reporting an error and
 12003 aborting if setting the rounding direction fails:

```
12004 #include <fenv.h>
12005 #include <assert.h>
12006 void f(int round_dir)
12007 {
12008     #pragma STDC FENV_ACCESS ON
12009     int save_round;
12010     int setround_ok;
12011     save_round = fegetround();
12012     setround_ok = fesetround(round_dir);
12013     assert(setround_ok == 0);
12014     /* ... */
12015     fesetround(save_round);
12016     /* ... */
12017 }
```

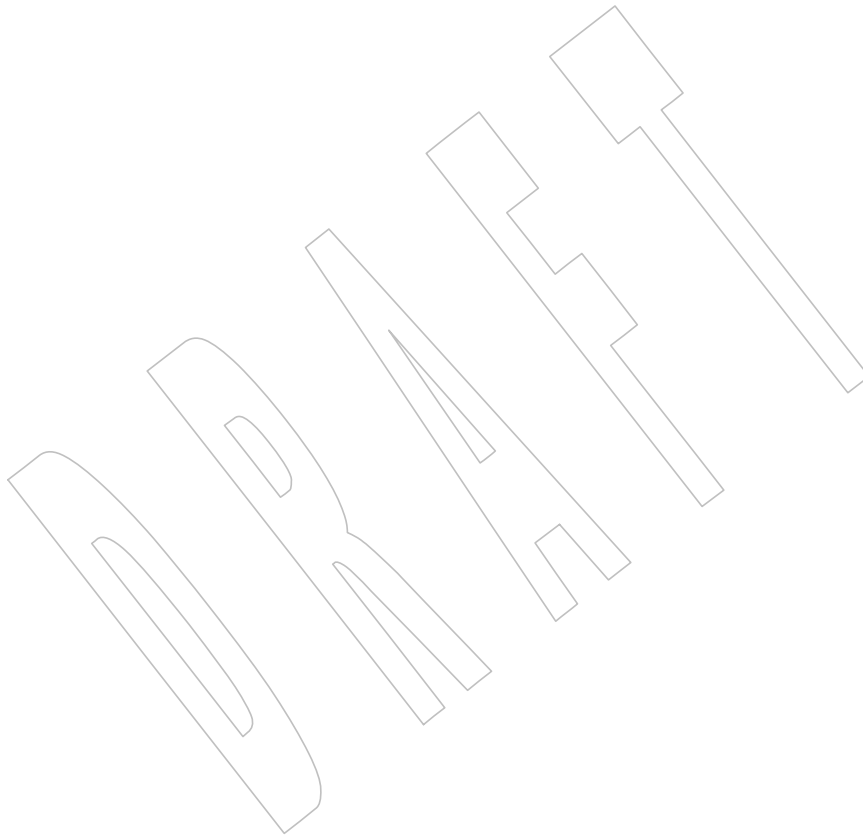
12018 **APPLICATION USAGE**

12019 None.

12020 **RATIONALE**

12021 None.

12022	FUTURE DIRECTIONS
12023	None.
12024	SEE ALSO
12025	The Base Definitions volume of IEEE Std 1003.1-200x, <fenv.h>
12026	CHANGE HISTORY
12027	First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.
12028	ISO/IEC 9899: 1999 standard, Technical Corrigendum 1 is incorporated.



12029 **NAME**

12030 feholdexcept — save current floating-point environment

12031 **SYNOPSIS**

12032 #include <fenv.h>

12033 int feholdexcept(fenv_t *envp);

12034 **DESCRIPTION**

12035 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 12036 conflict between the requirements described here and the ISO C standard is unintentional. This
 12037 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12038 The *feholdexcept()* function shall save the current floating-point environment in the object
 12039 pointed to by *envp*, clear the floating-point status flags, and then install a non-stop (continue on
 12040 floating-point exceptions) mode, if available, for all floating-point exceptions.

12041 **RETURN VALUE**

12042 The *feholdexcept()* function shall return zero if and only if non-stop floating-point exception
 12043 handling was successfully installed.

12044 **ERRORS**

12045 No errors are defined.

12046 **EXAMPLES**

12047 None.

12048 **APPLICATION USAGE**

12049 None.

12050 **RATIONALE**

12051 The *feholdexcept()* function should be effective on typical IEC 60559:1989 standard
 12052 implementations which have the default non-stop mode and at least one other mode for trap
 12053 handling or aborting. If the implementation provides only the non-stop mode, then installing the
 12054 non-stop mode is trivial.

12055 **FUTURE DIRECTIONS**

12056 None.

12057 **SEE ALSO**

12058 *fegetenv()*, *fesetenv()*, *feupdateenv()*, the Base Definitions volume of IEEE Std 1003.1-200x,
 12059 <fenv.h>

12060 **CHANGE HISTORY**

12061 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

12062 **NAME**
 12063 feof — test end-of-file indicator on a stream

12064 **SYNOPSIS**
 12065 #include <stdio.h>
 12066 int feof(FILE *stream);

12067 **DESCRIPTION**
 12068 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 12069 conflict between the requirements described here and the ISO C standard is unintentional. This
 12070 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12071 The *feof()* function shall test the end-of-file indicator for the stream pointed to by *stream*.

12072 **RETURN VALUE**
 12073 The *feof()* function shall return non-zero if and only if the end-of-file indicator is set for *stream*.

12074 **ERRORS**
 12075 No errors are defined.

12076 **EXAMPLES**
 12077 None.

12078 **APPLICATION USAGE**
 12079 None.

12080 **RATIONALE**
 12081 None.

12082 **FUTURE DIRECTIONS**
 12083 None.

12084 **SEE ALSO**
 12085 *clearerr()*, *ferror()*, *fopen()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>

12086 **CHANGE HISTORY**
 12087 First released in Issue 1. Derived from Issue 1 of the SVID.

12088 **NAME**
 12089 `feraiseexcept` — raise floating-point exception

12090 **SYNOPSIS**
 12091 `#include <fenv.h>`
 12092 `int feraiseexcept(int excepts);`

12093 **DESCRIPTION**
 12094 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 12095 conflict between the requirements described here and the ISO C standard is unintentional. This
 12096 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12097 The *feraiseexcept()* function shall attempt to raise the supported floating-point exceptions
 12098 represented by the argument *excepts*. The order in which these floating-point exceptions are
 12099 raised is unspecified. Whether the *feraiseexcept()* function additionally raises the inexact floating-
 12100 point exception whenever it raises the overflow or underflow floating-point exception is
 12101 implementation-defined.

12102 **RETURN VALUE**
 12103 If the argument is zero or if all the specified exceptions were successfully raised, *feraiseexcept()*
 12104 shall return zero. Otherwise, it shall return a non-zero value.

12105 **ERRORS**
 12106 No errors are defined.

12107 **EXAMPLES**
 12108 None.

12109 **APPLICATION USAGE**
 12110 The effect is intended to be similar to that of floating-point exceptions raised by arithmetic
 12111 operations. Hence, enabled traps for floating-point exceptions raised by this function are taken.

12112 **RATIONALE**
 12113 Raising overflow or underflow is allowed to also raise inexact because on some architectures the
 12114 only practical way to raise an exception is to execute an instruction that has the exception as a
 12115 side effect. The function is not restricted to accept only valid coincident expressions for atomic
 12116 operations, so the function can be used to raise exceptions accrued over several operations.

12117 **FUTURE DIRECTIONS**
 12118 None.

12119 **SEE ALSO**
 12120 *feclearexcept()*, *fegetexceptflag()*, *fesetexceptflag()*, *fetestexcept()*, the Base Definitions volume of
 12121 IEEE Std 1003.1-200x, `<fenv.h>`

12122 **CHANGE HISTORY**
 12123 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.
 12124 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

12125 **NAME**

12126 error — test error indicator on a stream

12127 **SYNOPSIS**

12128 #include <stdio.h>

12129 int ferror(FILE *stream);

12130 **DESCRIPTION**

12131 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 12132 conflict between the requirements described here and the ISO C standard is unintentional. This
 12133 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12134 The *ferror()* function shall test the error indicator for the stream pointed to by *stream*.12135 **RETURN VALUE**12136 The *ferror()* function shall return non-zero if and only if the error indicator is set for *stream*.12137 **ERRORS**

12138 No errors are defined.

12139 **EXAMPLES**

12140 None.

12141 **APPLICATION USAGE**

12142 None.

12143 **RATIONALE**

12144 None.

12145 **FUTURE DIRECTIONS**

12146 None.

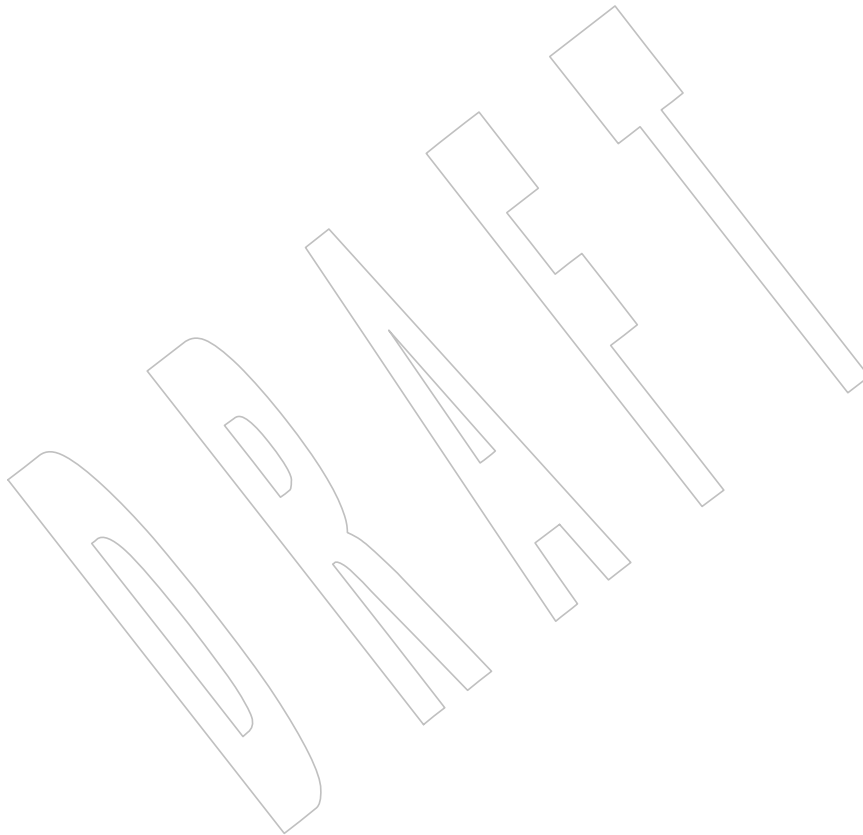
12147 **SEE ALSO**12148 *clearerr()*, *feof()*, *fopen()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>12149 **CHANGE HISTORY**

12150 First released in Issue 1. Derived from Issue 1 of the SVID.

12151 **NAME**
12152 fesetenv — set current floating-point environment

12153 **SYNOPSIS**
12154 #include <fenv.h>
12155 int fesetenv(const fenv_t *envp);

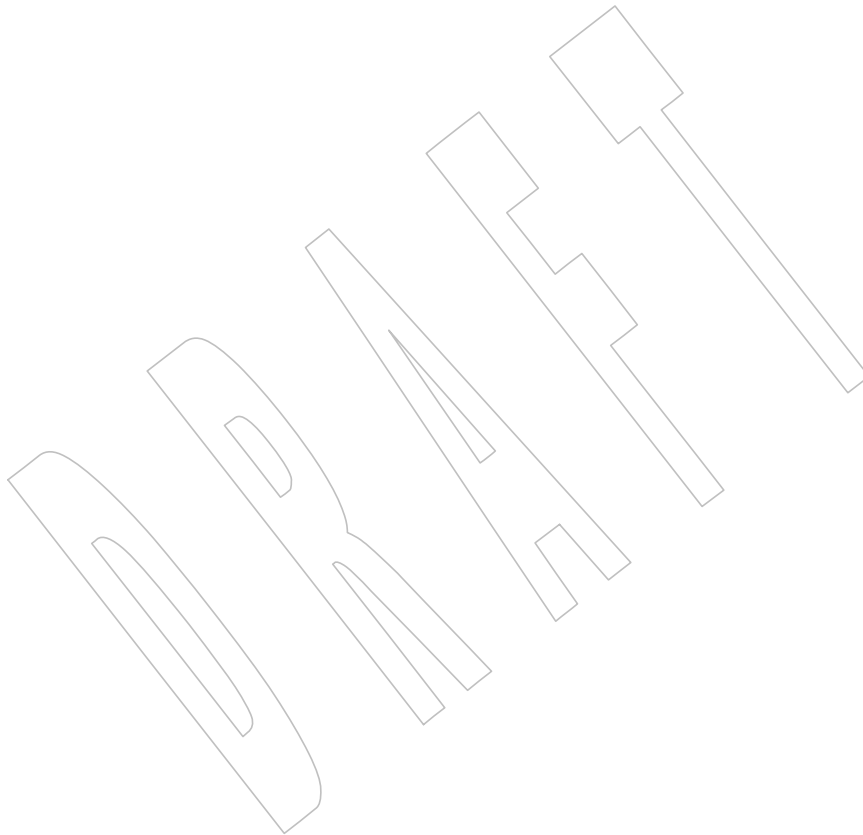
12156 **DESCRIPTION**
12157 Refer to *fegetenv()*.



12158 **NAME**
12159 `fesetexceptflag` — set floating-point status flags

12160 **SYNOPSIS**
12161 `#include <fenv.h>`
12162 `int fesetexceptflag(const fexcept_t *flagp, int excepts);`

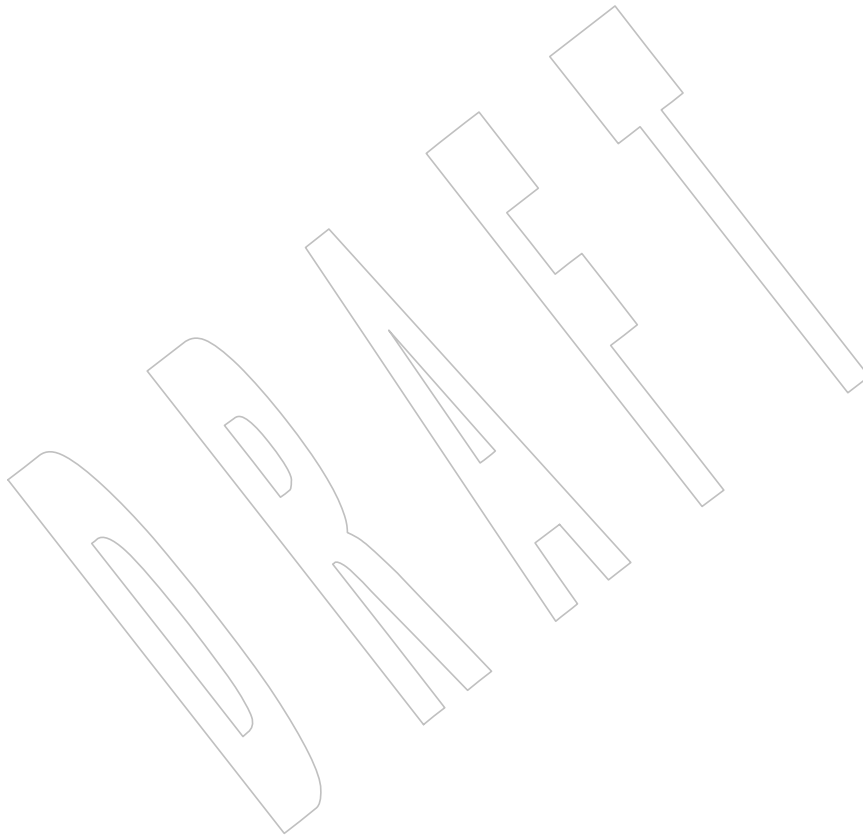
12163 **DESCRIPTION**
12164 Refer to *fegetexceptflag()*.



12165 **NAME**
12166 fesetround — set current rounding direction

12167 **SYNOPSIS**
12168 #include <fenv.h>
12169 int fesetround(int *round*);

12170 **DESCRIPTION**
12171 Refer to *fegetround()*.



12172 **NAME**
 12173 fetestexcept — test floating-point exception flags

12174 **SYNOPSIS**
 12175 #include <fenv.h>
 12176 int fetestexcept(int *excepts*);

12177 **DESCRIPTION**
 12178 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 12179 conflict between the requirements described here and the ISO C standard is unintentional. This
 12180 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12181 The *fetestexcept()* function shall determine which of a specified subset of the floating-point
 12182 exception flags are currently set. The *excepts* argument specifies the floating-point status flags to
 12183 be queried.

12184 **RETURN VALUE**
 12185 The *fetestexcept()* function shall return the value of the bitwise-inclusive OR of the floating-point
 12186 exception macros corresponding to the currently set floating-point exceptions included in
 12187 *excepts*.

12188 **ERRORS**
 12189 No errors are defined.

12190 **EXAMPLES**
 12191 The following example calls function *f()* if an invalid exception is set, and then function *g()* if an
 12192 overflow exception is set:

```
12193 #include <fenv.h>
12194 /* ... */
12195 {
12196     #pragma STDC FENV_ACCESS ON
12197     int set_excepts;
12198     feclearexcept(FE_INVALID | FE_OVERFLOW);
12199     // maybe raise exceptions
12200     set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
12201     if (set_excepts & FE_INVALID) f();
12202     if (set_excepts & FE_OVERFLOW) g();
12203     /* ... */
12204 }
```

12205 **APPLICATION USAGE**
 12206 None.

12207 **RATIONALE**
 12208 None.

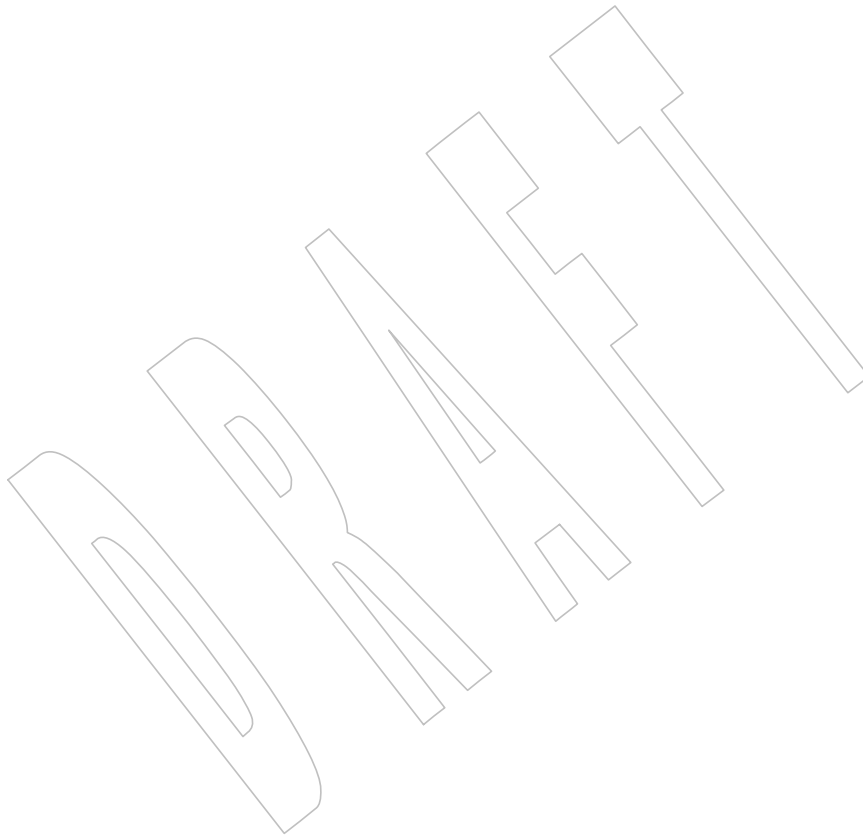
12209 **FUTURE DIRECTIONS**
 12210 None.

12211 **SEE ALSO**
 12212 *feclearexcept()*, *fegetexceptflag()*, *feraiseexcept()*, the Base Definitions volume of
 12213 IEEE Std 1003.1-200x, <fenv.h>

12214
12215

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.



12216 **NAME**
 12217 `feupdateenv` — update floating-point environment

12218 **SYNOPSIS**
 12219 `#include <fenv.h>`
 12220 `int feupdateenv(const fenv_t *envp);`

12221 **DESCRIPTION**
 12222 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 12223 conflict between the requirements described here and the ISO C standard is unintentional. This
 12224 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12225 The `feupdateenv()` function shall attempt to save the currently raised floating-point exceptions in
 12226 its automatic storage, attempt to install the floating-point environment represented by the object
 12227 pointed to by `envp`, and then attempt to raise the saved floating-point exceptions. The argument
 12228 `envp` shall point to an object set by a call to `feholdexcept()` or `fegetenv()`, or equal a floating-point
 12229 environment macro.

12230 **RETURN VALUE**
 12231 The `feupdateenv()` function shall return a zero value if and only if all the required actions were
 12232 successfully carried out.

12233 **ERRORS**
 12234 No errors are defined.

12235 **EXAMPLES**
 12236 The following example shows sample code to hide spurious underflow floating-point
 12237 exceptions:

```
12238 #include <fenv.h>
12239 double f(double x)
12240 {
12241     #pragma STDC FENV_ACCESS ON
12242     double result;
12243     fenv_t save_env;
12244     feholdexcept(&save_env);
12245     // compute result
12246     if (/* test spurious underflow */)
12247         feclearexcept(FE_UNDERFLOW);
12248     feupdateenv(&save_env);
12249     return result;
12250 }
```

12251 **APPLICATION USAGE**
 12252 None.

12253 **RATIONALE**
 12254 None.

12255 **FUTURE DIRECTIONS**
 12256 None.

12257 **SEE ALSO**
 12258 [*fegetenv\(\)*](#), [*feholdexcept\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<fenv.h>`

12259

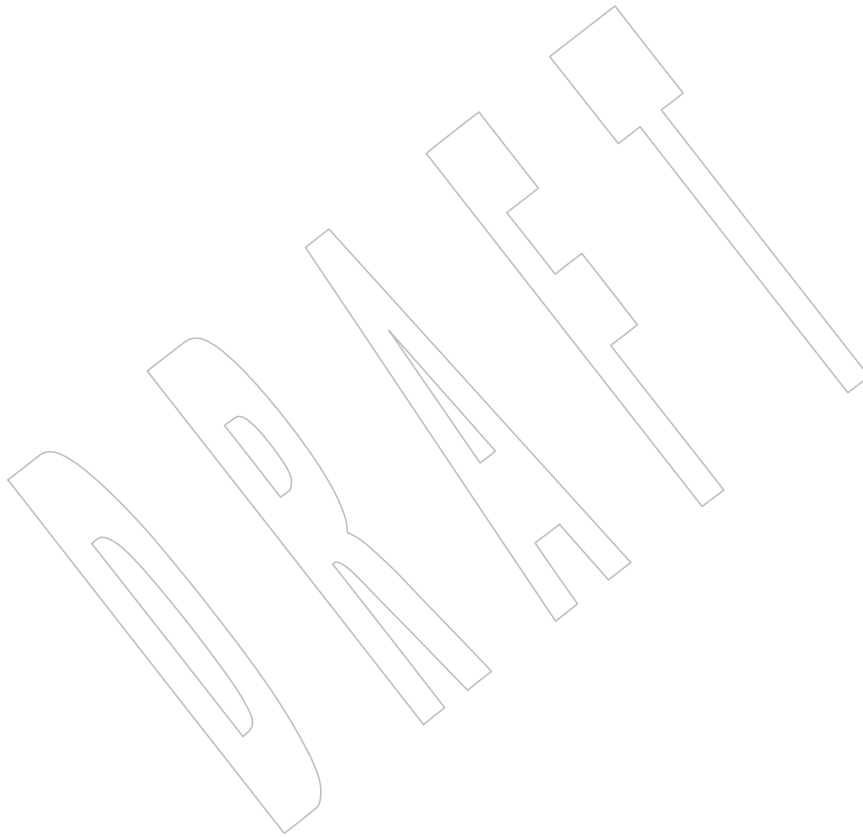
CHANGE HISTORY

12260

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

12261

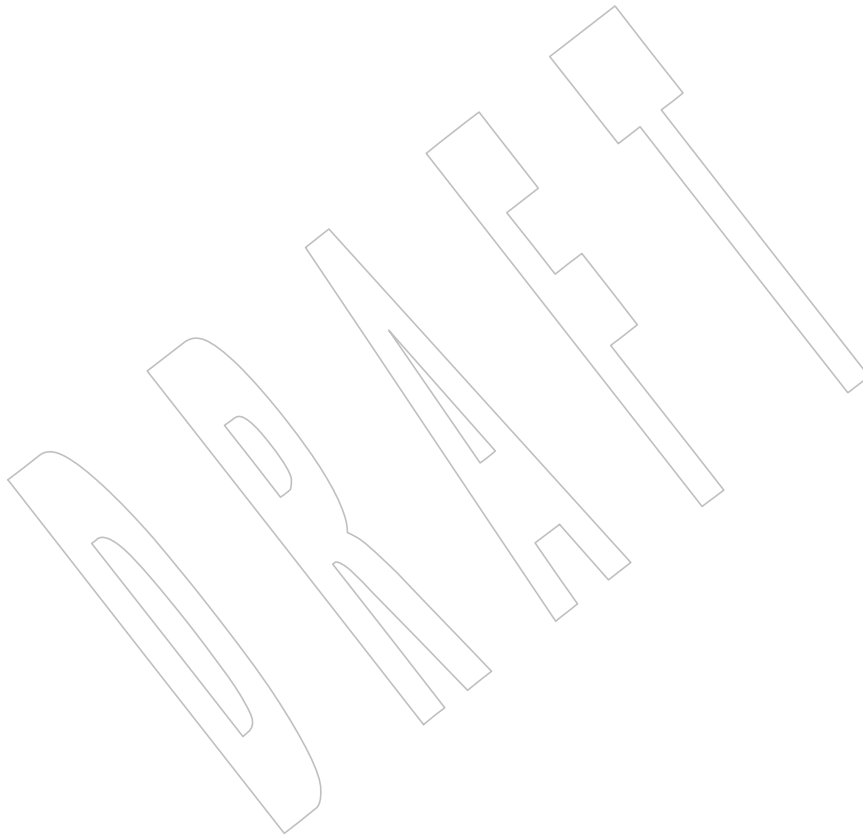
ISO/IEC 9899: 1999 standard, Technical Corrigendum 1 is incorporated.



12262 **NAME**
12263 fexecve — execute a file

12264 **SYNOPSIS**
12265 #include <unistd.h>
12266 int fexecve(int *fd*, char *const *argv*[], char *const *envp*[]);

12267 **DESCRIPTION**
12268 Refer to *exec*.



12269 **NAME**12270 `fflush` — flush a stream12271 **SYNOPSIS**12272 `#include <stdio.h>`12273 `int fflush(FILE *stream);`12274 **DESCRIPTION**

12275 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 12276 conflict between the requirements described here and the ISO C standard is unintentional. This
 12277 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12278 If *stream* points to an output stream or an update stream in which the most recent operation was
 12279 not input, *fflush()* shall cause any unwritten data for that stream to be written to the file, and the
 12280 *st_ctime* and *st_mtime* fields of the underlying file shall be marked for update.

12281 If *stream* is a null pointer, *fflush()* shall perform this flushing action on all streams for which the
 12282 behavior is defined above.

12283 CX For a stream open for reading, if the file is not already at EOF, and the file is one capable of
 12284 seeking, the file offset of the underlying open file description shall be adjusted so that the next
 12285 operation on the open file description deals with the byte after the last one read from or written
 12286 to the stream being flushed.

12287 **RETURN VALUE**

12288 Upon successful completion, *fflush()* shall return 0; otherwise, it shall set the error indicator for
 12289 the stream, return EOF, and set *errno* to indicate the error.

12290 **ERRORS**12291 The *fflush()* function shall fail if:

12292 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and
 12293 the thread would be delayed in the write operation.

12294 CX [EBADF] The file descriptor underlying *stream* is not valid.

12295 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size.

12296 XSI [EFBIG] An attempt was made to write a file that exceeds the file size limit of the
 12297 process.

12298 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
 12299 offset maximum associated with the corresponding stream.

12300 CX [EINTR] The *fflush()* function was interrupted by a signal.

12301 CX [EIO] The process is a member of a background process group attempting to write to
 12302 its controlling terminal, TOSTOP is set, the process is neither ignoring nor
 12303 blocking SIGTTOU, and the process group of the process is orphaned. This
 12304 error may also be returned under implementation-defined conditions.

12305 CX [ENOMEM] The underlying stream was created by *open_memstream()* or
 12306 *open_wmemstream()* and insufficient memory is available.

12307 CX [ENOSPC] There was no free space remaining on the device containing the file or in the
 12308 buffer used by the *fmemopen()* function.

12309 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
 12310 any process. A SIGPIPE signal shall also be sent to the thread.

12311 The *fflush()* function may fail if:

12312 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
12313 capabilities of the device.

12314 EXAMPLES

12315 Sending Prompts to Standard Output

12316 The following example uses *printf()* calls to print a series of prompts for information the user
12317 must enter from standard input. The *fflush()* calls force the output to standard output. The
12318 *fflush()* function is used because standard output is usually buffered and the prompt may not
12319 immediately be printed on the output or terminal. The *gets()* calls read strings from standard
12320 input and place the results in variables, for use later in the program.

```
12321 #include <stdio.h>
12322 ...
12323 char user[100];
12324 char oldpasswd[100];
12325 char newpasswd[100];
12326 ...
12327 printf("User name: ");
12328 fflush(stdout);
12329 gets(user);
12330
12331 printf("Old password: ");
12332 fflush(stdout);
12333 gets(oldpasswd);
12334
12335 printf("New password: ");
12336 fflush(stdout);
12337 gets(newpasswd);
12338 ...
```

12337 APPLICATION USAGE

12338 None.

12339 RATIONALE

12340 Data buffered by the system may make determining the validity of the position of the current
12341 file descriptor impractical. Thus, enforcing the repositioning of the file descriptor after *fflush()*
12342 on streams open for *read()* is not mandated by IEEE Std 1003.1-200x.

12343 FUTURE DIRECTIONS

12344 None.

12345 SEE ALSO

12346 *fnmopen()*, *getrlimit()*, *open_memstream()*, *ulimit()*, the Base Definitions volume of
12347 IEEE Std 1003.1-200x, **<stdio.h>**

12348 CHANGE HISTORY

12349 First released in Issue 1. Derived from Issue 1 of the SVID.

12350 Issue 5

12351 Large File Summit extensions are added.

12352 Issue 6

12353 Extensions beyond the ISO C standard are marked.

12354 The following new requirements on POSIX implementations derive from alignment with the
12355 Single UNIX Specification:

12356

- The [EFBIG] error is added as part of the large file support extensions.

12357

- The [ENXIO] optional error condition is added.

12358

The RETURN VALUE section is updated to note that the error indicator shall be set for the stream. This is for alignment with the ISO/IEC 9899:1999 standard.

12359

12360

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/31 is applied, updating the [EAGAIN] error in the ERRORS section from “the process would be delayed” to “the thread would be delayed”.

12361

12362

12363

Issue 7

12364

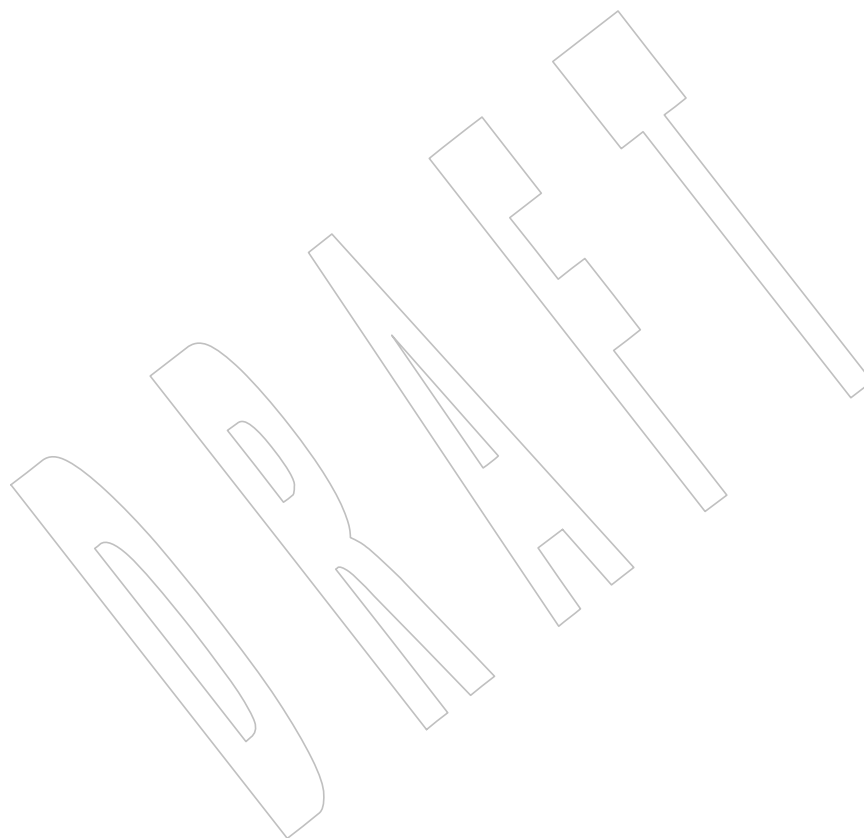
Austin Group Interpretation 1003.1-2001 #002 is applied, clarifying the interaction of file descriptors and streams.

12365

12366

The [ENOSPC] error condition is updated and the [ENOMEM] error is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

12367



12368 **NAME**
 12369 ffs — find first set bit

12370 **SYNOPSIS**

```
12371 xSI #include <strings.h>
12372 int ffs(int i);
```

12373 **DESCRIPTION**

12374 The *ffs()* function shall find the first bit set (beginning with the least significant bit) in *i*, and
 12375 return the index of that bit. Bits are numbered starting at one (the least significant bit).

12376 **RETURN VALUE**

12377 The *ffs()* function shall return the index of the first bit set. If *i* is 0, then *ffs()* shall return 0.

12378 **ERRORS**

12379 No errors are defined.

12380 **EXAMPLES**

12381 None.

12382 **APPLICATION USAGE**

12383 None.

12384 **RATIONALE**

12385 None.

12386 **FUTURE DIRECTIONS**

12387 None.

12388 **SEE ALSO**

12389 The Base Definitions volume of IEEE Std 1003.1-200x, **<strings.h>**

12390 **CHANGE HISTORY**

12391 First released in Issue 4, Version 2.

12392 **Issue 5**

12393 Moved from X/OPEN UNIX extension to BASE.

12394 **NAME**12395 `fgetc` — get a byte from a stream12396 **SYNOPSIS**12397 `#include <stdio.h>`12398 `int fgetc(FILE *stream);`12399 **DESCRIPTION**12400 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12401 conflict between the requirements described here and the ISO C standard is unintentional. This
12402 volume of IEEE Std 1003.1-200x defers to the ISO C standard.12403 If the end-of-file indicator for the input stream pointed to by *stream* is not set and a next byte is
12404 present, the *fgetc()* function shall obtain the next byte as an **unsigned char** converted to an **int**,
12405 from the input stream pointed to by *stream*, and advance the associated file position indicator for
12406 the stream (if defined). Since *fgetc()* operates on bytes, reading a character consisting of multiple
12407 bytes (or “a multi-byte character”) may require multiple calls to *fgetc()*.12408 CX The *fgetc()* function may mark the *st_atime* field of the file associated with *stream* for update. The
12409 *st_atime* field shall be marked for update by the first successful execution of *fgetc()*, *fgets()*,
12410 *fread()*, *fscanf()*, *getc()*, *getchar()*, *gets()*, or *scanf()* using *stream* that returns data not supplied by
12411 a prior call to *ungetc()*.12412 **RETURN VALUE**12413 Upon successful completion, *fgetc()* shall return the next byte from the input stream pointed to
12414 by *stream*. If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the
12415 end-of-file indicator for the stream shall be set and *fgetc()* shall return EOF. If a read error occurs,
12416 CX the error indicator for the stream shall be set, *fgetc()* shall return EOF, and shall set *errno* to
12417 indicate the error.12418 **ERRORS**12419 The *fgetc()* function shall fail if data needs to be read and:12420 CX **[EAGAIN]** The `O_NONBLOCK` flag is set for the file descriptor underlying *stream* and
12421 the thread would be delayed in the *fgetc()* operation.12422 CX **[EBADF]** The file descriptor underlying *stream* is not a valid file descriptor open for
12423 reading.12424 CX **[EINTR]** The read operation was terminated due to the receipt of a signal, and no data
12425 was transferred.12426 CX **[EIO]** A physical I/O error has occurred, or the process is in a background process
12427 group attempting to read from its controlling terminal, and either the process
12428 is ignoring or blocking the SIGTTIN signal or the process group is orphaned.
12429 This error may also be generated for implementation-defined reasons.12430 CX **[EOVERFLOW]** The file is a regular file and an attempt was made to read at or beyond the
12431 offset maximum associated with the corresponding stream.12432 The *fgetc()* function may fail if:12433 CX **[ENOMEM]** Insufficient storage space is available.12434 CX **[ENXIO]** A request was made of a nonexistent device, or the request was outside the
12435 capabilities of the device.

EXAMPLES

None.

APPLICATION USAGE

If the integer value returned by *fgetc()* is stored into a variable of type **char** and then compared against the integer constant EOF, the comparison may never succeed, because sign-extension of a variable of type **char** on widening to integer is implementation-defined.

The *ferror()* or *feof()* functions must be used to distinguish between an error condition and an end-of-file condition.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feof(), *ferror()*, *fgets()*, *fread()*, *fscanf()*, *getchar()*, *getc()*, *gets()*, *scanf()*, *ungetc()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] and [EOVERFLOW] mandatory error conditions are added.
- The [ENOMEM] and [ENXIO] optional error conditions are added.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The DESCRIPTION is updated to clarify the behavior when the end-of-file indicator for the input stream is not set.
- The RETURN VALUE section is updated to note that the error indicator shall be set for the stream.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/32 is applied, updating the [EAGAIN] error in the ERRORS section from “the process would be delayed” to “the thread would be delayed”.

Issue 7

Austin Group Interpretation 1003.1-2001 #051 is applied, updating the list of functions that mark the *st_atime* field for update.

NAME

`fgetpos` — get current file position information

SYNOPSIS

```
#include <stdio.h>
```

```
int fgetpos(FILE *restrict stream, fpos_t *restrict pos);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The `fgetpos()` function shall store the current values of the parse state (if any) and file position indicator for the stream pointed to by `stream` in the object pointed to by `pos`. The value stored contains unspecified information usable by `fsetpos()` for repositioning the stream to its position at the time of the call to `fgetpos()`.

RETURN VALUE

Upon successful completion, `fgetpos()` shall return 0; otherwise, it shall return a non-zero value and set `errno` to indicate the error.

ERRORS

The `fgetpos()` function shall fail if:

CX [EOVERFLOW] The current value of the file position cannot be represented correctly in an object of type `fpos_t`.

The `fgetpos()` function may fail if:

CX [EBADF] The file descriptor underlying `stream` is not valid.

CX [ESPIPE] The file descriptor underlying `stream` is associated with a pipe, FIFO, or socket.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fopen(), *ftell()*, *re*

12510
12511
12512
12513
12514
12515
12516

Issue 6

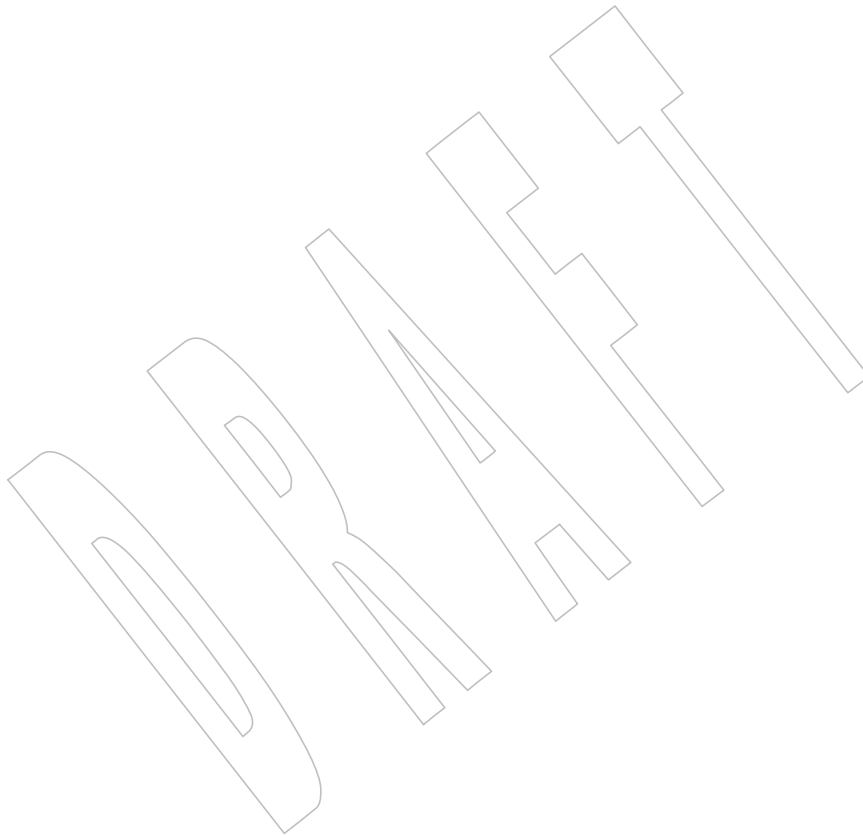
Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EBADF] and [ESPIPE] optional error conditions are added.

An additional [ESPIPE] error condition is added for sockets.

The prototype for *fgetpos()* is changed for alignment with the ISO/IEC 9899:1999 standard.



12517 **NAME**12518 `fgets` — get a string from a stream12519 **SYNOPSIS**12520 `#include <stdio.h>`12521 `char *fgets(char *restrict s, int n, FILE *restrict stream);`12522 **DESCRIPTION**12523 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12524 conflict between the requirements described here and the ISO C standard is unintentional. This
12525 volume of IEEE Std 1003.1-200x defers to the ISO C standard.12526 The `fgets()` function shall read bytes from *stream* into the array pointed to by *s*, until *n*−1 bytes
12527 are read, or a <newline> is read and transferred to *s*, or an end-of-file condition is encountered.
12528 The string is then terminated with a null byte.12529 CX The `fgets()` function may mark the *st_atime* field of the file associated with *stream* for update. The
12530 *st_atime* field shall be marked for update by the first successful execution of `fgetc()`, `fgets()`,
12531 `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()`, or `scanf()` using *stream* that returns data not supplied by
12532 a prior call to `ungetc()`.12533 **RETURN VALUE**12534 Upon successful completion, `fgets()` shall return *s*. If the stream is at end-of-file, the end-of-file
12535 indicator for the stream shall be set and `fgets()` shall return a null pointer. If a read error occurs,
12536 CX the error indicator for the stream shall be set, `fgets()` shall return a null pointer, and shall set
12537 *errno* to indicate the error.12538 **ERRORS**12539 Refer to `fgetc()`.12540 **EXAMPLES**12541 **Reading Input**12542 The following example uses `fgets()` to read each line of input. {LINE_MAX}, which defines the
12543 maximum size of the input line, is defined in the <limits.h> header.12544 `#include <stdio.h>`
12545 `...`
12546 `char line[LINE_MAX];`
12547 `...`
12548 `while (fgets(line, LINE_MAX, fp) != NULL) {`
12549 `...`
12550 `}`
12551 `...`12552 **APPLICATION USAGE**

12553 None.

12554 **RATIONALE**

12555 None.

12556 **FUTURE DIRECTIONS**

12557 None.

12558

SEE ALSO

12559

fgetc(), *fopen()*, *fread()*, *fscanf()*, *getc()*, *getchar()*, *getdelim()*, *gets()*, *scanf()*, *ungetc()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

12560

12561

CHANGE HISTORY

12562

First released in Issue 1. Derived from Issue 1 of the SVID.

12563

Issue 6

12564

Extensions beyond the ISO C standard are marked.

12565

The prototype for *fgets()* is changed for alignment with the ISO/IEC 9899:1999 standard.

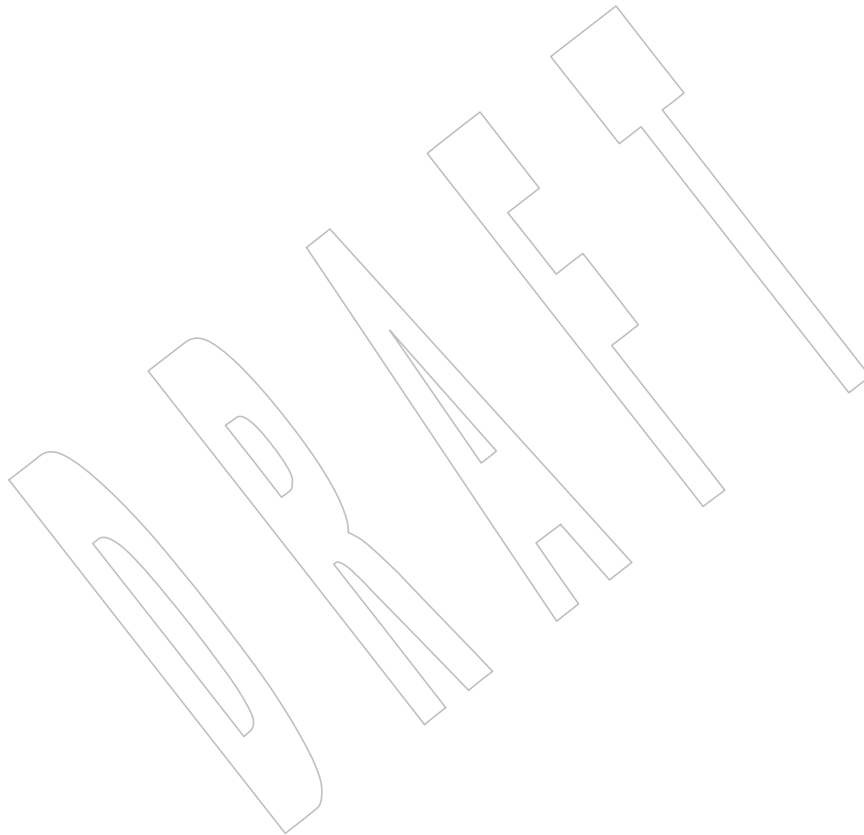
12566

Issue 7

12567

Austin Group Interpretation 1003.1-2001 #051 is applied, updating the list of functions that mark the *st_atime* field for update.

12568



12569 **NAME**12570 `fgetwc` — get a wide-character code from a stream12571 **SYNOPSIS**12572 `#include <stdio.h>`12573 `#include <wchar.h>`12574 `wint_t fgetwc(FILE *stream);`12575 **DESCRIPTION**12576 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12577 conflict between the requirements described here and the ISO C standard is unintentional. This
12578 volume of IEEE Std 1003.1-200x defers to the ISO C standard.12579 The `fgetwc()` function shall obtain the next character (if present) from the input stream pointed to
12580 by `stream`, convert that to the corresponding wide-character code, and advance the associated file
12581 position indicator for the stream (if defined).

12582 If an error occurs, the resulting value of the file position indicator for the stream is unspecified.

12583 CX The `fgetwc()` function may mark the `st_atime` field of the file associated with `stream` for update.
12584 The `st_atime` field shall be marked for update by the first successful execution of `fgetc()`, `fgets()`,
12585 `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()`, or `scanf()` using `stream` that returns
12586 data not supplied by a prior call to `ungetc()` or `ungetwc()`.12587 **RETURN VALUE**12588 Upon successful completion, the `fgetwc()` function shall return the wide-character code of the
12589 character read from the input stream pointed to by `stream` converted to a type `wint_t`. If the end-
12590 of-file indicator for the stream is set, or if the stream is at end-of-file, the end-of-file indicator for
12591 the stream shall be set and `fgetwc()` shall return WEOF. If a read error occurs, the error indicator
12592 CX for the stream shall be set, `fgetwc()` shall return WEOF, and shall set `errno` to indicate the error. If
12593 an encoding error occurs, the error indicator for the stream shall be set, `fgetwc()` shall return
12594 WEOF, and shall set `errno` to indicate the error.12595 **ERRORS**12596 The `fgetwc()` function shall fail if data needs to be read and:12597 CX **[EAGAIN]** The `O_NONBLOCK` flag is set for the file descriptor underlying `stream` and
12598 the thread would be delayed in the `fgetwc()` operation.12599 CX **[EBADF]** The file descriptor underlying `stream` is not a valid file descriptor open for
12600 reading.12601 **[EILSEQ]** The data obtained from the input stream does not form a valid character.12602 CX **[EINTR]** The read operation was terminated due to the receipt of a signal, and no data
12603 was transferred.12604 CX **[EIO]** A physical I/O error has occurred, or the process is in a background process
12605 group attempting to read from its controlling terminal, and either the process
12606 is ignoring or blocking the SIGTTIN signal or the process group is orphaned.
12607 This error may also be generated for implementation-defined reasons.12608 CX **[EOVERFLOW]** The file is a regular file and an attempt was made to read at or beyond the
12609 offset maximum associated with the corresponding stream.

12610 The *fgetwc()* function may fail if:

12611 CX [ENOMEM] Insufficient storage space is available.

12612 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
12613 capabilities of the device.

12614 EXAMPLES

12615 None.

12616 APPLICATION USAGE

12617 The *ferror()* or *feof()* functions must be used to distinguish between an error condition and an
12618 end-of-file condition.

12619 RATIONALE

12620 None.

12621 FUTURE DIRECTIONS

12622 None.

12623 SEE ALSO

12624 *feof()*, *ferror()*, *fopen()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdio.h>`,
12625 `<wchar.h>`

12626 CHANGE HISTORY

12627 First released in Issue 4. Derived from the MSE working draft.

12628 Issue 5

12629 The Optional Header (OH) marking is removed from `<stdio.h>`.

12630 Large File Summit extensions are added.

12631 Issue 6

12632 Extensions beyond the ISO C standard are marked.

12633 The following new requirements on POSIX implementations derive from alignment with the
12634 Single UNIX Specification:

- 12635 • The [EIO] and [Eoverflow] mandatory error conditions are added.
- 12636 • The [ENOMEM] and [ENXIO] optional error conditions are added.

12637 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/33 is applied, updating the [EAGAIN]
12638 error in the ERRORS section from “the process would be delayed” to “the thread would be
12639 delayed”.

12640 Issue 7

12641 Austin Group Interpretation 1003.1-2001 #051 is applied, clarifying the RETURN VALUE section.

12642 **NAME**
 12643 `fgetws` — get a wide-character string from a stream

12644 **SYNOPSIS**
 12645 `#include <stdio.h>`
 12646 `#include <wchar.h>`

12647 `wchar_t *fgetws(wchar_t *restrict ws, int n,`
 12648 `FILE *restrict stream);`

12649 DESCRIPTION

12650 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 12651 conflict between the requirements described here and the ISO C standard is unintentional. This
 12652 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12653 The `fgetws()` function shall read characters from the *stream*, convert these to the corresponding
 12654 wide-character codes, place them in the `wchar_t` array pointed to by *ws*, until *n*−1 characters are
 12655 read, or a <newline> is read, converted, and transferred to *ws*, or an end-of-file condition is
 12656 encountered. The wide-character string, *ws*, shall then be terminated with a null wide-character
 12657 code.

12658 If an error occurs, the resulting value of the file position indicator for the stream is unspecified.

12659 CX The `fgetws()` function may mark the `st_atime` field of the file associated with *stream* for update.
 12660 The `st_atime` field shall be marked for update by the first successful execution of `fgetc()`, `fgets()`,
 12661 `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()`, or `scanf()` using *stream* that returns
 12662 data not supplied by a prior call to `ungetc()` or `ungetwc()`.

12663 RETURN VALUE

12664 Upon successful completion, `fgetws()` shall return *ws*. If the end-of-file indicator for the stream is
 12665 set, or if the stream is at end-of-file, the end-of-file indicator for the stream shall be set and
 12666 `fgetws()` shall return a null pointer. If a read error occurs, the error indicator for the stream shall
 12667 CX be set, `fgetws()` shall return a null pointer, and shall set `errno` to indicate the error.

12668 ERRORS

12669 Refer to `fgetwc()`.

12670 EXAMPLES

12671 None.

12672 APPLICATION USAGE

12673 None.

12674 RATIONALE

12675 None.

12676 FUTURE DIRECTIONS

12677 None.

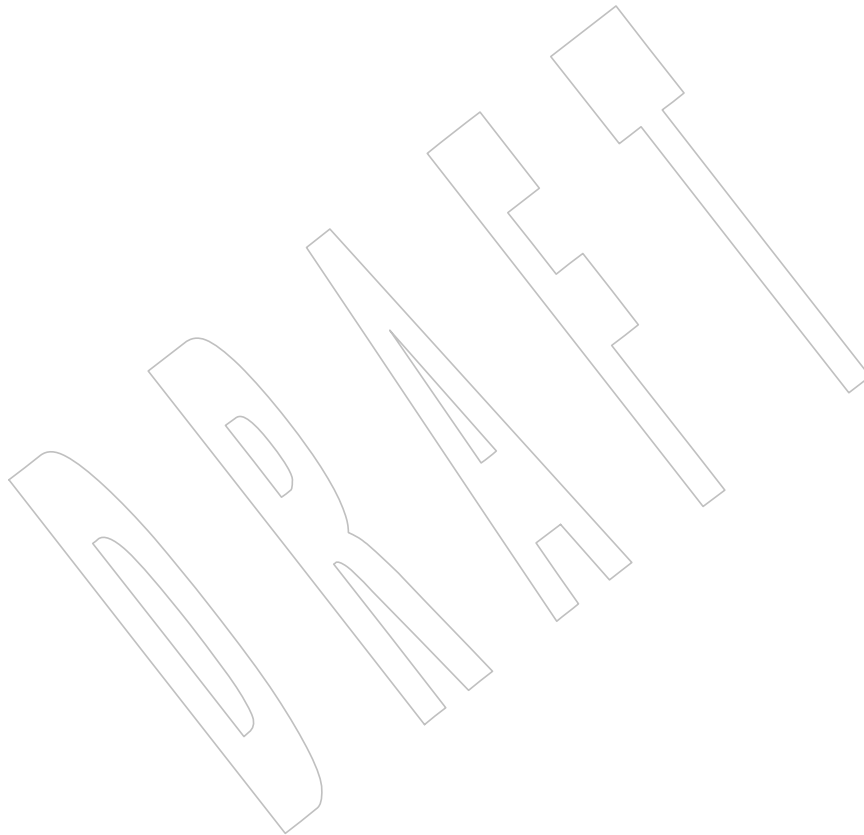
12678 SEE ALSO

12679 `fopen()`, `fread()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdio.h>`, `<wchar.h>`

12680 CHANGE HISTORY

12681 First released in Issue 4. Derived from the MSE working draft.

- 12682 **Issue 5**
12683 The Optional Header (OH) marking is removed from `<stdio.h>`.
- 12684 **Issue 6**
12685 Extensions beyond the ISO C standard are marked.
12686 The prototype for `fgetws()` is changed for alignment with the ISO/IEC 9899:1999 standard.
- 12687 **Issue 7**
12688 Austin Group Interpretation 1003.1-2001 #051 is applied, clarifying the RETURN VALUE section.



12689 **NAME**
 12690 `fileno` — map a stream pointer to a file descriptor

12691 **SYNOPSIS**

12692 CX

```
#include <stdio.h>
```


 12693

```
int fileno(FILE *stream);
```

12694 **DESCRIPTION**

12695 The `fileno()` function shall return the integer file descriptor associated with the stream pointed to
 12696 by `stream`.

12697 **RETURN VALUE**

12698 Upon successful completion, `fileno()` shall return the integer value of the file descriptor
 12699 associated with `stream`. Otherwise, the value `-1` shall be returned and `errno` set to indicate the
 12700 error.

12701 **ERRORS**

12702 The `fileno()` function may fail if:
 12703 [EBADF] The `stream` argument is not a valid stream.

12704 **EXAMPLES**

12705 None.

12706 **APPLICATION USAGE**

12707 None.

12708 **RATIONALE**

12709 Without some specification of which file descriptors are associated with these streams, it is
 12710 impossible for an application to set up the streams for another application it starts with `fork()`
 12711 and `exec`. In particular, it would not be possible to write a portable version of the `sh` command
 12712 interpreter (although there may be other constraints that would prevent that portability).

12713 **FUTURE DIRECTIONS**

12714 None.

12715 **SEE ALSO**

12716 [Section 2.5.1](#) (on page 35), `dirfd()`, `fdopen()`, `fopen()`, `stdin`, the Base Definitions volume of
 12717 IEEE Std 1003.1-200x, `<stdio.h>`

12718 **CHANGE HISTORY**

12719 First released in Issue 1. Derived from Issue 1 of the SVID.

12720 **Issue 6**

12721 The following new requirements on POSIX implementations derive from alignment with the
 12722 Single UNIX Specification:

- 12723 • The [EBADF] optional error condition is added.

12724 **NAME**
 12725 flockfile, ftrylockfile, funlockfile — stdio locking functions

12726 **SYNOPSIS**

```
12727 CX #include <stdio.h>
12728 void flockfile(FILE *file);
12729 int ftrylockfile(FILE *file);
12730 void funlockfile(FILE *file);
```

12731 **DESCRIPTION**

12732 These functions shall provide for explicit application-level locking of stdio (**FILE ***) objects.
 12733 These functions can be used by a thread to delineate a sequence of I/O statements that are
 12734 executed as a unit.

12735 The *flockfile()* function shall acquire for a thread ownership of a (**FILE ***) object.

12736 The *ftrylockfile()* function shall acquire for a thread ownership of a (**FILE ***) object if the object is
 12737 available; *ftrylockfile()* is a non-blocking version of *flockfile()*.

12738 The *funlockfile()* function shall relinquish the ownership granted to the thread. The behavior is
 12739 undefined if a thread other than the current owner calls the *funlockfile()* function.

12740 The functions shall behave as if there is a lock count associated with each (**FILE ***) object. This
 12741 count is implicitly initialized to zero when the (**FILE ***) object is created. The (**FILE ***) object is
 12742 unlocked when the count is zero. When the count is positive, a single thread owns the (**FILE ***)
 12743 object. When the *flockfile()* function is called, if the count is zero or if the count is positive and
 12744 the caller owns the (**FILE ***) object, the count shall be incremented. Otherwise, the calling thread
 12745 shall be suspended, waiting for the count to return to zero. Each call to *funlockfile()* shall
 12746 decrement the count. This allows matching calls to *flockfile()* (or successful calls to *ftrylockfile()*)
 12747 and *funlockfile()* to be nested.

12748 All functions that reference (**FILE ***) objects shall behave as if they use *flockfile()* and *funlockfile()*
 12749 internally to obtain ownership of these (**FILE ***) objects.

12750 **RETURN VALUE**

12751 None for *flockfile()* and *funlockfile()*.

12752 The *ftrylockfile()* function shall return zero for success and non-zero to indicate that the lock
 12753 cannot be acquired.

12754 **ERRORS**

12755 No errors are defined.

12756 **EXAMPLES**

12757 None.

12758 **APPLICATION USAGE**

12759 Applications using these functions may be subject to priority inversion, as discussed in the Base
 12760 Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

12761 **RATIONALE**

12762 The *flockfile()* and *funlockfile()* functions provide an orthogonal mutual-exclusion lock for each
 12763 **FILE**. The *ftrylockfile()* function provides a non-blocking attempt to acquire a file lock,
 12764 analogous to *pthread_mutex_trylock()*.

12765 These locks behave as if they are the same as those used internally by *stdio* for thread-safety.
 12766 This both provides thread-safety of these functions without requiring a second level of internal

12767 locking and allows functions in *stdio* to be implemented in terms of other *stdio* functions.

12768 Application writers and implementors should be aware that there are potential deadlock
 12769 problems on **FILE** objects. For example, the line-buffered flushing semantics of *stdio* (requested
 12770 via `{_IOLBF}`) require that certain input operations sometimes cause the buffered contents of
 12771 implementation-defined line-buffered output streams to be flushed. If two threads each hold the
 12772 lock on the other's **FILE**, deadlock ensues. This type of deadlock can be avoided by acquiring
 12773 **FILE** locks in a consistent order. In particular, the line-buffered output stream deadlock can
 12774 typically be avoided by acquiring locks on input streams before locks on output streams if a
 12775 thread would be acquiring both.

12776 In summary, threads sharing *stdio* streams with other threads can use *flockfile()* and *funlockfile()*
 12777 to cause sequences of I/O performed by a single thread to be kept bundled. The only case where
 12778 the use of *flockfile()* and *funlockfile()* is required is to provide a scope protecting uses of the
 12779 `*_unlocked` functions/macros. This moves the cost/performance tradeoff to the optimal point.

12780 **FUTURE DIRECTIONS**

12781 None.

12782 **SEE ALSO**

12783 *getc_unlocked()*, *putc_unlocked()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdio.h>`

12784 **CHANGE HISTORY**

12785 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

12786 **Issue 6**

12787 These functions are marked as part of the Thread-Safe Functions option.

12788 **Issue 7**

12789 The *flockfile()*, *ftrylockfile()*, and *funlockfile()* functions are moved from the Thread-Safe Functions
 12790 option to the Base.

12791 **NAME**

12792 floor, floorf, floorl — floor function

12793 **SYNOPSIS**

12794 #include <math.h>

12795 double floor(double x);

12796 float floorf(float x);

12797 long double floorl(long double x);

12798 **DESCRIPTION**12799 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12800 conflict between the requirements described here and the ISO C standard is unintentional. This
12801 volume of IEEE Std 1003.1-200x defers to the ISO C standard.12802 These functions shall compute the largest integral value not greater than x .12803 An application wishing to check for error situations should set *errno* to zero and call
12804 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
12805 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
12806 zero, an error has occurred.12807 **RETURN VALUE**12808 Upon successful completion, these functions shall return the largest integral value not greater
12809 than x , expressed as a **double**, **float**, or **long double**, as appropriate for the return type of the
12810 function.12811 MX If x is NaN, a NaN shall be returned.12812 If x is ± 0 or $\pm \text{Inf}$, x shall be returned.12813 XSI If the correct value would cause overflow, a range error shall occur and *floor()*, *floorf()*, and
12814 *floorl()* shall return the value of the macro `-HUGE_VAL`, `-HUGE_VALF`, and `-HUGE_VALL`,
12815 respectively.12816 **ERRORS**

12817 These functions shall fail if:

12818 XSI **Range Error** The result would cause an overflow.12819 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
12820 then *errno* shall be set to [ERANGE]. If the integer expression
12821 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
12822 floating-point exception shall be raised.12823 **EXAMPLES**

12824 None.

12825 **APPLICATION USAGE**12826 The integral value returned by these functions might not be expressible as an **int** or **long**. The
12827 return value should be tested before assigning it to an integer type to avoid the undefined
12828 results of an integer overflow.12829 The *floor()* function can only overflow when the floating-point representation has
12830 `DBL_MANT_DIG > DBL_MAX_EXP`.12831 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
12832 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

12833

RATIONALE

12834

None.

12835

FUTURE DIRECTIONS

12836

None.

12837

SEE ALSO

12838

ceil(), *feclearexcept()*, *fetetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

12839

12840

CHANGE HISTORY

12841

First released in Issue 1. Derived from Issue 1 of the SVID.

12842

Issue 5

12843

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

12844

12845

Issue 6

12846

The *floorf()* and *floorl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

12847

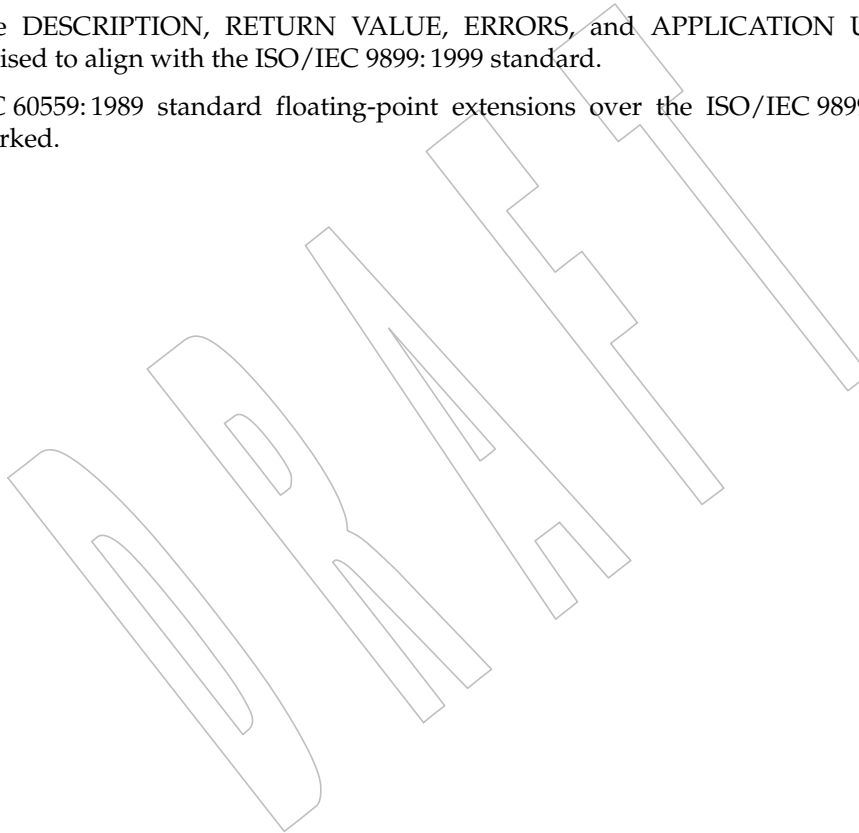
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

12848

12849

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

12850



12851 **NAME**12852 `fma, fmaf, fmal` — floating-point multiply-add12853 **SYNOPSIS**12854 `#include <math.h>`12855 `double fma(double x, double y, double z);`12856 `float fmaf(float x, float y, float z);`12857 `long double fmal(long double x, long double y, long double z);`12858 **DESCRIPTION**12859 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
12860 conflict between the requirements described here and the ISO C standard is unintentional. This
12861 volume of IEEE Std 1003.1-200x defers to the ISO C standard.12862 These functions shall compute $(x * y) + z$, rounded as one ternary operation: they shall compute
12863 the value (as if) to infinite precision and round once to the result format, according to the
12864 rounding mode characterized by the value of `FLT_ROUNDS`.12865 An application wishing to check for error situations should set `errno` to zero and call
12866 `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `errno` is non-zero or
12867 `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-
12868 zero, an error has occurred.12869 **RETURN VALUE**12870 Upon successful completion, these functions shall return $(x * y) + z$, rounded as one ternary
12871 operation.12872 **MX** If the result overflows or underflows, a range error may occur. On systems that support the IEC
12873 60559 Floating-Point option, if the result overflows a range error shall occur.12874 If x or y are NaN, a NaN shall be returned.12875 If x multiplied by y is an exact infinity and z is also an infinity but with the opposite sign, a
12876 domain error shall occur, and either a NaN (if supported), or an implementation-defined value
12877 shall be returned.12878 If one of x and y is infinite, the other is zero, and z is not a NaN, a domain error shall occur, and
12879 either a NaN (if supported), or an implementation-defined value shall be returned.12880 If one of x and y is infinite, the other is zero, and z is a NaN, a NaN shall be returned and a
12881 domain error may occur.12882 If $x*y$ is not $0*Inf$ nor $Inf*0$ and z is a NaN, a NaN shall be returned.12883 **ERRORS**

12884 These functions shall fail if:

12885 **MX** **Domain Error** The value of $x*y+z$ is invalid, or the value $x*y$ is invalid and z is not a NaN.12886 If the integer expression `(math_errhandling & MATH_ERRNO)` is non-zero,
12887 then `errno` shall be set to [EDOM]. If the integer expression `(math_errhandling`
12888 `& MATH_ERREXCEPT)` is non-zero, then the invalid floating-point exception
12889 shall be raised.12890 **MX** **Range Error** The result overflows.12891 If the integer expression `(math_errhandling & MATH_ERRNO)` is non-zero,
12892 then `errno` shall be set to [ERANGE]. If the integer expression
12893 `(math_errhandling & MATH_ERREXCEPT)` is non-zero, then the overflow

12894		floating-point exception shall be raised.
12895		These functions may fail if:
12896	MX	Domain Error The value $x*y$ is invalid and z is a NaN.
12897		If the integer expression (<i>math_errhandling</i> & MATH_ERRNO) is non-zero,
12898		then <i>errno</i> shall be set to [EDOM]. If the integer expression (<i>math_errhandling</i>
12899		& MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
12900		shall be raised.
12901		Range Error The result underflows.
12902		If the integer expression (<i>math_errhandling</i> & MATH_ERRNO) is non-zero,
12903		then <i>errno</i> shall be set to [ERANGE]. If the integer expression
12904		(<i>math_errhandling</i> & MATH_ERREXCEPT) is non-zero, then the underflow
12905		floating-point exception shall be raised.
12906		Range Error The result overflows.
12907		If the integer expression (<i>math_errhandling</i> & MATH_ERRNO) is non-zero,
12908		then <i>errno</i> shall be set to [ERANGE]. If the integer expression
12909		(<i>math_errhandling</i> & MATH_ERREXCEPT) is non-zero, then the overflow
12910		floating-point exception shall be raised.
12911		EXAMPLES
12912		None.
12913		APPLICATION USAGE
12914		On error, the expressions (<i>math_errhandling</i> & MATH_ERRNO) and (<i>math_errhandling</i> &
12915		MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.
12916		RATIONALE
12917		In many cases, clever use of floating (<i>fused</i>) multiply-add leads to much improved code; but its
12918		unexpected use by the compiler can undermine carefully written code. The FP_CONTRACT
12919		macro can be used to disallow use of floating multiply-add; and the <i>fma()</i> function guarantees
12920		its use where desired. Many current machines provide hardware floating multiply-add
12921		instructions; software implementation can be used for others.
12922		FUTURE DIRECTIONS
12923		None.
12924		SEE ALSO
12925		<i>feclearexcept()</i> , <i>fetestexcept()</i> , the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18,
12926		Treatment of Error Conditions for Mathematical Functions, < math.h >
12927		CHANGE HISTORY
12928		First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.
12929		Issue 7
12930		ISO/IEC 9899:1999 standard, Technical Corrigendum 2 #57 (SD5-XSH-ERN-69) is applied,
12931		adding a “may fail” range error for non-MX systems.

12932	NAME	
12933		fmax, fmaxf, fmaxl — determine maximum numeric value of two floating-point numbers
12934	SYNOPSIS	
12935		#include <math.h>
12936		double fmax(double x, double y);
12937		float fmaxf(float x, float y);
12938		long double fmaxl(long double x, long double y);
12939	DESCRIPTION	
12940	CX	The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.
12941		
12942		
12943	MX	These functions shall determine the maximum numeric value of their arguments. NaN arguments shall be treated as missing data: if one argument is a NaN and the other numeric, then these functions shall choose the numeric value.
12944		
12945		
12946	RETURN VALUE	
12947		Upon successful completion, these functions shall return the maximum numeric value of their arguments.
12948		
12949	MX	If just one argument is a NaN, the other argument shall be returned.
12950		If <i>x</i> and <i>y</i> are NaN, a NaN shall be returned.
12951	ERRORS	
12952		No errors are defined.
12953	EXAMPLES	
12954		None.
12955	APPLICATION USAGE	
12956		None.
12957	RATIONALE	
12958		None.
12959	FUTURE DIRECTIONS	
12960		None.
12961	SEE ALSO	
12962		<i>fdim()</i> , <i>fmin()</i> , the Base Definitions volume of IEEE Std 1003.1-200x, <math.h>
12963	CHANGE HISTORY	
12964		First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.
12965	Issue 7	
12966		Austin Group Interpretation 1003.1-2001 #007 is applied.

12967 **NAME**12968 `fmemopen` — open a memory buffer stream12969 **SYNOPSIS**

```
12970 CX #include <stdio.h>
12971 FILE *fmemopen(void *restrict buf, size_t size,
12972               const char *restrict mode);
```

12973 **DESCRIPTION**

12974 The `fmemopen()` function shall associate the buffer given by the `buf` and `size` arguments with a
 12975 stream. The `buf` argument shall be either a null pointer or point to a buffer that is at least `size`
 12976 bytes long.

12977 The `mode` argument is a character string having one of the following values:

12978	<code>r</code> or <code>rb</code>	Open the stream for reading.
12979	<code>w</code> or <code>wb</code>	Open the stream for writing.
12980	<code>a</code> or <code>ab</code>	Append; open the stream for writing at the first null byte.
12981	<code>r+</code> or <code>rb+</code> or <code>r+b</code>	Open the stream for update (reading and writing).
12982	<code>w+</code> or <code>wb+</code> or <code>w+b</code>	Open the stream for update (reading and writing). Truncate the buffer 12983 contents.
12984	<code>a+</code> or <code>ab+</code> or <code>a+b</code>	Append; open the stream for update (reading and writing); the initial 12985 position is at the first null byte.

12986 The character '`b`' shall have no effect.

12987 If a null pointer is specified as the `buf` argument, `fmemopen()` shall allocate `size` bytes of memory
 12988 as if by a call to `malloc()`. This buffer shall be automatically freed when the stream is closed.
 12989 Because this feature is only useful when the stream is opened for updating (because there is no
 12990 way to get a pointer to the buffer) the `fmemopen()` call may fail if the `mode` argument does not
 12991 include a '+'.

12992 The stream maintains a current position in the buffer. This position is initially set to either the
 12993 beginning of the buffer (for `r` and `w` modes) or to the first null byte in the buffer (for `a` modes). If
 12994 no null byte is found in append mode, the initial position is set to one byte after the end of the
 12995 buffer.

12996 If `buf` is a null pointer, the initial position shall always be set to the beginning of the buffer.

12997 The stream also maintains the size of the current buffer contents. For modes `r` and `r+` the size is
 12998 set to the value given by the `size` argument. For modes `w` and `w+` the initial size is zero and for
 12999 modes `a` and `a+` the initial size is either the position of the first null byte in the buffer or the value
 13000 of the `size` argument if no null byte is found.

13001 A read operation on the stream cannot advance the current buffer position behind the current
 13002 buffer size. Reaching the buffer size in a read operation counts as "end-of-file". Null bytes in the
 13003 buffer have no special meaning for reads. The read operation starts at the current buffer position
 13004 of the stream.

13005 A write operation starts either at the current position of the stream (if mode has not specified
 13006 '`a`' as the first character) or at the current size of the stream (if mode had '`a`' as the first
 13007 character). If the current position at the end of the write is larger than the current buffer size, the
 13008 current buffer size is set to the current position. A write operation on the stream cannot advance

13009 the current buffer size behind the size given in the *size* argument.

13010 When a stream open for writing is flushed or closed, a null byte is written at the current position
 13011 or at the end of the buffer, depending on the size of the contents. If a stream open for update is
 13012 flushed or closed and the last write has advanced the current buffer size, a null byte is written at
 13013 the end of the buffer if it fits.

13014 An attempt to seek a memory buffer stream to a negative position or to a position larger than the
 13015 buffer size given in the *size* argument shall fail.

13016 RETURN VALUE

13017 Upon successful completion, *fmemopen()* shall return a pointer to the object controlling the
 13018 stream. Otherwise, a null pointer shall be returned, and *errno* shall be set to indicate the error.

13019 ERRORS

13020 The *fmemopen()* function shall fail if:

13021 [EINVAL] The *size* argument specifies a buffer size of zero.

13022 The *fmemopen()* function may fail if:

13023 [EINVAL] The value of the *mode* argument is not valid.

13024 [EINVAL] The *buf* argument is a null pointer and the *mode* argument does not include a
 13025 '+' character.

13026 [ENOMEM] The *buf* argument is a null pointer and the allocation of a buffer of length *size*
 13027 has failed.

13028 [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

13029 EXAMPLES

```

13030 #include <stdio.h>
13031 static char buffer[] = "foobar";
13032 int
13033 main (void)
13034 {
13035     int ch;
13036     FILE *stream;
13037     stream = fmemopen(buffer, strlen (buffer), "r");
13038     if (stream == NULL)
13039         /* handle error */;
13040     while ((ch = fgetc(stream)) != EOF)
13041         printf("Got %c\n", ch);
13042     fclose(stream);
13043     return (0);
13044 }
  
```

13045 This program produces the following output:

```

13046 Got f
13047 Got o
13048 Got o
13049 Got b
13050 Got a
13051 Got r
  
```

fmemopen()

13052

APPLICATION USAGE

13053

None.

13054

RATIONALE

13055

This interface has been introduced to eliminate many of the errors encountered in the construction of strings, notably overflowing of strings. This interface prevents overflow.

13056

13057

FUTURE DIRECTIONS

13058

None.

13059

SEE ALSO

13060

fdopen(), *fopen()*, *freopen()*, *malloc()*, *open_memstream()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

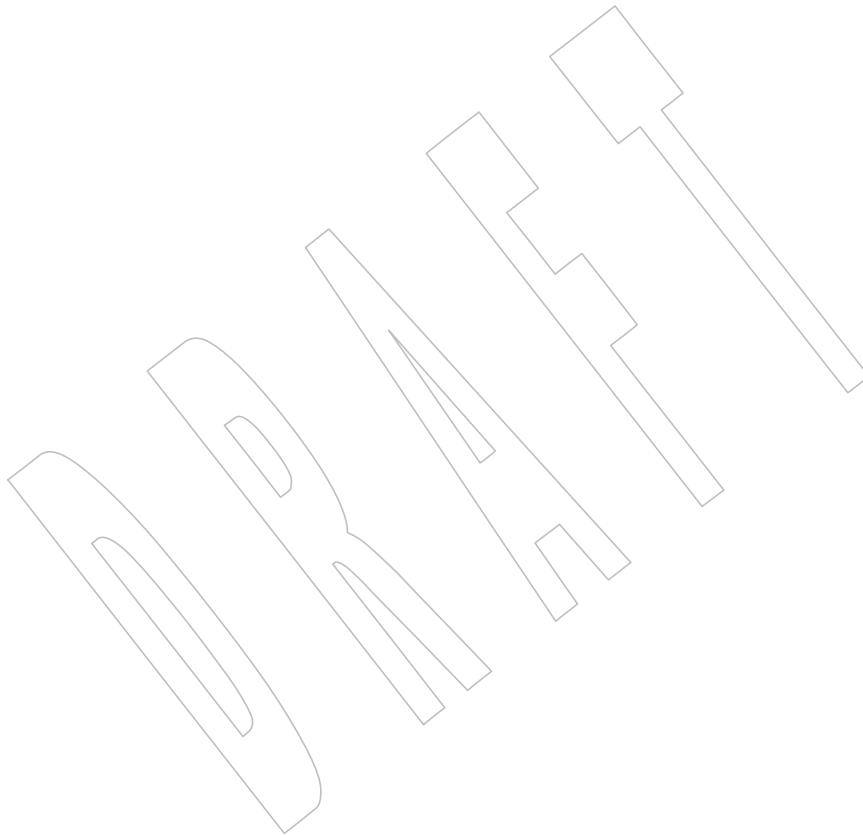
13061

13062

CHANGE HISTORY

13063

First released in Issue 7.



13064 **NAME**

13065 fmin, fminf, fminl — determine minimum numeric value of two floating-point numbers

13066 **SYNOPSIS**

13067 #include <math.h>

13068 double fmin(double x, double y);

13069 float fminf(float x, float y);

13070 long double fminl(long double x, long double y);

13071 **DESCRIPTION**

13072 CX The functionality described on this reference page is aligned with the ISO C standard. Any

13073 conflict between the requirements described here and the ISO C standard is unintentional. This

13074 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

13075 MX These functions shall determine the minimum numeric value of their arguments. NaN

13076 arguments shall be treated as missing data: if one argument is a NaN and the other numeric,

13077 then these functions shall choose the numeric value.

13078 **RETURN VALUE**

13079 Upon successful completion, these functions shall return the minimum numeric value of their

13080 arguments.

13081 MX If just one argument is a NaN, the other argument shall be returned.

13082 If *x* and *y* are NaN, a NaN shall be returned.

13083 **ERRORS**

13084 No errors are defined.

13085 **EXAMPLES**

13086 None.

13087 **APPLICATION USAGE**

13088 None.

13089 **RATIONALE**

13090 None.

13091 **FUTURE DIRECTIONS**

13092 None.

13093 **SEE ALSO**

13094 *fdim()*, *fmax()*, the Base Definitions volume of IEEE Std 1003.1-200x, <math.h>

13095 **CHANGE HISTORY**

13096 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

13097 **Issue 7**

13098 Austin Group Interpretation 1003.1-2001 #008 is applied.

13099 **NAME**13100 `fmod, fmodf, fmodl` — floating-point remainder value function13101 **SYNOPSIS**13102 `#include <math.h>`13103 `double fmod(double x, double y);`13104 `float fmodf(float x, float y);`13105 `long double fmodl(long double x, long double y);`13106 **DESCRIPTION**13107 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
13108 conflict between the requirements described here and the ISO C standard is unintentional. This
13109 volume of IEEE Std 1003.1-200x defers to the ISO C standard.13110 These functions shall return the floating-point remainder of the division of x by y .13111 An application wishing to check for error situations should set `errno` to zero and call
13112 `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `errno` is non-zero or
13113 `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-
13114 zero, an error has occurred.13115 **RETURN VALUE**13116 These functions shall return the value $x - i * y$, for some integer i such that, if y is non-zero, the
13117 result has the same sign as x and magnitude less than the magnitude of y .13118 If the correct value would cause underflow, and is not representable, a range error may occur,
13119 **MX** and either 0.0 (if supported), or an implementation-defined value shall be returned.13120 **MX** If x or y is NaN, a NaN shall be returned.13121 If y is zero, a domain error shall occur, and either a NaN (if supported), or an implementation-
13122 defined value shall be returned.13123 If x is infinite, a domain error shall occur, and either a NaN (if supported), or an
13124 implementation-defined value shall be returned.13125 If x is ± 0 and y is not zero, ± 0 shall be returned.13126 If x is not infinite and y is $\pm \text{Inf}$, x shall be returned.13127 If the correct value would cause underflow, and is representable, a range error may occur and
13128 the correct value shall be returned.13129 **ERRORS**

13130 These functions shall fail if:

13131 **MX** **Domain Error** The x argument is infinite or y is zero.13132 If the integer expression `(math_errhandling & MATH_ERRNO)` is non-zero,
13133 then `errno` shall be set to [EDOM]. If the integer expression `(math_errhandling`
13134 `& MATH_ERREXCEPT)` is non-zero, then the invalid floating-point exception
13135 shall be raised.

13136 These functions may fail if:

13137 **Range Error** The result underflows.13138 If the integer expression `(math_errhandling & MATH_ERRNO)` is non-zero,
13139 then `errno` shall be set to [ERANGE]. If the integer expression
13140 `(math_errhandling & MATH_ERREXCEPT)` is non-zero, then the underflow

13141 floating-point exception shall be raised.

13142 **EXAMPLES**

13143 None.

13144 **APPLICATION USAGE**

13145 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
13146 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

13147 **RATIONALE**

13148 None.

13149 **FUTURE DIRECTIONS**

13150 None.

13151 **SEE ALSO**

13152 *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-200x,
13153 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

13154 **CHANGE HISTORY**

13155 First released in Issue 1. Derived from Issue 1 of the SVID.

13156 **Issue 5**

13157 The DESCRIPTION is updated to indicate how an application should check for an error. This
13158 text was previously published in the APPLICATION USAGE section.

13159 **Issue 6**

13160 The behavior for when the *y* argument is zero is now defined.

13161 The *fmodf()* and *fmodl()* functions are added for alignment with the ISO/IEC 9899:1999
13162 standard.

13163 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
13164 revised to align with the ISO/IEC 9899:1999 standard.

13165 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
13166 marked.

13167 **NAME**13168 `fmtmsg` — display a message in the specified format on standard error and/or a system console13169 **SYNOPSIS**

```
13170 XSI #include <fmtmsg.h>
13171
13171 int fmtmsg(long classification, const char *label, int severity,
13172           const char *text, const char *action, const char *tag);
```

13173 **DESCRIPTION**13174 The `fmtmsg()` function shall display messages in a specified format instead of the traditional
13175 `printf()` function.13176 Based on a message's classification component, `fmtmsg()` shall write a formatted message either
13177 to standard error, to the console, or to both.13178 A formatted message consists of up to five components as defined below. The component
13179 *classification* is not part of a message displayed to the user, but defines the source of the message
13180 and directs the display of the formatted message.

13181 *classification* Contains the sum of identifying values constructed from the constants defined
13182 below. Any one identifier from a subclass may be used in combination with a
13183 single identifier from a different subclass. Two or more identifiers from the
13184 same subclass should not be used together, with the exception of identifiers
13185 from the display subclass. (Both display subclass identifiers may be used so
13186 that messages can be displayed to both standard error and the system
13187 console.)

13188 **Major Classifications**13189 Identifies the source of the condition. Identifiers are: MM_HARD
13190 (hardware), MM_SOFT (software), and MM_FIRM (firmware).13191 **Message Source Subclassifications**13192 Identifies the type of software in which the problem is detected.
13193 Identifiers are: MM_APPL (application), MM_UTIL (utility), and
13194 MM_OPSSYS (operating system).13195 **Display Subclassifications**13196 Indicates where the message is to be displayed. Identifiers are:
13197 MM_PRINT to display the message on the standard error stream,
13198 MM_CONSOLE to display the message on the system console. One or
13199 both identifiers may be used.13200 **Status Subclassifications**13201 Indicates whether the application can recover from the condition.
13202 Identifiers are: MM_RECOVER (recoverable) and MM_NRECOV (non-
13203 recoverable).13204 An additional identifier, MM_NULLMC, indicates that no classification
13205 component is supplied for the message.13206 *label* Identifies the source of the message. The format is two fields separated by a
13207 colon. The first field is up to 10 bytes, the second is up to 14 bytes.13208 *severity* Indicates the seriousness of the condition. Identifiers for the levels of *severity*
13209 are:

13210		MM_HALT	Indicates that the application has encountered a severe fault and is halting. Produces the string "HALT".
13211			
13212		MM_ERROR	Indicates that the application has detected a fault. Produces the string "ERROR".
13213			
13214		MM_WARNING	Indicates a condition that is out of the ordinary, that might be a problem, and should be watched. Produces the string "WARNING".
13215			
13216			
13217		MM_INFO	Provides information about a condition that is not in error. Produces the string "INFO".
13218			
13219		MM_NOSEV	Indicates that no severity level is supplied for the message.
13220	<i>text</i>		Describes the error condition that produced the message. The character string is not limited to a specific size. If the character string is empty, then the text produced is unspecified.
13221			
13222			
13223	<i>action</i>		Describes the first step to be taken in the error-recovery process. The <i>fmtmsg()</i> function precedes the action string with the prefix: "TO FIX:". The <i>action</i> string is not limited to a specific size.
13224			
13225			
13226	<i>tag</i>		An identifier that references on-line documentation for the message. Suggested usage is that <i>tag</i> includes the <i>label</i> and a unique identifying number. A sample <i>tag</i> is "XSI:cat:146".
13227			
13228			

The *MSGVERB* environment variable (for message verbosity) shall determine for *fmtmsg()* which message components it is to select when writing messages to standard error. The value of *MSGVERB* shall be a colon-separated list of optional keywords. Valid keywords are: *label*, *severity*, *text*, *action*, and *tag*. If *MSGVERB* contains a keyword for a component and the component's value is not the component's null value, *fmtmsg()* shall include that component in the message when writing the message to standard error. If *MSGVERB* does not include a keyword for a message component, that component shall not be included in the display of the message. The keywords may appear in any order. If *MSGVERB* is not defined, if its value is the null string, if its value is not of the correct format, or if it contains keywords other than the valid ones listed above, *fmtmsg()* shall select all components.

MSGVERB shall determine which components are selected for display to standard error. All message components shall be included in console messages.

RETURN VALUE

The *fmtmsg()* function shall return one of the following values:

13243	MM_OK	The function succeeded.
13244	MM_NOTOK	The function failed completely.
13245	MM_NOMSG	The function was unable to generate a message on standard error, but otherwise succeeded.
13246		
13247	MM_NOCON	The function was unable to generate a console message, but otherwise succeeded.
13248		

ERRORS

None.

13251

EXAMPLES

13252

1. The following example of *fmtmsg()*:

13253

```
fmtmsg(MM_PRINT, "XSI:cat", MM_ERROR, "illegal option",
13254 "refer to cat in user's reference manual", "XSI:cat:001")
```

13255

produces a complete message in the specified message format:

13256

```
XSI:cat: ERROR: illegal option
```

13257

```
TO FIX: refer to cat in user's reference manual XSI:cat:001
```

13258

2. When the environment variable *MSGVERB* is set as follows:

13259

```
MSGVERB=severity:text:action
```

13260

and Example 1 is used, *fmtmsg()* produces:

13261

```
ERROR: illegal option
```

13262

```
TO FIX: refer to cat in user's reference manual
```

13263

APPLICATION USAGE

13264

One or more message components may be systematically omitted from messages generated by an application by using the null value of the argument for that component.

13265

13266

RATIONALE

13267

None.

13268

FUTURE DIRECTIONS

13269

None.

13270

SEE ALSO

13271

printf(), the Base Definitions volume of IEEE Std 1003.1-200x, <*fmtmsg.h*>

13272

CHANGE HISTORY

13273

First released in Issue 4, Version 2.

13274

Issue 5

13275

Moved from X/OPEN UNIX extension to BASE.

13276 **NAME**13277 `fnmatch` — match a filename or a pathname13278 **SYNOPSIS**13279 `#include <fnmatch.h>`13280 `int fnmatch(const char *pattern, const char *string, int flags);`13281 **DESCRIPTION**

13282 The `fnmatch()` function shall match patterns as described in the Shell and Utilities volume of
 13283 IEEE Std 1003.1-200x, Section 2.13.1, Patterns Matching a Single Character, and Section 2.13.2,
 13284 Patterns Matching Multiple Characters. It checks the string specified by the `string` argument to
 13285 see if it matches the pattern specified by the `pattern` argument.

13286 The `flags` argument shall modify the interpretation of `pattern` and `string`. It is the bitwise-
 13287 inclusive OR of zero or more of the flags defined in `<fnmatch.h>`. If the `FNM_PATHNAME` flag
 13288 is set in `flags`, then a slash character (`'/'`) in `string` shall be explicitly matched by a slash in
 13289 `pattern`; it shall not be matched by either the asterisk or question-mark special characters, nor by
 13290 a bracket expression. If the `FNM_PATHNAME` flag is not set, the slash character shall be treated
 13291 as an ordinary character.

13292 If `FNM_NOESCAPE` is not set in `flags`, a backslash character (`'\'`) in `pattern` followed by any
 13293 other character shall match that second character in `string`. In particular, `"\\\"` shall match a
 13294 backslash in `string`. If `FNM_NOESCAPE` is set, a backslash character shall be treated as an
 13295 ordinary character.

13296 If `FNM_PERIOD` is set in `flags`, then a leading period (`'.'`) in `string` shall match a period in
 13297 `pattern`; as described by rule 2 in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section
 13298 2.13.3, Patterns Used for Filename Expansion where the location of “leading” is indicated by the
 13299 value of `FNM_PATHNAME`:

- 13300 • If `FNM_PATHNAME` is set, a period is “leading” if it is the first character in `string` or if it
- 13301 immediately follows a slash.
- 13302 • If `FNM_PATHNAME` is not set, a period is “leading” only if it is the first character of
- 13303 `string`.

13304 If `FNM_PERIOD` is not set, then no special restrictions are placed on matching a period.

13305 **RETURN VALUE**

13306 If `string` matches the pattern specified by `pattern`, then `fnmatch()` shall return 0. If there is no
 13307 match, `fnmatch()` shall return `FNM_NOMATCH`, which is defined in `<fnmatch.h>`. If an error
 13308 occurs, `fnmatch()` shall return another non-zero value.

13309 **ERRORS**

13310 No errors are defined.

13311 **EXAMPLES**

13312 None.

13313 **APPLICATION USAGE**

13314 The `fnmatch()` function has two major uses. It could be used by an application or utility that
 13315 needs to read a directory and apply a pattern against each entry. The `find` utility is an example of
 13316 this. It can also be used by the `pax` utility to process its `pattern` operands, or by applications that
 13317 need to match strings in a similar manner.

13318 The name `fnmatch()` is intended to imply *filename* match, rather than *pathname* match. The
 13319 default action of this function is to match filenames, rather than pathnames, since it gives no
 13320 special significance to the slash character. With the `FNM_PATHNAME` flag, `fnmatch()` does

fnmatch()

13321 match pathnames, but without tilde expansion, parameter expansion, or special treatment for a
 13322 period at the beginning of a filename.

RATIONALE

13323 This function replaced the REG_FILENAME flag of *regcomp()* in early proposals of this volume
 13324 of IEEE Std 1003.1-200x. It provides virtually the same functionality as the *regcomp()* and
 13325 *regexexec()* functions using the REG_FILENAME and REG_FSLASH flags (the REG_FSLASH flag
 13326 was proposed for *regcomp()*, and would have had the opposite effect from FNM_PATHNAME),
 13327 but with a simpler function and less system overhead.
 13328

FUTURE DIRECTIONS

13329 None.
 13330

SEE ALSO

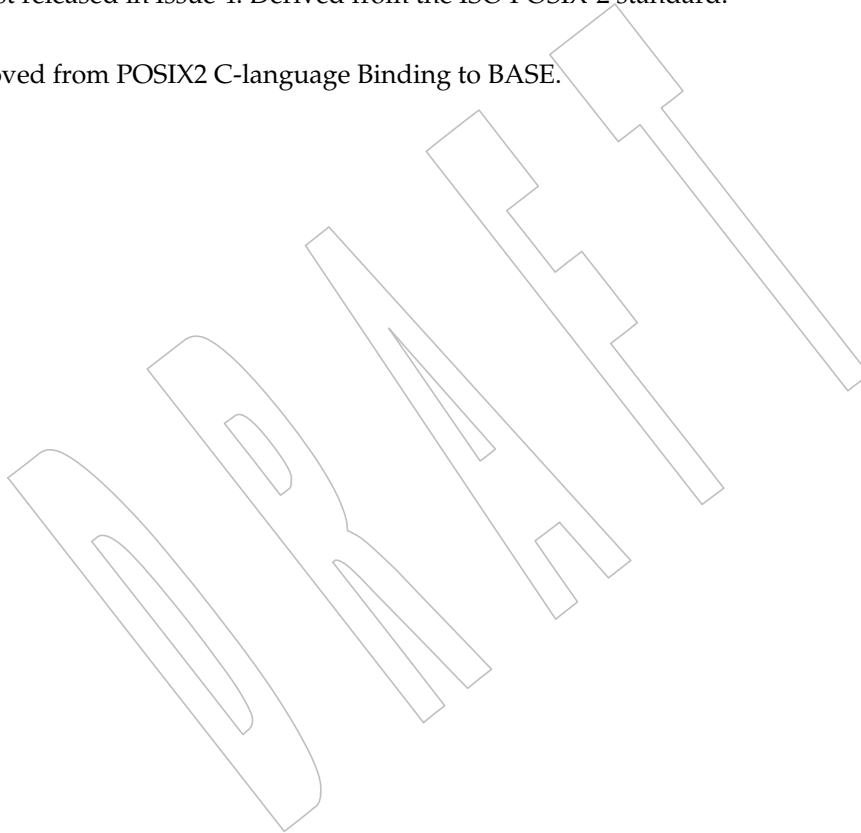
13331 *glob()*, *wordexp()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<fnmatch.h>**, the Shell
 13332 and Utilities volume of IEEE Std 1003.1-200x
 13333

CHANGE HISTORY

13334 First released in Issue 4. Derived from the ISO POSIX-2 standard.
 13335

Issue 5

13336 Moved from POSIX2 C-language Binding to BASE.
 13337



13338 **NAME**

13339 fopen — open a stream

13340 **SYNOPSIS**

13341 #include <stdio.h>

13342 FILE *fopen(const char *restrict filename, const char *restrict mode);

13343 **DESCRIPTION**

13344 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 13345 conflict between the requirements described here and the ISO C standard is unintentional. This
 13346 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

13347 The *fopen()* function shall open the file whose pathname is the string pointed to by *filename*, and
 13348 associates a stream with it.

13349 The *mode* argument points to a string. If the string is one of the following, the file shall be opened
 13350 in the indicated mode. Otherwise, the behavior is undefined.

13351 *r* or *rb* Open file for reading.

13352 *w* or *wb* Truncate to zero length or create file for writing.

13353 *a* or *ab* Append; open or create file for writing at end-of-file.

13354 *r+* or *rb+* or *r+b* Open file for update (reading and writing).

13355 *w+* or *wb+* or *w+b* Truncate to zero length or create file for update.

13356 *a+* or *ab+* or *a+b* Append; open or create file for update, writing at end-of-file.

13357 CX The character 'b' shall have no effect, but is allowed for ISO C standard conformance. Opening
 13358 a file with read mode (*r* as the first character in the *mode* argument) shall fail if the file does not
 13359 exist or cannot be read.

13360 Opening a file with append mode (*a* as the first character in the *mode* argument) shall cause all
 13361 subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening
 13362 calls to *fseek()*.

13363 When a file is opened with update mode ('+' as the second or third character in the *mode*
 13364 argument), both input and output may be performed on the associated stream. However, the
 13365 application shall ensure that output is not directly followed by input without an intervening call
 13366 to *fflush()* or to a file positioning function (*fseek()*, *fsetpos()*, or *rewind()*), and input is not directly
 13367 followed by output without an intervening call to a file positioning function, unless the input
 13368 operation encounters end-of-file.

13369 When opened, a stream is fully buffered if and only if it can be determined not to refer to an
 13370 interactive device. The error and end-of-file indicators for the stream shall be cleared.

13371 CX If *mode* is *w*, *wb*, *a*, *ab*, *w+*, *wb+*, *w+b*, *a+*, *ab+*, or *a+b*, and the file did not previously exist, upon
 13372 successful completion, the *fopen()* function shall mark for update the *st_atime*, *st_ctime*, and
 13373 *st_mtime* fields of the file and the *st_ctime* and *st_mtime* fields of the parent directory.

13374 If *mode* is *w*, *wb*, *a*, *ab*, *w+*, *wb+*, *w+b*, *a+*, *ab+*, or *a+b*, and the file did not previously exist, the
 13375 *fopen()* function shall create a file as if it called the *creat()* function with a value appropriate for
 13376 the *path* argument interpreted from *filename* and a value of S_IRUSR | S_IWUSR | S_IRGRP |
 13377 S_IWGRP | S_IROTH | S_IWOTH for the *mode* argument.

13378 If *mode* is *w*, *wb*, *w+*, *wb+*, or *w+b*, and the file did previously exist, upon successful completion,
 13379 *fopen()* shall mark for update the *st_ctime* and *st_mtime* fields of the file. The *fopen()* function
 13380 shall allocate a file descriptor as *open()* does.

fopen()

13381 XSI After a successful call to the *fopen()* function, the orientation of the stream shall be cleared, the
 13382 encoding rule shall be cleared, and the associated **mbstate_t** object shall be set to describe an
 13383 initial conversion state.

13384 CX The largest value that can be represented correctly in an object of type **off_t** shall be established
 13385 as the offset maximum in the open file description.

RETURN VALUE

13386 Upon successful completion, *fopen()* shall return a pointer to the object controlling the stream.
 13387
 13388 CX Otherwise, a null pointer shall be returned, and *errno* shall be set to indicate the error.

ERRORS

13389 The *fopen()* function shall fail if:

13391 CX [EACCES] Search permission is denied on a component of the path prefix, or the file
 13392 exists and the permissions specified by *mode* are denied, or the file does not
 13393 exist and write permission is denied for the parent directory of the file to be
 13394 created.

13395 CX [EINTR] A signal was caught during *fopen()*.

13396 CX [EISDIR] The named file is a directory and *mode* requires write access.

13397 CX [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 13398 argument.

13399 CX [EMFILE] All file descriptors available to the process are currently open.

13400 CX [ENAMETOOLONG]

13401 The length of the *filename* argument exceeds {PATH_MAX} or a pathname
 13402 component is longer than {NAME_MAX}.

13403 CX [ENFILE] The maximum allowable number of files is currently open in the system.

13404 CX [ENOENT] A component of *filename* does not name an existing file or *filename* is an empty
 13405 string.

13406 CX [ENOSPC] The directory or file system that would contain the new file cannot be
 13407 expanded, the file does not exist, and the file was to be created.

13408 CX [ENOTDIR] A component of the path prefix is not a directory.

13409 CX [ENXIO] The named file is a character special or block special file, and the device
 13410 associated with this special file does not exist.

13411 CX [EOVERFLOW] The named file is a regular file and the size of the file cannot be represented
 13412 correctly in an object of type **off_t**.

13413 CX [EROFS] The named file resides on a read-only file system and *mode* requires write
 13414 access.

13415 The *fopen()* function may fail if:

13416 CX [EINVAL] The value of the *mode* argument is not valid.

13417 CX [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 13418 resolution of the *path* argument.

13419 CX [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

13420 CX [EMFILE] {STREAM_MAX} streams are currently open in the calling process.

13421 CX [ENAMETOOLONG]

13422 Pathname resolution of a symbolic link produced an intermediate result
 13423 whose length exceeds {PATH_MAX}.

13424	CX	[ENOMEM]	Insufficient storage space is available.
13425	CX	[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and <i>mode</i>
13426			requires write access.

EXAMPLES**Opening a File**

The following example tries to open the file named **file** for reading. The *fopen()* function returns a file pointer that is used in subsequent *fgets()* and *fclose()* calls. If the program cannot open the file, it just ignores it.

```
#include <stdio.h>
...
FILE *fp;
...
void rgrep(const char *file)
{
...
    if ((fp = fopen(file, "r")) == NULL)
        return;
...
}
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

creat(), *fclose()*, *fdopen()*, *fmemopen()*, *freopen()*, *open_memstream()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file description. This change is to support large files.
- In the ERRORS section, the [Eoverflow] condition is added. This change is to support large files.
- The [ELOOP] mandatory error condition is added.
- The [EINVAL], [EMFILE], [ENAMETOOLONG], [ENOMEM], and [ETXTBSY] optional error conditions are added.

The normative text is updated to avoid use of the term “must” for application requirements.

13468

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

13469

- The prototype for *fopen()* is updated.

13470

- The DESCRIPTION is updated to note that if the argument *mode* points to a string other than those listed, then the behavior is undefined.

13471

13472

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

13473

13474

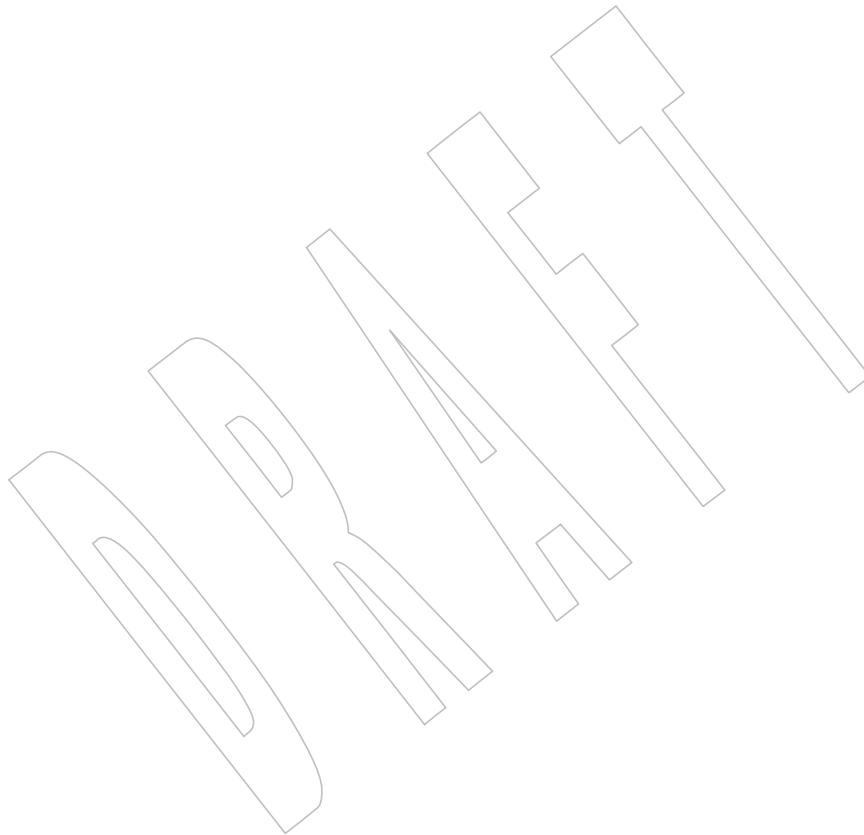
Issue 7

13475

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

13476

Austin Group Interpretation 1003.1-2001 #025 is applied, clarifying the file creation mode.



13477 **NAME**

13478 fork — create a new process

13479 **SYNOPSIS**

13480 #include <unistd.h>

13481 pid_t fork(void);

13482 **DESCRIPTION**13483 The *fork()* function shall create a new process. The new process (child process) shall be an exact
13484 copy of the calling process (parent process) except as detailed below:

- 13485 • The child process shall have a unique process ID.
- 13486 • The child process ID also shall not match any active process group ID.
- 13487 • The child process shall have a different parent process ID, which shall be the process ID of
13488 the calling process.
- 13489 • The child process shall have its own copy of the parent's file descriptors. Each of the
13490 child's file descriptors shall refer to the same open file description with the corresponding
13491 file descriptor of the parent.
- 13492 • The child process shall have its own copy of the parent's open directory streams. Each
13493 open directory stream in the child process may share directory stream positioning with the
13494 corresponding directory stream of the parent.
- 13495 • The child process shall have its own copy of the parent's message catalog descriptors.
- 13496 • The child process values of *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* shall be set to
13497 0.
- 13498 • The time left until an alarm clock signal shall be reset to zero, and the alarm, if any, shall be
13499 canceled; see *alarm()*.
- 13500 XSI • All *semadj* values shall be cleared.
- 13501 • File locks set by the parent process shall not be inherited by the child process.
- 13502 • The set of signals pending for the child process shall be initialized to the empty set.
- 13503 XSI • Interval timers shall be reset in the child process.
- 13504 • Any semaphores that are open in the parent process shall also be open in the child process.
- 13505 ML • The child process shall not inherit any address space memory locks established by the
13506 parent process via calls to *mlockall()* or *mlock()*.
- 13507 • Memory mappings created in the parent shall be retained in the child process.
13508 MAP_PRIVATE mappings inherited from the parent shall also be MAP_PRIVATE
13509 mappings in the child, and any modifications to the data in these mappings made by the
13510 parent prior to calling *fork()* shall be visible to the child. Any modifications to the data in
13511 MAP_PRIVATE mappings made by the parent after *fork()* returns shall be visible only to
13512 the parent. Modifications to the data in MAP_PRIVATE mappings made by the child shall
13513 be visible only to the child.
- 13514 PS • For the SCHED_FIFO and SCHED_RR scheduling policies, the child process shall inherit
13515 the policy and priority settings of the parent process during a *fork()* function. For other
13516 scheduling policies, the policy and priority settings on *fork()* are implementation-defined.

fork()

- 13517
- Per-process timers created by the parent shall not be inherited by the child process.
- 13518 MSG
- The child process shall have its own copy of the message queue descriptors of the parent. Each of the message descriptors of the child shall refer to the same open message queue description as the corresponding message descriptor of the parent.
- 13520
- No asynchronous input or asynchronous output operations shall be inherited by the child process. Any use of asynchronous control blocks created by the parent produces undefined behavior.
- 13521
- A process shall be created with a single thread. If a multi-threaded process calls *fork()*, the new process shall contain a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources. Consequently, to avoid errors, the child process may only execute async-signal-safe operations until such time as one of the *exec* functions is called. Fork handlers may be established by means of the *pthread_atfork()* function in order to maintain application invariants across *fork()* calls.
- 13522
- 13523
- 13524
- 13525
- 13526
- 13527
- 13528
- 13529
- 13530
- 13531
- 13532
- When the application calls *fork()* from a signal handler and any of the fork handlers registered by *pthread_atfork()* calls a function that is not asynch-signal-safe, the behavior is undefined.
- 13533 OB TRC TRI
- If the Trace option and the Trace Inherit option are both supported:
- 13534
- 13535
- 13536
- 13537
- 13538
- 13539
- 13540
- If the calling process was being traced in a trace stream that had its inheritance policy set to `POSIX_TRACE_INHERITED`, the child process shall be traced into that trace stream, and the child process shall inherit the parent's mapping of trace event names to trace event type identifiers. If the trace stream in which the calling process was being traced had its inheritance policy set to `POSIX_TRACE_CLOSE_FOR_CHILD`, the child process shall not be traced into that trace stream. The inheritance policy is set by a call to the *posix_trace_attr_setinherited()* function.
- 13541 OB TRC
- If the Trace option is supported, but the Trace Inherit option is not supported:
- 13542
- The child process shall not be traced into any of the trace streams of its parent process.
- 13543 OB TRC
- If the Trace option is supported, the child process of a trace controller process shall not
- 13544
- control the trace streams controlled by its parent process.
- 13545 CPT
- The initial value of the CPU-time clock of the child process shall be set to zero.
- 13546 TCT
- The initial value of the CPU-time clock of the single thread of the child process shall be set
- 13547
- to zero.

13548 All other process characteristics defined by IEEE Std 1003.1-200x shall be the same in the parent

13549 and child processes. The inheritance of process characteristics not defined by

13550 IEEE Std 1003.1-200x is unspecified by IEEE Std 1003.1-200x.

13551 After *fork()*, both the parent and the child processes shall be capable of executing independently

13552 before either one terminates.

RETURN VALUE

13553

13554 Upon successful completion, *fork()* shall return 0 to the child process and shall return the

13555 process ID of the child process to the parent process. Both processes shall continue to execute

13556 from the *fork()* function. Otherwise, `-1` shall be returned to the parent process, no child process

13557 shall be created, and *errno* shall be set to indicate the error.

ERRORS

13558 The *fork()* function shall fail if:

13560 [EAGAIN] The system lacked the necessary resources to create another process, or the

13561 system-imposed limit on the total number of processes under execution

13562 system-wide or by a single user {CHILD_MAX} would be exceeded.

The *fork()* function may fail if:

[ENOMEM] Insufficient storage space is available.

EXAMPLES

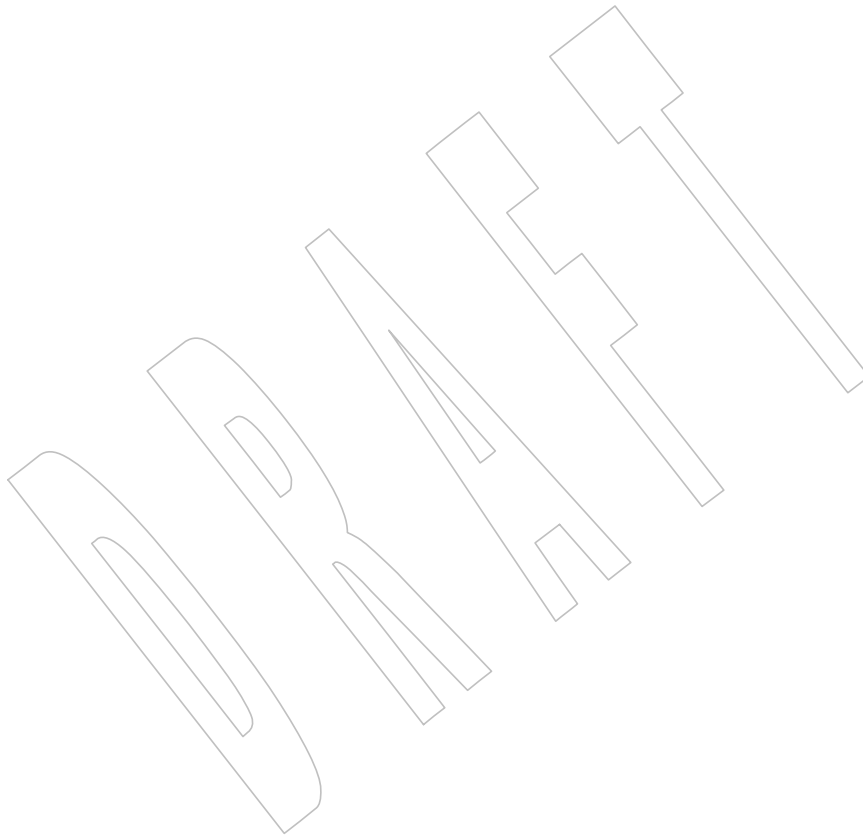
None.

APPLICATION USAGE

None.

RATIONALE

Many historical implementations have timing windows where a signal sent to a process group



13613 writer. Validation writers should be cognizant of this limitation.

13614 There are two reasons why POSIX programmers call *fork()*. One reason is to create a new thread
13615 of control within the same program (which was originally only possible in POSIX by creating a
13616 new process); the other is to create a new process running a different program. In the latter case,
13617 the call to *fork()* is soon followed by a call to one of the *exec* functions.

13618 The general problem with making *fork()* work in a multi-threaded world is what to do with all
13619 of the threads. There are two alternatives. One is to copy all of the threads into the new process.
13620 This causes the programmer or implementation to deal with threads that are suspended on
13621 system calls or that might be about to execute system calls that should not be executed in the
13622 new process. The other alternative is to copy only the thread that calls *fork()*. This creates the
13623 difficulty that the state of process-local resources is usually held in process memory. If a thread
13624 that is not calling *fork()* holds a resource, that resource is never released in the child process
13625 because the thread whose job it is to release the resource does not exist in the child process.

13626 When a programmer is writing a multi-threaded program, the first described use of *fork()*,
13627 creating new threads in the same program, is provided by the *pthread_create()* function. The
13628 *fork()* function is thus used only to run new programs, and the effects of calling functions that
13629 require certain resources between the call to *fork()* and the call to an *exec* function are undefined.

13630 The addition of the *forkall()* function to the standard was considered and rejected. The *forkall()*
13631 function lets all the threads in the parent be duplicated in the child. This essentially duplicates
13632 the state of the parent in the child. This allows threads in the child to continue processing and
13633 allows locks and the state to be preserved without explicit *pthread_atfork()* code. The calling
13634 process has to ensure that the threads processing state that is shared between the parent and
13635 child (that is, file descriptors or MAP_SHARED memory) behaves properly after *forkall()*. For
13636 example, if a thread is reading a file descriptor in the parent when *forkall()* is called, then two
13637 threads (one in the parent and one in the child) are reading the file descriptor after the *forkall()*.
13638 If this is not desired behavior, the parent process has to synchronize with such threads before
13639 calling *forkall()*.

13640 While the *fork()* function is async-signal-safe, there is no way for an implementation to
13641 determine whether the fork handlers established by *pthread_atfork()* are async-signal-safe. The
13642 fork handlers may attempt to execute portions of the implementation that are not async-signal-
13643 safe, such as those that are protected by mutexes, leading to a deadlock condition. It is therefore
13644 undefined for the fork handlers to execute functions that are not async-signal-safe when *fork()*
13645 is called from a signal handler.

13646 When *forkall()* is called, threads, other than the calling thread, that are in functions that can
13647 return with an [EINTR] error may have those functions return [EINTR] if the implementation
13648 cannot ensure that the function behaves correctly in the parent and child. In particular,
13649 *pthread_cond_wait()* and *pthread_cond_timedwait()* need to return in order to ensure that the
13650 condition has not changed. These functions can be awakened by a spurious condition wakeup
13651 rather than returning [EINTR].

13652 FUTURE DIRECTIONS

13653 None.

13654 SEE ALSO

13655 *alarm()*, *exec*, *fcntl()*, *posix_trace_attr_getinherited()*, *posix_trace_trid_eventid_open()*,
13656 *pthread_atfork()*, *semop()*, *signal()*, *times()*, the Base Definitions volume of IEEE Std 1003.1-200x,
13657 Section 4.10, Memory Synchronization, `<sys/types.h>`, `<unistd.h>`

13658 CHANGE HISTORY

13659 First released in Issue 1. Derived from Issue 1 of the SVID.

13660
13661
13662

13663
13664
13665

13666
13667
13668

13669

13670

13671

13672

13673
13674
13675

13676
13677
13678

13679
13680

13681

Issue 5

The DESCRIPTION is changed for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The effect of `fork()` on a pending alarm call in the child process is clarified.

The description of CPU-time clock semantics is added for alignment with IEEE Std 1003.1d-1999.

The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/17 is applied, adding text to the DESCRIPTION and RATIONALE relating to fork handlers registered by the `pthread_atfork()` function and async-signal safety.

Issue 7

Austin Group Interpretation 1003.1-2001 #080 is applied, clarifying the status of asynchronous input and asynchronous output operations and asynchronous control lists in the DESCRIPTION.

Functionality relating to the Asynchronous Input and Output, Memory Mapped Files, Timers, and Threads options is moved to the Base.

Functionality relating to message catalog descriptors is moved from the XSI option to the Base.

13682 **NAME**
 13683 `fpathconf, pathconf` — get configurable pathname variables

13684 **SYNOPSIS**
 13685 `#include <unistd.h>`
 13686 `long fpathconf(int filde, int name);`
 13687 `long pathconf(const char *path, int name);`

13688 **DESCRIPTION**
 13689 The `fpathconf()` and `pathconf()` functions shall determine the current value of a configurable limit
 13690 or option (*variable*) that is associated with a file or directory.

13691 For `pathconf()`, the `path` argument points to the pathname of a file or directory.

13692 For `fpathconf()`, the `filde` argument is an open file descriptor.

13693 The `name` argument represents the variable to be queried relative to that file or directory.
 13694 Implementations shall support all of the variables listed in the following table and may support
 13695 others. The variables in the following table come from `<limits.h>` or `<unistd.h>` and the
 13696 symbolic constants, defined in `<unistd.h>`, are the corresponding values used for `name`.

Variable	Value of <i>name</i>	Requirements
{FILESIZEBITS}	_PC_FILESIZEBITS	3,4
{LINK_MAX}	_PC_LINK_MAX	1
{MAX_CANON}	_PC_MAX_CANON	2
{MAX_INPUT}	_PC_MAX_INPUT	2
{NAME_MAX}	_PC_NAME_MAX	3,4
{PATH_MAX}	_PC_PATH_MAX	4,5
{PIPE_BUF}	_PC_PIPE_BUF	6
{POSIX2_SYMLINKS}	_PC_2_SYMLINKS	4
{POSIX_ALLOC_SIZE_MIN}	_PC_ALLOC_SIZE_MIN	10
{POSIX_REC_INCR_XFER_SIZE}	_PC_REC_INCR_XFER_SIZE	10
{POSIX_REC_MAX_XFER_SIZE}	_PC_REC_MAX_XFER_SIZE	10
{POSIX_REC_MIN_XFER_SIZE}	_PC_REC_MIN_XFER_SIZE	10
{POSIX_REC_XFER_ALIGN}	_PC_REC_XFER_ALIGN	10
{SYMLINK_MAX}	_PC_SYMLINK_MAX	4,9
_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED	7
_POSIX_NO_TRUNC	_PC_NO_TRUNC	3,4
_POSIX_VDISABLE	_PC_VDISABLE	2
_POSIX_ASYNC_IO	_PC_ASYNC_IO	8
_POSIX_PRIO_IO	_PC_PRIO_IO	8
_POSIX_SYNC_IO	_PC_SYNC_IO	8

13718 Requirements

- 13719 1. If `path` or `filde` refers to a directory, the value returned shall apply to the directory itself.
- 13720 2. If `path` or `filde` does not refer to a terminal file, it is unspecified whether an
 13721 implementation supports an association of the variable name with the specified file.
- 13722 3. If `path` or `filde` refers to a directory, the value returned shall apply to filenames within the
 13723 directory.
- 13724 4. If `path` or `filde` does not refer to a directory, it is unspecified whether an implementation
 13725 supports an association of the variable name with the specified file.

- 13726 5. If *path* or *filde*s refers to a directory, the value returned shall be the maximum length of a
13727 relative pathname when the specified directory is the working directory.
- 13728 6. If *path* refers to a FIFO, or *filde*s refers to a pipe or FIFO, the value returned shall apply to
13729 the referenced object. If *path* or *filde*s refers to a directory, the value returned shall apply to
13730 any FIFO that exists or can be created within the directory. If *path* or *filde*s refers to any
13731 other type of file, it is unspecified whether an implementation supports an association of
13732 the variable name with the specified file.
- 13733 7. If *path* or *filde*s refers to a directory, the value returned shall apply to any files, other than
13734 directories, that exist or can be created within the directory.
- 13735 8. If *path* or *filde*s refers to a directory, it is unspecified whether an implementation supports
13736 an association of the variable name with the specified file.
- 13737 9. If *path* or *filde*s refers to a directory, the value returned shall be the maximum length of the
13738 string that a symbolic link in that directory can contain.
- 13739 10. If *path* or *filde*s does not refer to a regular file, it is unspecified whether an
13740 implementation supports an association of the variable name with the specified file. If an
13741 implementation supports such an association for other than a regular file, the value
13742 returned is unspecified.

RETURN VALUE

13743 If *name* is an invalid value, both *pathconf*(*n*) and *fpathconf*(*n*) shall return -1 and set *errno* to
13744 indicate the error.
13745

13746 If the variable corresponding to *name* has no limit for the *path* or file descriptor, both *pathconf*(*n*)
13747 and *fpathconf*(*n*) shall return -1 without changing *errno*. If the implementation needs to use *path*
13748 to determine the value of *name* and the implementation does not support the association of *name*
13749 with the file specified by *path*, or if the process did not have appropriate privileges to query the
13750 file specified by *path*, or *path* does not exist, *pathconf*(*n*) shall return -1 and set *errno* to indicate the
13751 error.

13752 If the implementation needs to use *filde*s to determine the value of *name* and the implementation
13753 does not support the association of *name* with the file specified by *filde*s, or if *filde*s is an invalid
13754 file descriptor, *fpathconf*(*n*) shall return -1 and set *errno* to indicate the error.

13755 Otherwise, *pathconf*(*n*) or *fpathconf*(*n*) shall return the current variable value for the file or
13756 directory without changing *errno*. The value returned shall not be more restrictive than the
13757 corresponding value available to the application when it was compiled with the
13758 implementation's `<limits.h>` or `<unistd.h>`.

13759 If the variable corresponding to *name* is dependent on an unsupported option, the results are
13760 unspecified.

ERRORS

13761 The *pathconf*(*n*) function shall fail if:
13762

13763 [EINVAL] The value of *name* is not valid.

13764 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
13765 argument.

13766 The *pathconf*(*n*) function may fail if:

13767 [EACCES] Search permission is denied for a component of the path prefix.

13768 [EINVAL] The implementation does not support an association of the variable *name* with
13769 the specified file.

- 13770 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
13771 resolution of the *path* argument.
- 13772 [ENAMETOOLONG]
13773 The length of the *path* argument exceeds {PATH_MAX} or a pathname
13774 component is longer than {NAME_MAX}.
- 13775 [ENAMETOOLONG]
13776 As a result of encountering a symbolic link in resolution of the *path* argument,
13777 the length of the substituted pathname string exceeded {PATH_MAX}.
- 13778 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.
- 13779 [ENOTDIR] A component of the path prefix is not a directory.
- 13780 The *fpathconf()* function shall fail if:
- 13781 [EINVAL] The value of *name* is not valid.
- 13782 The *fpathconf()* function may fail if:
- 13783 [EBADF] The *fildev* argument is not a valid file descriptor.
- 13784 [EINVAL] The implementation does not support an association of the variable *name* with
13785 the specified file.

EXAMPLES

13786 None.
13787

APPLICATION USAGE

13788 Application writers should check whether an option, such as `_POSIX_ADVISORY_INFO`, is
13789 supported prior to obtaining and using values for related variables such as
13790 `{POSIX_ALLOC_SIZE_MIN}`.
13791

RATIONALE

13792 The *pathconf()* function was proposed immediately after the *sysconf()* function when it was
13793 realized that some configurable values may differ across file system, directory, or device
13794 boundaries.
13795

13796 For example, {NAME_MAX} frequently changes between System V and BSD-based file systems;
13797 System V uses a maximum of 14, BSD 255. On an implementation that provides both types of file
13798 systems, an application would be forced to limit all pathname components to 14 bytes, as this
13799 would be the value specified in `<limits.h>` on such a system.

13800 Therefore, various useful values can be queried on any pathname or file descriptor, assuming
13801 that the appropriate permissions are in place.

13802 The value returned for the variable {PATH_MAX} indicates the longest relative pathname that
13803 could be given if the specified directory is the current working directory of the process. A
13804 process may not always be able to generate a name that long and use it if a subdirectory in the
13805 pathname crosses into a more restrictive file system.

13806 The value returned for the variable `_POSIX_CHOWN_RESTRICTED` also applies to directories
13807 that do not have file systems mounted on them. The value may change when crossing a mount
13808 point, so applications that need to know should check for each directory. (An even easier check
13809 is to try the *chown()* function and look for an error in case it happens.)

13810 Unlike the values returned by *sysconf()*, the pathname-oriented variables are potentially more
13811 volatile and are not guaranteed to remain constant throughout the lifetime of the process. For
13812 example, in between two calls to *pathconf()*, the file system in question may have been
13813 unmounted and remounted with different characteristics.

13814 Also note that most of the errors are optional. If one of the variables always has the same value

13815 on an implementation, the implementation need not look at *path* or *fildev* to return that value and
 13816 is, therefore, not required to detect any of the errors except the meaning of [EINVAL] that
 13817 indicates that the value of *name* is not valid for that variable.

13818 If the value of any of the limits is unspecified (logically infinite), they will not be defined in
 13819 `<limits.h>` and the *pathconf()* and *fpathconf()* functions return `-1` without changing *errno*. This
 13820 can be distinguished from the case of giving an unrecognized *name* argument because *errno* is set
 13821 to [EINVAL] in this case.

13822 Since `-1` is a valid return value for the *pathconf()* and *fpathconf()* functions, applications should
 13823 set *errno* to zero before calling them and check *errno* only if the return value is `-1`.

13824 For the case of {SYMLINK_MAX}, since both *pathconf()* and *open()* follow symbolic links, there
 13825 is no way that *path* or *fildev* could refer to a symbolic link.

13826 It was the intention of IEEE Std 1003.1d-1999 that the following variables:

```
13827 {POSIX_ALLOC_SIZE_MIN}
13828 {POSIX_REC_INCR_XFER_SIZE}
13829 {POSIX_REC_MAX_XFER_SIZE}
13830 {POSIX_REC_MIN_XFER_SIZE}
13831 {POSIX_REC_XFER_ALIGN}
```

13832 only applied to regular files, but Note 10 also permits implementation of the advisory semantics
 13833 on other file types unique to an implementation (for example, a character special device).

13834 FUTURE DIRECTIONS

13835 None.

13836 SEE ALSO

13837 *chown()*, *confstr()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<limits.h>`,
 13838 `<unistd.h>`, the Shell and Utilities volume of IEEE Std 1003.1-200x, *getconf*

13839 CHANGE HISTORY

13840 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

13841 Issue 5

13842 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

13843 Large File Summit extensions are added.

13844 Issue 6

13845 The following new requirements on POSIX implementations derive from alignment with the
 13846 Single UNIX Specification:

- 13847 • The DESCRIPTION is updated to include {FILESIZEBITS}.
- 13848 • The [ELOOP] mandatory error condition is added.
- 13849 • A second [ENAMETOOLONG] is added as an optional error condition.

13850 The following changes were made to align with the IEEE P1003.1a draft standard:

- 13851 • The `_PC_SYMLINK_MAX` entry is added to the table in the DESCRIPTION.

13852 The following *pathconf()* variables and their associated names are added for alignment with
 13853 IEEE Std 1003.1d-1999:

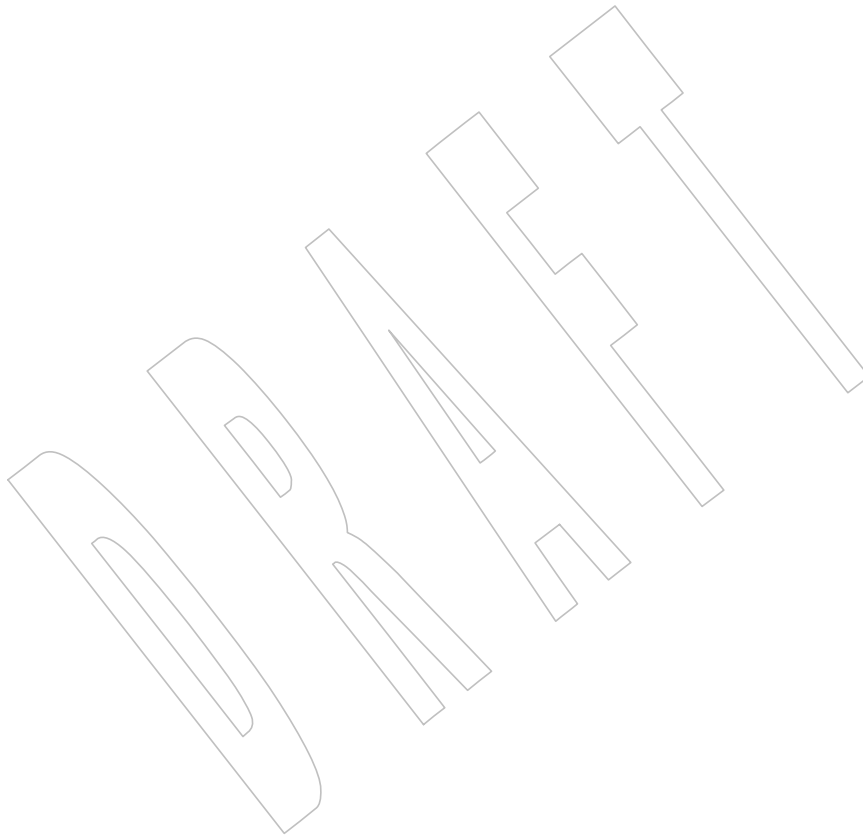
```
13854 {POSIX_ALLOC_SIZE_MIN}
13855 {POSIX_REC_INCR_XFER_SIZE}
13856 {POSIX_REC_MAX_XFER_SIZE}
13857 {POSIX_REC_MIN_XFER_SIZE}
13858 {POSIX_REC_XFER_ALIGN}
```

13859 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/18 is applied, changing the fourth
13860 paragraph of the DESCRIPTION and removing shading and margin markers from the table.
13861 This change is needed since implementations are required to support all of these symbols.

13862 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/34 is applied, adding the table entry for
13863 POSIX2_SYMLINKS in the DESCRIPTION.

13864 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/35 is applied, updating the
13865 DESCRIPTION and RATIONALE sections to clarify behavior for the
13866 {POSIX_ALLOC_SIZE_MIN}, {POSIX_REC_INCR_XFER_SIZE},
13867 {POSIX_REC_MAX_XFER_SIZE}, {POSIX_REC_MIN_XFER_SIZE}, and
13868 {POSIX_REC_XFER_ALIGN} variables.

13869 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/36 is applied, updating the RETURN
13870 VALUE and APPLICATION USAGE sections to state that the results are unspecified if a variable
13871 is dependent on an unsupported option, and advising application writers to check for supported
13872 options prior to obtaining and using such values.



13873 **NAME**
 13874 fpclassify — classify real floating type

13875 **SYNOPSIS**
 13876 #include <math.h>
 13877 int fpclassify(real-floating x);

13878 **DESCRIPTION**
 13879 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 13880 conflict between the requirements described here and the ISO C standard is unintentional. This
 13881 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

13882 The *fpclassify()* macro shall classify its argument value as NaN, infinite, normal, subnormal,
 13883 zero, or into another implementation-defined category. First, an argument represented in a
 13884 format wider than its semantic type is converted to its semantic type. Then classification is based
 13885 on the type of the argument.

13886 **RETURN VALUE**
 13887 The *fpclassify()* macro shall return the value of the number classification macro appropriate to
 13888 the value of its argument.

13889 **ERRORS**
 13890 No errors are defined.

13891 **EXAMPLES**
 13892 None.

13893 **APPLICATION USAGE**
 13894 None.

13895 **RATIONALE**
 13896 None.

13897 **FUTURE DIRECTIONS**
 13898 None.

13899 **SEE ALSO**
 13900 *isfinite()*, *isinf()*, *isnan()*, *isnormal()*, *signbit()*, the Base Definitions volume of
 13901 IEEE Std 1003.1-200x, <math.h>

13902 **CHANGE HISTORY**
 13903 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

13904 **NAME**13905 `dprintf, fprintf, printf, snprintf, sprintf` — print formatted output13906 **SYNOPSIS**13907 `#include <stdio.h>`

```

13908 CX int dprintf(int fildes, const char *format, ...);
13909 int fprintf(FILE *restrict stream, const char *restrict format, ...);
13910 int printf(const char *restrict format, ...);
13911 int snprintf(char *restrict s, size_t n,
13912 const char *restrict format, ...);
13913 int sprintf(char *restrict s, const char *restrict format, ...);

```

13914 **DESCRIPTION**

13915 CX Excluding `dprintf()`: The functionality described on this reference page is aligned with the ISO C
 13916 standard. Any conflict between the requirements described here and the ISO C standard is
 13917 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

13918 The `fprintf()` function shall place output on the named output *stream*. The `printf()` function shall
 13919 place output on the standard output stream *stdout*. The `sprintf()` function shall place output
 13920 followed by the null byte, `'\0'`, in consecutive bytes starting at **s*; it is the user's responsibility
 13921 to ensure that enough space is available.

13922 CX The `dprintf()` function shall be equivalent to the `fprintf()` function, except that `dprintf()` shall
 13923 write output to the file associated with the file descriptor specified by the *fildes* argument rather
 13924 than place output on a stream.

13925 The `snprintf()` function shall be equivalent to `sprintf()`, with the addition of the *n* argument
 13926 which states the size of the buffer referred to by *s*. If *n* is zero, nothing shall be written and *s*
 13927 may be a null pointer. Otherwise, output bytes beyond the *n*-1st shall be discarded instead of
 13928 being written to the array, and a null byte is written at the end of the bytes actually written into
 13929 the array.

13930 If copying takes place between objects that overlap as a result of a call to `sprintf()` or `snprintf()`,
 13931 the results are undefined.

13932 Each of these functions converts, formats, and prints its arguments under control of the *format*.
 13933 The *format* is a character string, beginning and ending in its initial shift state, if any. The *format* is
 13934 composed of zero or more directives: *ordinary characters*, which are simply copied to the output
 13935 stream, and *conversion specifications*, each of which shall result in the fetching of zero or more
 13936 arguments. The results are undefined if there are insufficient arguments for the *format*. If the
 13937 *format* is exhausted while arguments remain, the excess arguments shall be evaluated but are
 13938 otherwise ignored.

13939 CX Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
 13940 to the next unused argument. In this case, the conversion specifier character `%` (see below) is
 13941 replaced by the sequence `"%n$"`, where *n* is a decimal integer in the range `[1,{NL_ARGMAX}]`,
 13942 giving the position of the argument in the argument list. This feature provides for the definition
 13943 of format strings that select arguments in an order appropriate to specific languages (see the
 13944 EXAMPLES section).

13945 The *format* can contain either numbered argument conversion specifications (that is, `"%n$"` and
 13946 `"*m$"`), or unnumbered argument conversion specifications (that is, `%` and `*`), but not both. The
 13947 only exception to this is that `%%` can be mixed with the `"%n$"` form. The results of mixing
 13948 numbered and unnumbered argument specifications in a *format* string are undefined. When
 13949 numbered argument specifications are used, specifying the *N*th argument requires that all the
 13950 leading arguments, from the first to the $(N-1)$ th, are specified in the format string.

- 13951 In format strings containing the "%n\$" form of conversion specification, numbered arguments
13952 in the argument list can be referenced from the format string as many times as required.
- 13953 In format strings containing the % form of conversion specification, each conversion specification
13954 uses the first unused argument in the argument list.
- 13955 CX All forms of the *fprintf()* functions allow for the insertion of a language-dependent radix
13956 character in the output string. The radix character is defined in the process' locale (category
13957 *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the
13958 radix character shall default to a period ('.').
- 13959 CX Each conversion specification is introduced by the '%' character or by the character sequence
13960 "%n\$", after which the following appear in sequence:
- 13961 • Zero or more *flags* (in any order), which modify the meaning of the conversion
13962 specification.
 - 13963 • An optional minimum *field width*. If the converted value has fewer bytes than the field
13964 width, it shall be padded with spaces by default on the left; it shall be padded on the right
13965 if the left-adjustment flag ('-'), described below, is given to the field width. The field
13966 width takes the form of an asterisk ('*'), described below, or a decimal integer.
 - 13967 • An optional *precision* that gives the minimum number of digits to appear for the d, i, o, u,
13968 x, and X conversion specifiers; the number of digits to appear after the radix character for
13969 the a, A, e, E, f, and F conversion specifiers; the maximum number of significant digits for
13970 the g and G conversion specifiers; or the maximum number of bytes to be printed from a
13971 string in the s and S conversion specifiers. The precision takes the form of a period ('.')
13972 followed either by an asterisk ('*'), described below, or an optional decimal digit string,
13973 where a null digit string is treated as zero. If a precision appears with any other conversion
13974 specifier, the behavior is undefined.
 - 13975 • An optional length modifier that specifies the size of the argument.
 - 13976 • A *conversion specifier* character that indicates the type of conversion to be applied.
- 13977 A field width, or precision, or both, may be indicated by an asterisk ('*'). In this case an
13978 argument of type *int* supplies the field width or precision. Applications shall ensure that
13979 arguments specifying field width, or precision, or both appear in that order before the argument,
13980 if any, to be converted. A negative field width is taken as a '-' flag followed by a positive field
13981 width. A negative precision is taken as if the precision were omitted. In *format* strings
13982 containing the "%n\$" form of a conversion specification, a field width or precision may be
13983 indicated by the sequence "*m\$", where *m* is a decimal integer in the range [1,{NL_ARGMAX}]
13984 giving the position in the argument list (after the *format* argument) of an integer argument
13985 containing the field width or precision, for example:
- ```
13986 printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```
- 13987 The flag characters and their meanings are:
- 13988 CX ' The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %F, %g, or %G)  
13989 shall be formatted with thousands' grouping characters. For other conversions the  
13990 behavior is undefined. The non-monetary grouping character is used.
- 13991 - The result of the conversion shall be left-justified within the field. The conversion is  
13992 right-justified if this flag is not specified.
- 13993 + The result of a signed conversion shall always begin with a sign ('+' or '-'). The  
13994 conversion shall begin with a sign only when a negative value is converted if this flag is  
13995 not specified.



|       |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-------|----|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 13996 |    | <space>      | If the first character of a signed conversion is not a sign or if a signed conversion results in no characters, a <space> shall be prefixed to the result. This means that if the <space> and '+' flags both appear, the <space> flag shall be ignored.                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 13997 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 13998 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 13999 |    | #            | Specifies that the value is to be converted to an alternative form. For o conversion, it increases the precision (if necessary) to force the first digit of the result to be zero. For x or X conversion specifiers, a non-zero result shall have 0x (or 0X) prefixed to it. For a, A, e, E, f, F, g, and G conversion specifiers, the result shall always contain a radix character, even if no digits follow the radix character. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For g and G conversion specifiers, trailing zeros shall <i>not</i> be removed from the result as they normally are. For other conversion specifiers, the behavior is undefined. |
| 14000 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14001 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14002 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14003 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14004 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14005 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14006 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14007 |    | 0            | For d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversion specifiers, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the '0' and '-' flags both appear, the '0' flag is ignored. For d, i, o, u, x, and X conversion specifiers, if a precision is specified, the '0' flag shall be ignored. If the '0' and '' flags both appear, the grouping characters are inserted before zero padding. For other conversions, the behavior is undefined.                                                                                                                                                                                                 |
| 14008 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14009 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14010 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14011 | CX |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14012 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14013 |    |              | The length modifiers and their meanings are:                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| 14014 |    | hh           | Specifies that a following d, i, o, u, x, or X conversion specifier applies to a <b>signed char</b> or <b>unsigned char</b> argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to <b>signed char</b> or <b>unsigned char</b> before printing); or that a following n conversion specifier applies to a pointer to a <b>signed char</b> argument.                                                                                                                                                                                                                                                                                                              |
| 14015 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14016 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14017 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14018 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14019 |    | h            | Specifies that a following d, i, o, u, x, or X conversion specifier applies to a <b>short</b> or <b>unsigned short</b> argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to <b>short</b> or <b>unsigned short</b> before printing); or that a following n conversion specifier applies to a pointer to a <b>short</b> argument.                                                                                                                                                                                                                                                                                                                              |
| 14020 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14021 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14022 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14023 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14024 |    | l (ell)      | Specifies that a following d, i, o, u, x, or X conversion specifier applies to a <b>long</b> or <b>unsigned long</b> argument; that a following n conversion specifier applies to a pointer to a <b>long</b> argument; that a following c conversion specifier applies to a <b>wint_t</b> argument; that a following s conversion specifier applies to a pointer to a <b>wchar_t</b> argument; or has no effect on a following a, A, e, E, f, F, g, or G conversion specifier.                                                                                                                                                                                                                                                  |
| 14025 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14026 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14027 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14028 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14029 |    | ll (ell-ell) | Specifies that a following d, i, o, u, x, or X conversion specifier applies to a <b>long long</b> or <b>unsigned long long</b> argument; or that a following n conversion specifier applies to a pointer to a <b>long long</b> argument.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| 14030 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14031 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14032 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14033 |    | j            | Specifies that a following d, i, o, u, x, or X conversion specifier applies to an <b>intmax_t</b> or <b>uintmax_t</b> argument; or that a following n conversion specifier applies to a pointer to an <b>intmax_t</b> argument.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14034 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14035 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14036 |    | z            | Specifies that a following d, i, o, u, x, or X conversion specifier applies to a <b>size_t</b> or the corresponding signed integer type argument; or that a following n conversion specifier applies to a pointer to a signed integer type corresponding to a <b>size_t</b> argument.                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 14037 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14038 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14039 |    | t            | Specifies that a following d, i, o, u, x, or X conversion specifier applies to a <b>ptrdiff_t</b> or the corresponding <b>unsigned</b> type argument; or that a following n conversion specifier applies to a pointer to a <b>ptrdiff_t</b> argument.                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 14040 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14041 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 14042 |    | L            | Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a <b>long double</b> argument.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| 14043 |    |              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |

- 14044 If a length modifier appears with any conversion specifier other than as specified above, the  
14045 behavior is undefined.
- 14046 The conversion specifiers and their meanings are:
- 14047 **d, i** The **int** argument shall be converted to a signed decimal in the style "`[-]dddd`". The  
14048 precision specifies the minimum number of digits to appear; if the value being  
14049 converted can be represented in fewer digits, it shall be expanded with leading zeros.  
14050 The default precision is 1. The result of converting zero with an explicit precision of  
14051 zero shall be no characters.
- 14052 **o** The **unsigned** argument shall be converted to unsigned octal format in the style  
14053 "`dddd`". The precision specifies the minimum number of digits to appear; if the value  
14054 being converted can be represented in fewer digits, it shall be expanded with leading  
14055 zeros. The default precision is 1. The result of converting zero with an explicit precision  
14056 of zero shall be no characters.
- 14057 **u** The **unsigned** argument shall be converted to unsigned decimal format in the style  
14058 "`dddd`". The precision specifies the minimum number of digits to appear; if the value  
14059 being converted can be represented in fewer digits, it shall be expanded with leading  
14060 zeros. The default precision is 1. The result of converting zero with an explicit precision  
14061 of zero shall be no characters.
- 14062 **x** The **unsigned** argument shall be converted to unsigned hexadecimal format in the style  
14063 "`dddd`"; the letters "abcdef" are used. The precision specifies the minimum number  
14064 of digits to appear; if the value being converted can be represented in fewer digits, it  
14065 shall be expanded with leading zeros. The default precision is 1. The result of  
14066 converting zero with an explicit precision of zero shall be no characters.
- 14067 **X** Equivalent to the x conversion specifier, except that letters "ABCDEF" are used instead  
14068 of "abcdef".
- 14069 **f, F** The **double** argument shall be converted to decimal notation in the style  
14070 "`[-]ddd.d1d2d3`", where the number of digits after the radix character is equal to the  
14071 precision specification. If the precision is missing, it shall be taken as 6; if the precision  
14072 is explicitly zero and no '#' flag is present, no radix character shall appear. If a radix  
14073 character appears, at least one digit appears before it. The low-order digit shall be  
14074 rounded in an implementation-defined manner.
- 14075 A **double** argument representing an infinity shall be converted in one of the styles  
14076 "`[-]inf`" or "`[-]infinity`"; which style is implementation-defined. A **double**  
14077 argument representing a NaN shall be converted in one of the styles "`[-]nan(n-char-sequence)`" or "`[-]nan`"; which style, and the meaning of any *n-char-sequence*,  
14078 is implementation-defined. The **F** conversion specifier produces "INF", "INFINITY",  
14079 or "NAN" instead of "inf", "infinity", or "nan", respectively.
- 14081 **e, E** The **double** argument shall be converted in the style "`[-]d.ddde±dd`", where there is  
14082 one digit before the radix character (which is non-zero if the argument is non-zero) and  
14083 the number of digits after it is equal to the precision; if the precision is missing, it shall  
14084 be taken as 6; if the precision is zero and no '#' flag is present, no radix character shall  
14085 appear. The low-order digit shall be rounded in an implementation-defined manner.  
14086 The **E** conversion specifier shall produce a number with 'E' instead of 'e'  
14087 introducing the exponent. The exponent shall always contain at least two digits. If the  
14088 value is zero, the exponent shall be zero.
- 14089 A **double** argument representing an infinity or NaN shall be converted in the style of  
14090 an **f** or **F** conversion specifier.



14140 to equal the character sequence length given by the precision, the function would need  
 14141 to access a wide character one past the end of the array. In no case shall a partial  
 14142 character be written.

14143 p The argument shall be a pointer to **void**. The value of the pointer is converted to a  
 14144 sequence of printable characters, in an implementation-defined manner.

14145 n The argument shall be a pointer to an integer into which is written the number of bytes  
 14146 written to the output so far by this call to one of the *fprintf()* functions. No argument is  
 14147 converted.

14148 XSI C Equivalent to `lc`.

14149 XSI S Equivalent to `ls`.

14150 % Print a ' % ' character; no argument is converted. The complete conversion specification  
 14151 shall be `%%`.

14152 If a conversion specification does not match one of the above forms, the behavior is undefined. If  
 14153 any argument is not the correct type for the corresponding conversion specification, the  
 14154 behavior is undefined.

14155 In no case shall a nonexistent or small field width cause truncation of a field; if the result of a  
 14156 conversion is wider than the field width, the field shall be expanded to contain the conversion  
 14157 result. Characters generated by *fprintf()* and *printf()* are printed as if *fputc()* had been called.

14158 For the `a` and `A` conversion specifiers, if `FLT_RADIX` is a power of 2, the value shall be correctly  
 14159 rounded to a hexadecimal floating number with the given precision.

14160 For `a` and `A` conversions, if `FLT_RADIX` is not a power of 2 and the result is not exactly  
 14161 representable in the given precision, the result should be one of the two adjacent numbers in  
 14162 hexadecimal floating style with the given precision, with the extra stipulation that the error  
 14163 should have a correct sign for the current rounding direction.

14164 For the `e`, `E`, `f`, `F`, `g`, and `G` conversion specifiers, if the number of significant decimal digits is at  
 14165 most `DECIMAL_DIG`, then the result should be correctly rounded. If the number of significant  
 14166 decimal digits is more than `DECIMAL_DIG` but the source value is exactly representable with  
 14167 `DECIMAL_DIG` digits, then the result should be an exact representation with trailing zeros.  
 14168 Otherwise, the source value is bounded by two adjacent decimal strings  $L < U$ , both having  
 14169 `DECIMAL_DIG` significant digits; the value of the resultant decimal string  $D$  should satisfy  $L \leq$   
 14170  $D \leq U$ , with the extra stipulation that the error should have a correct sign for the current  
 14171 rounding direction.

14172 CX The `st_ctime` and `st_mtime` fields of the file shall be marked for update between the call to a  
 14173 successful execution of *fprintf()* or *printf()* and the next successful completion of a call to *fflush()*  
 14174 or *fclose()* on the same stream or a call to *exit()* or *abort()*.

## 14175 RETURN VALUE

14176 CX Upon successful completion, the *dprintf()*, *fprintf()*, and *printf()* functions shall return the  
 14177 number of bytes transmitted.

14178 Upon successful completion, the *sprintf()* function shall return the number of bytes written to *s*,  
 14179 excluding the terminating null byte.

14180 Upon successful completion, the *snprintf()* function shall return the number of bytes that would  
 14181 be written to *s* had *n* been sufficiently large excluding the terminating null byte.

14182 If an output error was encountered, these functions shall return a negative value.

14183 If the value of *n* is zero on a call to *snprintf()*, nothing shall be written, the number of bytes that  
 14184 would have been written had *n* been sufficiently large excluding the terminating null shall be  
 14185 returned, and *s* may be a null pointer.

**14186 ERRORS**

14187 CX For the conditions under which `dprintf()`, `fprintf()`, and `printf()` fail and may fail, refer to `fputc()`  
14188 or `fputcwc()`.

14189 In addition, all forms of `fprintf()` may fail if:

14190 CX **[EILSEQ]** A wide-character code that does not correspond to a valid character has been  
14191 detected.

14192 CX **[EINVAL]** There are insufficient arguments.

14193 The `dprintf()` function may fail if:

14194 **[EBADF]** The `fildev` argument is not a valid file descriptor.

14195 CX The `dprintf()`, `fprintf()`, and `printf()` functions may fail if:

14196 CX **[ENOMEM]** Insufficient storage space is available.

14197 The `snprintf()` function shall fail if:

14198 CX **[EOVERFLOW]** The value of `n` is greater than `{INT_MAX}` or the number of bytes needed to  
14199 hold the output excluding the terminating null is greater than `{INT_MAX}`.

**14200 EXAMPLES****14201 Printing Language-Independent Date and Time**

14202 The following statement can be used to print date and time using a language-independent  
14203 format:

14204 `printf(format, weekday, month, day, hour, min);`

14205 For American usage, `format` could be a pointer to the following string:

14206 `"%s, %s %d, %d:%.2d\n"`

14207 This example would produce the following message:

14208 `Sunday, July 3, 10:02`

14209 For German usage, `format` could be a pointer to the following string:

14210 `"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"`

14211 This definition of `format` would produce the following message:

14212 `Sonntag, 3. Juli, 10:02`

**14213 Printing File Information**

14214 The following example prints information about the type, permissions, and number of links of a  
14215 specific file in a directory.

14216 The first two calls to `printf()` use data decoded from a previous `stat()` call. The user-defined  
14217 `strperm()` function shall return a string similar to the one at the beginning of the output for the  
14218 following command:

14219 `ls -l`

14220 The next call to `printf()` outputs the owner's name if it is found using `getpwuid()`; the `getpwuid()`  
14221 function shall return a `passwd` structure from which the name of the user is extracted. If the user  
14222 name is not found, the program instead prints out the numeric value of the user ID.

14223 The next call prints out the group name if it is found using `getgrgid()`; `getgrgid()` is very similar  
14224 to `getpwuid()` except that it shall return group information based on the group number. Once

14225 again, if the group is not found, the program prints the numeric value of the group for the entry.

14226 The final call to *printf()* prints the size of the file.

```

14227 #include <stdio.h>
14228 #include <sys/types.h>
14229 #include <pwd.h>
14230 #include <grp.h>

14231 char *strperm (mode_t);
14232 ...
14233 struct stat statbuf;
14234 struct passwd *pwd;
14235 struct group *grp;
14236 ...
14237 printf("%10.10s", strperm (statbuf.st_mode));
14238 printf("%4d", statbuf.st_nlink);

14239 if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
14240 printf(" %-8.8s", pwd->pw_name);
14241 else
14242 printf(" %-8ld", (long) statbuf.st_uid);

14243 if ((grp = getgrgid(statbuf.st_gid)) != NULL)
14244 printf(" %-8.8s", grp->gr_name);
14245 else
14246 printf(" %-8ld", (long) statbuf.st_gid);

14247 printf("%9jd", (intmax_t) statbuf.st_size);
14248 ...

```

#### 14249 **Printing a Localized Date String**

14250 The following example gets a localized date string. The *nl\_langinfo()* function shall return the  
 14251 localized date string, which specifies the order and layout of the date. The *strftime()* function  
 14252 takes this information and, using the **tm** structure for values, places the date and time  
 14253 information into *datestring*. The *printf()* function then outputs *datestring* and the name of the  
 14254 entry.

```

14255 #include <stdio.h>
14256 #include <time.h>
14257 #include <langinfo.h>
14258 ...
14259 struct dirent *dp;
14260 struct tm *tm;
14261 char datestring[256];
14262 ...
14263 strftime(datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm);

14264 printf(" %s %s\n", datestring, dp->d_name);
14265 ...

```

14266

**Printing Error Information**

14267

The following example uses *fprintf()* to write error information to standard error.

14268

14269

14270

14271

In the first group of calls, the program tries to open the password lock file named **LOCKFILE**. If the file already exists, this is an error, as indicated by the **O\_EXCL** flag on the *open()* function. If the call fails, the program assumes that someone else is updating the password file, and the program exits.

14272

14273

The next group of calls saves a new password file as the current password file by creating a link between **LOCKFILE** and the new password file **PASSWDFILE**.

14274

14275

14276

14277

14278

14279

14280

14281

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"
...
int pfd;
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
 S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
 fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
 exit(1);
}
...
if (link(LOCKFILE, PASSWDFILE) == -1) {
 fprintf(stderr, "Link error: %s\n", strerror(errno));
 exit(1);
}
...

```

14299

**Printing Usage Information**

14300

14301

The following example checks to make sure the program has the necessary arguments, and uses *fprintf()* to print usage information if the expected number of arguments is not present.

14302

14303

14304

14305

14306

14307

14308

14309

14310

```
#include <stdio.h>
#include <stdlib.h>
...
char *Options = "hdbt1";
...
if (argc < 2) {
 fprintf(stderr, "Usage: %s -%s <file\n", argv[0], Options); exit(1);
}
...

```



14311

**Formatting a Decimal String**

14312

The following example prints a key and data pair on *stdout*. Note use of the '\*' (asterisk) in the format string; this ensures the correct number of decimal places for the element based on the number of elements requested.

14313

14314

14315

```
#include <stdio.h>
```

14316

```
...
```

14317

```
long i;
```

14318

```
char *keystr;
```

14319

```
int elementlen, len;
```

14320

```
...
```

14321

```
while (len < elementlen) {
```

14322

```
...
```

14323

```
 printf("%s Element%0*ld\n", keystr, elementlen, i);
```

14324

```
...
```

14325

```
}
```

14326

**Creating a Filename**

14327

The following example creates a filename using information from a previous *getpwnam()* function that returned the HOME directory of the user.

14328

14329

```
#include <stdio.h>
```

14330

```
#include <sys/types.h>
```

14331

```
#include <unistd.h>
```

14332

```
...
```

14333

```
char filename[PATH_MAX+1];
```

14334

```
struct passwd *pw;
```

14335

```
...
```

14336

```
sprintf(filename, "%s/%d.out", pw->pw_dir, getpid());
```

14337

```
...
```

14338

**Reporting an Event**

14339

The following example loops until an event has timed out. The *pause()* function waits forever unless it receives a signal. The *fprintf()* statement should never occur due to the possible return values of *pause()*.

14340

14341

14342

```
#include <stdio.h>
```

14343

```
#include <unistd.h>
```

14344

```
#include <string.h>
```

14345

```
#include <errno.h>
```

14346

```
...
```

14347

```
while (!event_complete) {
```

14348

```
...
```

14349

```
 if (pause() != -1 || errno != EINTR)
```

14350

```
 fprintf(stderr, "pause: unknown error: %s\n", strerror(errno));
```

14351

```
}
```

14352

```
...
```



14353

**Printing Monetary Information**

14354

The following example uses *strfmon()* to convert a number and store it as a formatted monetary string named *convbuf*. If the first number is printed, the program prints the format and the description; otherwise, it just prints the number.

14355

14356

14357

```
#include <monetary.h>
```

14358

```
#include <stdio.h>
```

14359

```
...
```

14360

```
struct tblfmt {
```

14361

```
 char *format;
```

14362

```
 char *description;
```

14363

```
};
```

14364

```
struct tblfmt table[] = {
```

14365

```
 { "%n", "default formatting" },
```

14366

```
 { "%11n", "right align within an 11 character field" },
```

14367

```
 { "%#5n", "aligned columns for values up to 99999" },
```

14368

```
 { "%=*#5n", "specify a fill character" },
```

14369

```
 { "%=0#5n", "fill characters do not use grouping" },
```

14370

```
 { "%^#5n", "disable the grouping separator" },
```

14371

```
 { "%^#5.0n", "round off to whole units" },
```

14372

```
 { "%^#5.4n", "increase the precision" },
```

14373

```
 { "%(#5n", "use an alternative pos/neg style" },
```

14374

```
 { "%!(#5n", "disable the currency symbol" },
```

14375

```
};
```

14376

```
...
```

14377

```
float input[3];
```

14378

```
int i, j;
```

14379

```
char convbuf[100];
```

14380

```
...
```

14381

```
strfmon(convbuf, sizeof(convbuf), table[i].format, input[j]);
```

14382

```
if (j == 0) {
```

14383

```
 printf("%s%s\n", table[i].format,
```

14384

```
 convbuf, table[i].description);
```

14385

```
}
```

14386

```
else {
```

14387

```
 printf("%s\n", convbuf);
```

14388

```
}
```

14388

```
...
```

14389

14390

**Printing Wide Characters**

14391

The following example prints a series of wide characters. Suppose that "L'@'" expands to three bytes:

14392

14393

```
wchar_t wz [3] = L"@@"; // Zero-terminated
```

14394

```
wchar_t wn [3] = L"@@"; // Unterminated
```

14395

```
fprintf (stdout, "%ls", wz); // Outputs 6 bytes
```

14396

```
fprintf (stdout, "%ls", wn); // Undefined because wn has no terminator
```

14397

```
fprintf (stdout, "%4ls", wz); // Outputs 3 bytes
```

14398

```
fprintf (stdout, "%4ls", wn); // Outputs 3 bytes; no terminator needed
```

14399

```
fprintf (stdout, "%9ls", wz); // Outputs 6 bytes
```

14400

```
fprintf (stdout, "%9ls", wn); // Outputs 9 bytes; no terminator needed
```

14401

```
fprintf (stdout, "%10ls", wz); // Outputs 6 bytes
```

14402        `fprintf (stdout, "%10ls", wn); // Undefined because wn has no terminator`

14403        In the last line of the example, after processing three characters, nine bytes have been output.  
 14404        The fourth character must then be examined to determine whether it converts to one byte or  
 14405        more. If it converts to more than one byte, the output is only nine bytes. Since there is no fourth  
 14406        character in the array, the behavior is undefined.

#### 14407        APPLICATION USAGE

14408        If the application calling `fprintf()` has any objects of type `wint_t` or `wchar_t`, it must also include  
 14409        the `<wchar.h>` header to have these objects defined.

#### 14410        RATIONALE

14411        None.

#### 14412        FUTURE DIRECTIONS

14413        None.

#### 14414        SEE ALSO

14415        `fputc()`, `fscanf()`, `setlocale()`, `strfmon()`, `wcrtomb()`, the Base Definitions volume of  
 14416        IEEE Std 1003.1-200x, Chapter 7, Locale, `<stdio.h>`, `<wchar.h>`

#### 14417        CHANGE HISTORY

14418        First released in Issue 1. Derived from Issue 1 of the SVID.

##### 14419        Issue 5

14420        Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the `l` (ell) qualifier can  
 14421        now be used with `c` and `s` conversion specifiers.

14422        The `snprintf()` function is new in Issue 5.

##### 14423        Issue 6

14424        Extensions beyond the ISO C standard are marked.

14425        The normative text is updated to avoid use of the term “must” for application requirements.

14426        The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 14427        • The prototypes for `fprintf()`, `printf()`, `snprintf()`, and `sprintf()` are updated, and the XSI  
 14428        shading is removed from `snprintf()`.
- 14429        • The description of `snprintf()` is aligned with the ISO C standard. Note that this supersedes  
 14430        the `snprintf()` description in The Open Group Base Resolution bwg98-006, which changed  
 14431        the behavior from Issue 5.
- 14432        • The DESCRIPTION is updated.

14433        The DESCRIPTION is updated to use the terms “conversion specifier” and “conversion  
 14434        specification” consistently.

14435        ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

14436        An example of printing wide characters is added.

##### 14437        Issue 7

14438        ISO/IEC 9899:1999 standard, Technical Corrigendum 2 #68 (SD5-XSH-ERN-70) is applied,  
 14439        revising the description of `g` and `G`.

14440        The `dprintf()` function is added from The Open Group Technical Standard, 2006, Extended API  
 14441        Set Part 1.

14442        Functionality relating to the `%n$` form of conversion specification and the `'` (apostrophe) flag  
 14443        is moved from the XSI option to the Base.

14444 **NAME**14445 `fputc` — put a byte on a stream14446 **SYNOPSIS**14447 `#include <stdio.h>`14448 `int fputc(int c, FILE *stream);`14449 **DESCRIPTION**14450 CX The functionality described on this reference page is aligned with the ISO C standard. Any  
14451 conflict between the requirements described here and the ISO C standard is unintentional. This  
14452 volume of IEEE Std 1003.1-200x defers to the ISO C standard.14453 The `fputc()` function shall write the byte specified by `c` (converted to an **unsigned char**) to the  
14454 output stream pointed to by `stream`, at the position indicated by the associated file-position  
14455 indicator for the stream (if defined), and shall advance the indicator appropriately. If the file  
14456 cannot support positioning requests, or if the stream was opened with append mode, the byte  
14457 shall be appended to the output stream.14458 CX The `st_ctime` and `st_mtime` fields of the file shall be marked for update between the successful  
14459 execution of `fputc()` and the next successful completion of a call to `fflush()` or `fclose()` on the same  
14460 stream or a call to `exit()` or `abort()`.14461 **RETURN VALUE**14462 Upon successful completion, `fputc()` shall return the value it has written. Otherwise, it shall  
14463 CX return EOF, the error indicator for the stream shall be set, and `errno` shall be set to indicate the  
14464 error.14465 **ERRORS**14466 The `fputc()` function shall fail if either the `stream` is unbuffered or the `stream`'s buffer needs to be  
14467 flushed, and:14468 CX **[EAGAIN]** The `O_NONBLOCK` flag is set for the file descriptor underlying `stream` and  
14469 the thread would be delayed in the write operation.14470 CX **[EBADF]** The file descriptor underlying `stream` is not a valid file descriptor open for  
14471 writing.14472 CX **[EFBIG]** An attempt was made to write to a file that exceeds the maximum file size.14473 XSI **[EFBIG]** An attempt was made to write to a file that exceeds the file size limit of the  
14474 process.14475 CX **[EFBIG]** The file is a regular file and an attempt was made to write at or beyond the  
14476 offset maximum.14477 CX **[EINTR]** The write operation was terminated due to the receipt of a signal, and no data  
14478 was transferred.14479 CX **[EIO]** A physical I/O error has occurred, or the process is a member of a background  
14480 process group attempting to write to its controlling terminal, `TOSTOP` is set,  
14481 the process is neither ignoring nor blocking `SIGTTOU`, and the process group  
14482 of the process is orphaned. This error may also be returned under  
14483 implementation-defined conditions.14484 CX **[ENOSPC]** There was no free space remaining on the device containing the file.14485 CX **[EPIPE]** An attempt is made to write to a pipe or FIFO that is not open for reading by  
14486 any process. A `SIGPIPE` signal shall also be sent to the thread.

14487 The `fputc()` function may fail if:

14488 CX [ENOMEM] Insufficient storage space is available.

14489 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the  
14490 capabilities of the device.

#### 14491 EXAMPLES

14492 None.

#### 14493 APPLICATION USAGE

14494 None.

#### 14495 RATIONALE

14496 None.

#### 14497 FUTURE DIRECTIONS

14498 None.

#### 14499 SEE ALSO

14500 *error()*, *fopen()*, *getrlimit()*, *putc()*, *puts()*, *setbuf()*, *ulimit()*, the Base Definitions volume of  
14501 IEEE Std 1003.1-200x, `<stdio.h>`

#### 14502 CHANGE HISTORY

14503 First released in Issue 1. Derived from Issue 1 of the SVID.

#### 14504 Issue 5

14505 Large File Summit extensions are added.

#### 14506 Issue 6

14507 Extensions beyond the ISO C standard are marked.

14508 The following new requirements on POSIX implementations derive from alignment with the  
14509 Single UNIX Specification:

- 14510 • The [EIO] and [EFBIG] mandatory error conditions are added.
- 14511 • The [ENOMEM] and [ENXIO] optional error conditions are added.

14512 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/37 is applied, updating the [EAGAIN]  
14513 error in the ERRORS section from “the process would be delayed” to “the thread would be  
14514 delayed”.

14515 **NAME**

14516 fputs — put a string on a stream

14517 **SYNOPSIS**

14518 #include &lt;stdio.h&gt;

14519 int fputs(const char \*restrict s, FILE \*restrict stream);

14520 **DESCRIPTION**

14521 CX The functionality described on this reference page is aligned with the ISO C standard. Any  
 14522 conflict between the requirements described here and the ISO C standard is unintentional. This  
 14523 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

14524 The *fputs()* function shall write the null-terminated string pointed to by *s* to the stream pointed  
 14525 to by *stream*. The terminating null byte shall not be written.

14526 CX The *st\_ctime* and *st\_mtime* fields of the file shall be marked for update between the successful  
 14527 execution of *fputs()* and the next successful completion of a call to *fflush()* or *fclose()* on the same  
 14528 stream or a call to *exit()* or *abort()*.

14529 **RETURN VALUE**

14530 Upon successful completion, *fputs()* shall return a non-negative number. Otherwise, it shall  
 14531 CX return EOF, set an error indicator for the stream, and set *errno* to indicate the error.

14532 **ERRORS**14533 Refer to *fputc()*.14534 **EXAMPLES**14535 **Printing to Standard Output**

14536 The following example gets the current time, converts it to a string using *localtime()* and  
 14537 *asctime()*, and prints it to standard output using *fputs()*. It then prints the number of minutes to  
 14538 an event for which it is waiting.

```

14539 #include <time.h>
14540 #include <stdio.h>
14541 ...
14542 time_t now;
14543 int minutes_to_event;
14544 ...
14545 time(&now);
14546 printf("The time is ");
14547 fputs(asctime(localtime(&now)), stdout);
14548 printf("There are still %d minutes to the event.\n",
14549 minutes_to_event);
14550 ...

```

14551 **APPLICATION USAGE**14552 The *puts()* function appends a <newline> while *fputs()* does not.14553 **RATIONALE**

14554 None.

14555

**FUTURE DIRECTIONS**

14556

None.

14557

**SEE ALSO**

14558

*fopen()*, *putc()*, *puts()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdio.h>`

14559

**CHANGE HISTORY**

14560

First released in Issue 1. Derived from Issue 1 of the SVID.

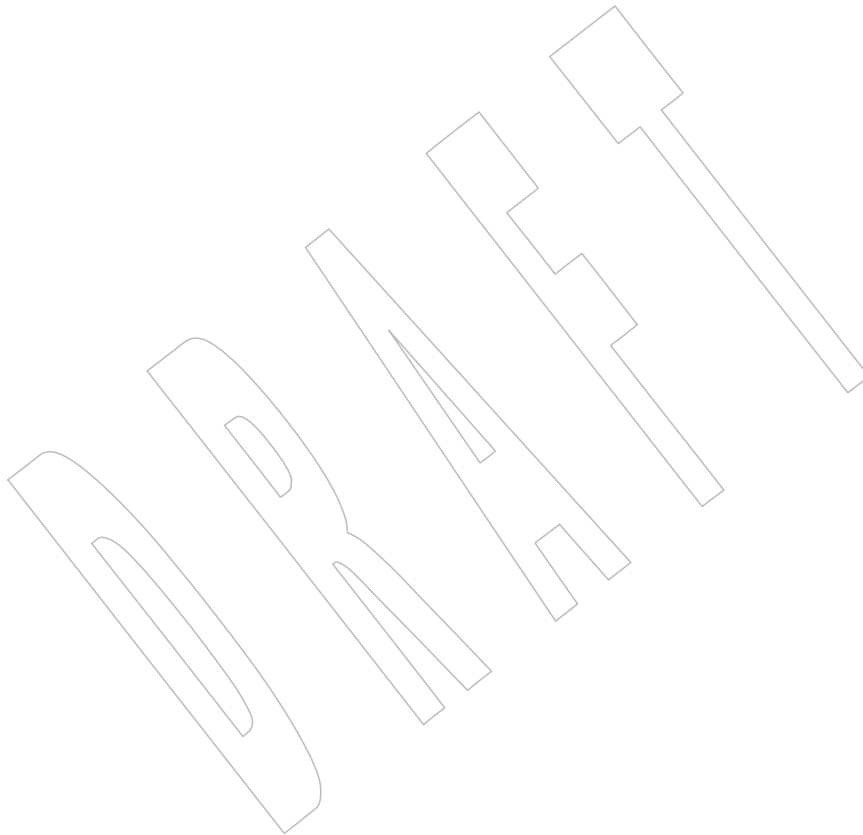
14561

**Issue 6**

14562

Extensions beyond the ISO C standard are marked.

14563

The *fputs()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

14564 **NAME**14565 `fputc` — put a wide-character code on a stream14566 **SYNOPSIS**14567 `#include <stdio.h>`14568 `#include <wchar.h>`14569 `wint_t fputc(wchar_t wc, FILE *stream);`14570 **DESCRIPTION**

14571 CX The functionality described on this reference page is aligned with the ISO C standard. Any  
 14572 conflict between the requirements described here and the ISO C standard is unintentional. This  
 14573 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

14574 The `fputc()` function shall write the character corresponding to the wide-character code `wc` to  
 14575 the output stream pointed to by `stream`, at the position indicated by the associated file-position  
 14576 indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot  
 14577 support positioning requests, or if the stream was opened with append mode, the character is  
 14578 appended to the output stream. If an error occurs while writing the character, the shift state of  
 14579 the output file is left in an undefined state.

14580 CX The `st_ctime` and `st_mtime` fields of the file shall be marked for update between the successful  
 14581 execution of `fputc()` and the next successful completion of a call to `fflush()` or `fclose()` on the  
 14582 same stream or a call to `exit()` or `abort()`.

14583 **RETURN VALUE**

14584 Upon successful completion, `fputc()` shall return `wc`. Otherwise, it shall return WEOF, the error  
 14585 indicator for the stream shall be set, and `errno` shall be set to indicate the error.

14586 **ERRORS**

14587 The `fputc()` function shall fail if either the stream is unbuffered or data in the `stream`'s buffer  
 14588 needs to be written, and:

14589 CX **[EAGAIN]** The `O_NONBLOCK` flag is set for the file descriptor underlying `stream` and  
 14590 the thread would be delayed in the write operation.

14591 CX **[EBADF]** The file descriptor underlying `stream` is not a valid file descriptor open for  
 14592 writing.

14593 CX **[EFBIG]** An attempt was made to write to a file that exceeds the maximum file size or  
 14594 the file size limit of the process.

14595 CX **[EFBIG]** The file is a regular file and an attempt was made to write at or beyond the  
 14596 offset maximum associated with the corresponding stream.

14597 **[EILSEQ]** The wide-character code `wc` does not correspond to a valid character.

14598 CX **[EINTR]** The write operation was terminated due to the receipt of a signal, and no data  
 14599 was transferred.

14600 CX **[EIO]** A physical I/O error has occurred, or the process is a member of a background  
 14601 process group attempting to write to its controlling terminal, `TOSTOP` is set,  
 14602 the process is neither ignoring nor blocking `SIGTTOU`, and the process group  
 14603 of the process is orphaned. This error may also be returned under  
 14604 implementation-defined conditions.

14605 CX **[ENOSPC]** There was no free space remaining on the device containing the file.

14606 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by  
14607 any process. A SIGPIPE signal shall also be sent to the thread.

14608 The `fputc()` function may fail if:

14609 CX [ENOMEM] Insufficient storage space is available.

14610 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the  
14611 capabilities of the device.

#### 14612 EXAMPLES

14613 None.

#### 14614 APPLICATION USAGE

14615 None.

#### 14616 RATIONALE

14617 None.

#### 14618 FUTURE DIRECTIONS

14619 None.

#### 14620 SEE ALSO

14621 *error()*, *fopen()*, *setbuf()*, *ulimit()*, the Base Definitions volume of IEEE Std 1003.1-200x,  
14622 `<stdio.h>`, `<wchar.h>`

#### 14623 CHANGE HISTORY

14624 First released in Issue 4. Derived from the MSE working draft.

#### 14625 Issue 5

14626 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument *wc*  
14627 is changed from `wint_t` to `wchar_t`.

14628 The Optional Header (OH) marking is removed from `<stdio.h>`.

14629 Large File Summit extensions are added.

#### 14630 Issue 6

14631 Extensions beyond the ISO C standard are marked.

14632 The following new requirements on POSIX implementations derive from alignment with the  
14633 Single UNIX Specification:

- 14634 • The [EFBIG] and [EIO] mandatory error conditions are added.
- 14635 • The [ENOMEM] and [ENXIO] optional error conditions are added.

14636 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/38 is applied, updating the [EAGAIN]  
14637 error in the ERRORS section from “the process would be delayed” to “the thread would be  
14638 delayed”.



14639 **NAME**14640 `fputws` — put a wide-character string on a stream14641 **SYNOPSIS**14642 `#include <stdio.h>`14643 `#include <wchar.h>`14644 `int fputws(const wchar_t *restrict ws, FILE *restrict stream);`14645 **DESCRIPTION**14646 CX The functionality described on this reference page is aligned with the ISO C standard. Any  
14647 conflict between the requirements described here and the ISO C standard is unintentional. This  
14648 volume of IEEE Std 1003.1-200x defers to the ISO C standard.14649 The `fputws()` function shall write a character string corresponding to the (null-terminated) wide-  
14650 character string pointed to by `ws` to the stream pointed to by `stream`. No character corresponding  
14651 to the terminating null wide-character code shall be written.14652 CX The `st_ctime` and `st_mtime` fields of the file shall be marked for update between the successful  
14653 execution of `fputws()` and the next successful completion of a call to `fflush()` or `fclose()` on the  
14654 same stream or a call to `exit()` or `abort()`.14655 **RETURN VALUE**14656 Upon successful completion, `fputws()` shall return a non-negative number. Otherwise, it shall14657 CX return `-1`, set an error indicator for the stream, and set `errno` to indicate the error.14658 **ERRORS**14659 Refer to `fputwc()`.14660 **EXAMPLES**

14661 None.

14662 **APPLICATION USAGE**14663 The `fputws()` function does not append a <newline>.14664 **RATIONALE**

14665 None.

14666 **FUTURE DIRECTIONS**

14667 None.

14668 **SEE ALSO**14669 `fopen()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdio.h>`, `<wchar.h>`14670 **CHANGE HISTORY**

14671 First released in Issue 4. Derived from the MSE working draft.

14672 **Issue 5**14673 The Optional Header (OH) marking is removed from `<stdio.h>`.14674 **Issue 6**

14675 Extensions beyond the ISO C standard are marked.

14676 The `fputws()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

14677 **NAME**

14678 fread — binary input

14679 **SYNOPSIS**

14680 #include &lt;stdio.h&gt;

14681 size\_t fread(void \*restrict ptr, size\_t size, size\_t nitems,  
14682 FILE \*restrict stream);14683 **DESCRIPTION**14684 CX The functionality described on this reference page is aligned with the ISO C standard. Any  
14685 conflict between the requirements described here and the ISO C standard is unintentional. This  
14686 volume of IEEE Std 1003.1-200x defers to the ISO C standard.14687 The *fread()* function shall read into the array pointed to by *ptr* up to *nitems* elements whose size  
14688 is specified by *size* in bytes, from the stream pointed to by *stream*. For each object, *size* calls shall  
14689 be made to the *fgetc()* function and the results stored, in the order read, in an array of **unsigned**  
14690 **char** exactly overlaying the object. The file position indicator for the stream (if defined) shall be  
14691 advanced by the number of bytes successfully read. If an error occurs, the resulting value of the  
14692 file position indicator for the stream is unspecified. If a partial element is read, its value is  
14693 unspecified.14694 CX The *fread()* function may mark the *st\_atime* field of the file associated with *stream* for update. The  
14695 *st\_atime* field shall be marked for update by the first successful execution of *fgetc()*, *fgets()*,  
14696 *fgetwc()*, *fgetws()*, *fread()*, *fscanf()*, *getc()*, *getchar()*, *gets()*, or *scanf()* using *stream* that returns  
14697 data not supplied by a prior call to *ungetc()* or *ungetwc()*.14698 **RETURN VALUE**14699 Upon successful completion, *fread()* shall return the number of elements successfully read which  
14700 is less than *nitems* only if a read error or end-of-file is encountered. If *size* or *nitems* is 0, *fread()*  
14701 shall return 0 and the contents of the array and the state of the stream remain unchanged.14702 CX Otherwise, if a read error occurs, the error indicator for the stream shall be set, and *errno* shall  
14703 be set to indicate the error.14704 **ERRORS**14705 Refer to *fgetc()*.14706 **EXAMPLES**14707 **Reading from a Stream**14708 The following example reads a single element from the *fp* stream into the array pointed to by  
14709 *buf*.14710 #include <stdio.h>  
14711 ...  
14712 size\_t bytes\_read;  
14713 char buf[100];  
14714 FILE \*fp;  
14715 ...  
14716 bytes\_read = fread(buf, sizeof(buf), 1, fp);  
14717 ...

14718  
14719  
14720  
  
14721  
14722  
14723  
  
14724  
14725  
  
14726  
14727  
  
14728  
14729  
14730  
  
14731  
14732  
  
14733  
14734  
14735  
14736  
14737

**APPLICATION USAGE**

The *ferror()* or *feof()* functions must be used to distinguish between an error condition and an end-of-file condition.

Because of possible differences in element length and byte ordering, files written using *fwrite()* are application-dependent, and possibly cannot be read using *fread()* by a different application or by the same application on a different processor.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

*feof()*, *ferror()*, *fgetc()*, *fopen()*, *getc()*, *gets()*, *scanf()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

**CHANGE HISTORY**

First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 6**

Extensions beyond the ISO C standard are marked.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The *fread()* prototype is updated.
- The DESCRIPTION is updated to describe how the bytes from a call to *fgetc()* are stored.

14738 **NAME**  
 14739 free — free allocated memory

14740 **SYNOPSIS**  
 14741 #include <stdlib.h>  
 14742 void free(void \*ptr);

14743 **DESCRIPTION**  
 14744 CX The functionality described on this reference page is aligned with the ISO C standard. Any  
 14745 conflict between the requirements described here and the ISO C standard is unintentional. This  
 14746 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

14747 The *free()* function shall cause the space pointed to by *ptr* to be deallocated; that is, made  
 14748 available for further allocation. If *ptr* is a null pointer, no action shall occur. Otherwise, if the  
 14749 ADV argument does not match a pointer earlier returned by the *calloc()*, *malloc()*, *posix\_memalign()*,  
 14750 *realloc()*, or *strdup()* function, or if the space has been deallocated by a call to *free()* or *realloc()*,  
 14751 the behavior is undefined.

14752 Any use of a pointer that refers to freed space results in undefined behavior.

14753 **RETURN VALUE**  
 14754 The *free()* function shall not return a value.

14755 **ERRORS**  
 14756 No errors are defined.

14757 **EXAMPLES**  
 14758 None.

14759 **APPLICATION USAGE**  
 14760 There is now no requirement for the implementation to support the inclusion of <malloc.h>.

14761 **RATIONALE**  
 14762 None.

14763 **FUTURE DIRECTIONS**  
 14764 None.

14765 **SEE ALSO**  
 14766 *calloc()*, *getdelim()*, *malloc()*, *open\_memstream()*, *realloc()*, *strdup()*, *wcsdup()*, the Base Definitions  
 14767 volume of IEEE Std 1003.1-200x, <stdlib.h>

14768 **CHANGE HISTORY**  
 14769 First released in Issue 1. Derived from Issue 1 of the SVID.

14770 **Issue 6**  
 14771 Reference to the *valloc()* function is removed.

**NAME**

freeaddrinfo, getaddrinfo — get address information

**SYNOPSIS**

```
#include <sys/socket.h>
#include <netdb.h>

void freeaddrinfo(struct addrinfo *ai);
int getaddrinfo(const char *restrict nodename,
 const char *restrict servname,
 const struct addrinfo *restrict hints,
 struct addrinfo **restrict res);
```

**DESCRIPTION**

The *freeaddrinfo()* function shall free one or more **addrinfo** structures returned by *getaddrinfo()*, along with any additional storage associated with those structures. If the *ai\_next* field of the structure is not null, the entire list of structures shall be freed. The *freeaddrinfo()* function shall support the freeing of arbitrary sublists of an **addrinfo** list originally returned by *getaddrinfo()*.

The *getaddrinfo()* function shall translate the name of a service location (for example, a host name) and/or a service name and shall return a set of socket addresses and associated information to be used in creating a socket with which to address the specified service.

**Note:** In many cases it is implemented by the Domain Name System, as documented in RFC 1034, RFC 1035, and RFC 1886.

The *freeaddrinfo()* and *getaddrinfo()* f

14818 for *ai\_socktype* means that the caller shall accept any socket type. A value of zero for *ai\_protocol*  
 14819 means that the caller shall accept any protocol. If *hints* is a null pointer, the behavior shall be as if  
 14820 it referred to a structure containing the value zero for the *ai\_flags*, *ai\_socktype*, and *ai\_protocol*  
 14821 fields, and AF\_UNSPEC for the *ai\_family* field.

14822 The *ai\_flags* field to which the *hints* parameter points shall be set to zero or be the bitwise-  
 14823 inclusive OR of one or more of the values AI\_PASSIVE, AI\_CANONNAME,  
 14824 AI\_NUMERICHOST, AI\_NUMERICSERV, AI\_V4MAPPED, AI\_ALL, and AI\_ADDRCONFIG.

14825 If the AI\_PASSIVE flag is specified, the returned address information shall be suitable for use in  
 14826 binding a socket for accepting incoming connections for the specified service. In this case, if the  
 14827 *nodename* argument is null, then the IP address portion of the socket address structure shall be  
 14828 set to INADDR\_ANY for an IPv4 address or IN6ADDR\_ANY\_INIT for an IPv6 address. If the  
 14829 AI\_PASSIVE flag is not specified, the returned address information shall be suitable for a call to  
 14830 *connect()* (for a connection-mode protocol) or for a call to *connect()*, *sendto()*, or *sendmsg()* (for a  
 14831 connectionless protocol). In this case, if the *nodename* argument is null, then the IP address  
 14832 portion of the socket address structure shall be set to the loopback address. The AI\_PASSIVE  
 14833 flag shall be ignored if the *nodename* argument is not null.

14834 If the AI\_CANONNAME flag is specified and the *nodename* argument is not null, the function  
 14835 shall attempt to determine the canonical name corresponding to *nodename* (for example, if  
 14836 *nodename* is an alias or shorthand notation for a complete name).

14837 **Note:** Since different implementations use different conceptual models, the terms “canonical name”  
 14838 and “alias” cannot be precisely defined for the general case. However, Domain Name System  
 14839 implementations are expected to interpret them as they are used in RFC 1034.

14840 A numeric host address string is not a “name”, and thus does not have a “canonical name”  
 14841 form; no address to host name translation is performed. See below for handling of the case  
 14842 where a canonical name cannot be obtained.

14843 If the AI\_NUMERICHOST flag is specified, then a non-null *nodename* string supplied shall be a  
 14844 numeric host address string. Otherwise, an [EAI\_NONAME] error is returned. This flag shall  
 14845 prevent any type of name resolution service (for example, the DNS) from being invoked.

14846 If the AI\_NUMERICSERV flag is specified, then a non-null *servname* string supplied shall be a  
 14847 numeric port string. Otherwise, an [EAI\_NONAME] error shall be returned. This flag shall  
 14848 prevent any type of name resolution service (for example, NIS+) from being invoked.

14849 IP6 If the AI\_V4MAPPED flag is specified along with an *ai\_family* of AF\_INET6, then *getaddrinfo()*  
 14850 shall return IPv4-mapped IPv6 addresses on finding no matching IPv6 addresses (*ai\_addrlen*  
 14851 shall be 16). The AI\_V4MAPPED flag shall be ignored unless *ai\_family* equals AF\_INET6. If the  
 14852 AI\_ALL flag is used with the AI\_V4MAPPED flag, then *getaddrinfo()* shall return all matching  
 14853 IPv6 and IPv4 addresses. The AI\_ALL flag without the AI\_V4MAPPED flag is ignored.

14854 IP6 If the AI\_ADDRCONFIG flag is specified, IPv4 addresses shall be returned only if an IPv4  
 14855 address is configured on the local system, and IPv6 addresses shall be returned only if an IPv6  
 14856 address is configured on the local system.

14857 The *ai\_socktype* field to which argument *hints* points specifies the socket type for the service, as  
 14858 defined in *socket()*. If a specific socket type is not given (for example, a value of zero) and the  
 14859 service name could be interpreted as valid with multiple supported socket types, the  
 14860 implementation shall attempt to resolve the service name for all supported socket types and, in  
 14861 the absence of errors, all possible results shall be returned. A non-zero socket type value shall  
 14862 limit the returned information to values with the specified socket type.

14863 If the *ai\_family* field to which *hints* points has the value AF\_UNSPEC, addresses shall be  
 14864 returned for use with any address family that can be used with the specified *nodename* and/or  
 14865 *servname*. Otherwise, addresses shall be returned for use only with the specified address family.  
 14866 If *ai\_family* is not AF\_UNSPEC and *ai\_protocol* is not zero, then addresses are returned for use

14867 only with the specified address family and protocol; the value of *ai\_protocol* shall be interpreted  
 14868 as in a call to the *socket()* function with the corresponding values of *ai\_family* and *ai\_protocol*.

**RETURN VALUE**

14869 A zero return value for *getaddrinfo()* indicates successful completion; a non-zero return value  
 14870 indicates failure. The possible values for the failures are listed in the ERRORS section.  
 14871

14872 Upon successful return of *getaddrinfo()*, the location to which *res* points shall refer to a linked list  
 14873 of **addrinfo** structures, each of which shall specify a socket address and information for use in  
 14874 creating a socket with which to use that socket address. The list shall include at least one  
 14875 **addrinfo** structure. The *ai\_next* field of each structure contains a pointer to the next structure on  
 14876 the list, or a null pointer if it is the last structure on the list. Each structure on the list shall  
 14877 include values for use with a call to the *socket()* function, and a socket address for use with the  
 14878 *connect()* function or, if the AI\_PASSIVE flag was specified, for use with the *bind()* function. The  
 14879 fields *ai\_family*, *ai\_socktype*, and *ai\_protocol* shall be usable as the arguments to the *socket()*  
 14880 function to create a socket suitable for use with the returned address. The fields *ai\_addr* and  
 14881 *ai\_addrlen* are usable as the arguments to the *connect()* or *bind()* functions with such a socket,  
 14882 according to the AI\_PASSIVE flag.

14883 If *nodename* is not null, and if requested by the AI\_CANONNAME flag, the *ai\_canonname* field of  
 14884 the first returned **addrinfo** structure shall point to a null-terminated string containing the  
 14885 canonical name corresponding to the input *nodename*; if the canonical name is not available, then  
 14886 *ai\_canonname* shall refer to the *nodename* argument or a string with the same contents. The  
 14887 contents of the *ai\_flags* field of the returned structures are undefined.

14888 All fields in socket address structures returned by *getaddrinfo()* that are not filled in through an  
 14889 explicit argument (for example, *sin6\_flowinfo*) shall be set to zero.

14890 **Note:** This makes it easier to compare socket address structures.

**ERRORS**

14891 The *getaddrinfo()* function shall fail and return the corresponding error value if:

14892 [EAI\_AGAIN] The name could not be resolved at this time. Future attempts may succeed.

14893 [EAI\_BADFLAGS] The *flags* parameter had an invalid value.  
 14894

14895 [EAI\_FAIL] A non-recoverable error occurred when attempting to resolve the name.

14896 [EAI\_FAMILY] The address family was not recognized.

14897 [EAI\_MEMORY] There was a memory allocation failure when trying to allocate storage for the  
 14898 return value.  
 14899

14900 [EAI\_NONAME] The name does not resolve for the supplied parameters.

14901 Neither *nodename* nor *servname* were supplied. At least one of these shall be  
 14902 supplied.

14903 [EAI\_SERVICE] The service passed was not recognized for the specified socket type.

14904 [EAI\_SOCKTYPE] The intended socket type was not recognized.  
 14905

14906 [EAI\_SYSTEM] A system error occurred; the error code can be found in *errno*.



**EXAMPLES**

None.

**APPLICATION USAGE**

If the caller handles only TCP and not UDP, for example, then the *ai\_protocol* member of the *hints* structure should be set to IPPROTO\_TCP when *getaddrinfo()* is called.

If the caller handles only IPv4 and not IPv6, then the *ai\_family* member of the *hints* structure should be set to AF\_INET when *getaddrinfo()* is called.

The term “canonical name” is misleading; it is taken from the Domain Name System (RFC 2181). It should be noted that the canonical name is a result of alias processing, and not necessarily a unique attribute of a host, address, or set of addresses. See RFC 2181 for more discussion of this in the Domain Name System context.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

*connect()*, *gai\_strerror()*, *getnameinfo()*, *getserobyname()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<netdb.h>`, `<sys/socket.h>`

**CHANGE HISTORY**

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The **restrict** keyword is added to the *getaddrinfo()* prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/19 is applied, adding three notes to the DESCRIPTION and adding text to the APPLICATION USAGE related to the term “canonical name”. A reference to RFC 2181 is also added to the Informative References.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/20 is applied, making changes for alignment with IPv6. These include the following:

- Adding AI\_V4MAPPED, AI\_ALL, and AI\_ADDRCONFIG to the allowed values for the *ai\_flags* field
- Adding a description of AI\_ADDRCONFIG
- Adding a description of the consequences of ignoring the AI\_PASSIVE flag.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/39 is applied, changing “corresponding value” to “corresponding error value” in the ERRORS section.

**Issue 7**

Austin Group Interpretation 1003.1-2001 #013 is applied.



14942 **NAME**14943 `freelocale` — free resources allocated for a locale object14944 **SYNOPSIS**

```
14945 CX #include <locale.h>
14946 void freelocale(locale_t locobj);
```

14947 **DESCRIPTION**

14948 The `freelocale()` function shall cause the resources allocated for a locale object returned by a call  
 14949 to the `newlocale()` or `duplocale()` functions to be released.

14950 Any use of a locale object that has been freed results in undefined behavior.

14951 **RETURN VALUE**

14952 None.

14953 **ERRORS**

14954 None.

14955 **EXAMPLES**14956 **Freeing Up a Locale Object**

14957 The following example shows a code fragment to free a locale object created by `newlocale()`:

```
14958 #include <locale.h>
14959 ...
14960 /* Every locale object allocated with newlocale() should be
14961 * freed using freelocale():
14962 */
14963 locale_t loc;
14964 /* Get the locale. */
14965 loc = newlocale (LC_CTYPE_MASK | LC_TIME_MASK, "locname", NULL);
14966 /* ... Use the locale object ... */
14967 ...
14968 /* Free the locale object resources. */
14969 freelocale (loc);
```

14970 **APPLICATION USAGE**

14971 None.

14972 **RATIONALE**

14973 None.

14974 **FUTURE DIRECTIONS**

14975 None.

14976 **SEE ALSO**

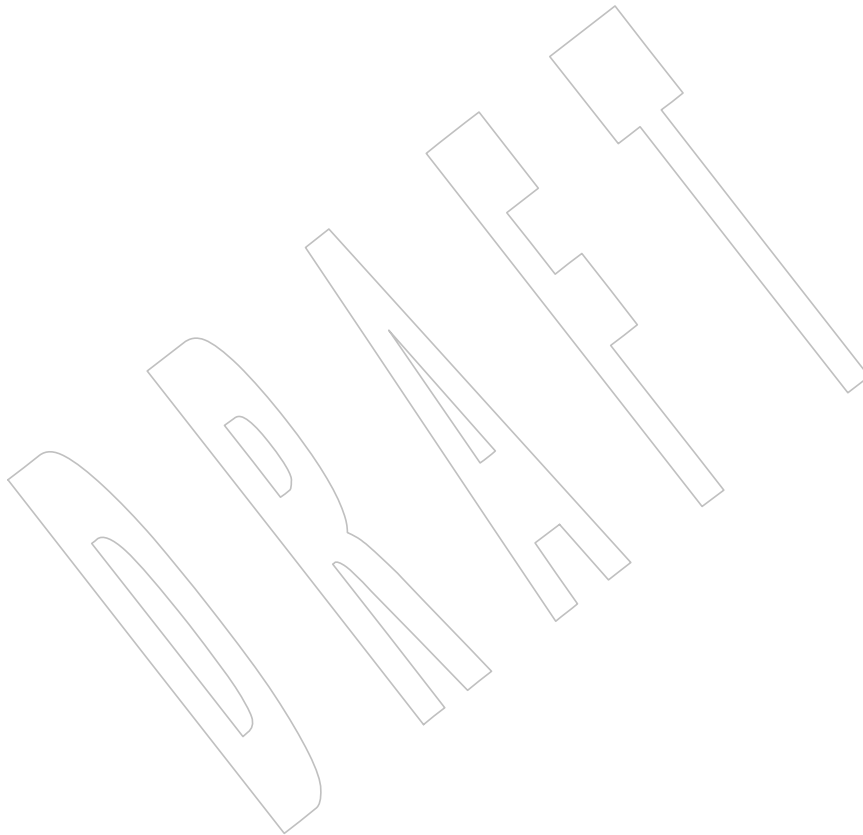
14977 [duplocale\(\)](#), [newlocale\(\)](#), [uselocale\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x,  
 14978 [<locale.h>](#)

14979

14980

**CHANGE HISTORY**

First released in Issue 7.



14981 **NAME**14982 `freopen` — open a stream14983 **SYNOPSIS**14984 `#include <stdio.h>`14985 `FILE *freopen(const char *restrict filename, const char *restrict mode,`  
14986 `FILE *restrict stream);`14987 **DESCRIPTION**14988 CX The functionality described on this reference page is aligned with the ISO C standard. Any  
14989 conflict between the requirements described here and the ISO C standard is unintentional. This  
14990 volume of IEEE Std 1003.1-200x defers to the ISO C standard.14991 The `freopen()` function shall first attempt to flush the stream and close any file descriptor  
14992 associated with `stream`. Failure to flush or close the file descriptor successfully shall be ignored.  
14993 The error and end-of-file indicators for the stream shall be cleared.14994 The `freopen()` function shall open the file whose pathname is the string pointed to by `filename`  
14995 and associate the stream pointed to by `stream` with it. The `mode` argument shall be used just as in  
14996 `fopen()`. The `freopen()` function shall allocate a file descriptor in the same way as `open()`.

14997 The original stream shall be closed regardless of whether the subsequent open succeeds.

14998 If `filename` is a null pointer, the `freopen()` function shall attempt to change the mode of the stream  
14999 to that specified by `mode`, as if the name of the file currently associated with the stream had been  
15000 used. In this case, the file descriptor associated with the stream need not be closed if the call to  
15001 `freopen()` succeeds. It is implementation-defined which changes of mode are permitted (if any),  
15002 and under what circumstances.15003 XSI After a successful call to the `freopen()` function, the orientation of the stream shall be cleared, the  
15004 encoding rule shall be cleared, and the associated `mbstate_t` object shall be set to describe an  
15005 initial conversion state.15006 CX The largest value that can be represented correctly in an object of type `off_t` shall be established  
15007 as the offset maximum in the open file description.15008 **RETURN VALUE**15009 Upon successful completion, `freopen()` shall return the value of `stream`. Otherwise, a null pointer  
15010 shall be returned, and `errno` shall be set to indicate the error.15011 **ERRORS**15012 The `freopen()` function shall fail if:15013 CX [EACCES] Search permission is denied on a component of the path prefix, or the file  
15014 exists and the permissions specified by `mode` are denied, or the file does not  
15015 exist and write permission is denied for the parent directory of the file to be  
15016 created.15017 CX [EBADF] The file descriptor underlying the stream is not a valid file descriptor when  
15018 `filename` is a null pointer.15019 CX [EINTR] A signal was caught during `freopen()`.15020 CX [EISDIR] The named file is a directory and `mode` requires write access.15021 CX [ELOOP] A loop exists in symbolic links encountered during resolution of the `path`  
15022 argument.

|       |    |                |                                                                                                                                                      |
|-------|----|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| 15023 | CX | [EMFILE]       | All file descriptors available to the process are currently open.                                                                                    |
| 15024 | CX | [ENAMETOOLONG] |                                                                                                                                                      |
| 15025 |    |                | The length of the <i>filename</i> argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.                                     |
| 15026 |    |                |                                                                                                                                                      |
| 15027 | CX | [ENFILE]       | The maximum allowable number of files is currently open in the system.                                                                               |
| 15028 | CX | [ENOENT]       | A component of <i>filename</i> does not name an existing file or <i>filename</i> is an empty string.                                                 |
| 15029 |    |                |                                                                                                                                                      |
| 15030 | CX | [ENOSPC]       | The directory or file system that would contain the new file cannot be expanded, the file does not exist, and it was to be created.                  |
| 15031 |    |                |                                                                                                                                                      |
| 15032 | CX | [ENOTDIR]      | A component of the path prefix is not a directory.                                                                                                   |
| 15033 | CX | [ENXIO]        | The named file is a character special or block special file, and the device associated with this special file does not exist.                        |
| 15034 |    |                |                                                                                                                                                      |
| 15035 | CX | [EOVERFLOW]    | The named file is a regular file and the size of the file cannot be represented correctly in an object of type <code>off_t</code> .                  |
| 15036 |    |                |                                                                                                                                                      |
| 15037 | CX | [EROFS]        | The named file resides on a read-only file system and <i>mode</i> requires write access.                                                             |
| 15038 |    |                |                                                                                                                                                      |
| 15039 |    |                | The <i>freopen()</i> function may fail if:                                                                                                           |
| 15040 | CX | [EBADF]        | The mode with which the file descriptor underlying the stream was opened does not support the requested mode when <i>filename</i> is a null pointer. |
| 15041 |    |                |                                                                                                                                                      |
| 15042 | CX | [EINVAL]       | The value of the <i>mode</i> argument is not valid.                                                                                                  |
| 15043 | CX | [ELOOP]        | More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the <i>path</i> argument.                                               |
| 15044 |    |                |                                                                                                                                                      |
| 15045 | CX | [ENAMETOOLONG] |                                                                                                                                                      |
| 15046 |    |                | Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.                                              |
| 15047 |    |                |                                                                                                                                                      |
| 15048 | CX | [ENOMEM]       | Insufficient storage space is available.                                                                                                             |
| 15049 | CX | [ENXIO]        | A request was made of a nonexistent device, or the request was outside the capabilities of the device.                                               |
| 15050 |    |                |                                                                                                                                                      |
| 15051 | CX | [ETXTBSY]      | The file is a pure procedure (shared text) file that is being executed and <i>mode</i> requires write access.                                        |
| 15052 |    |                |                                                                                                                                                      |

**EXAMPLES****Directing Standard Output to a File**

The following example logs all standard output to the `/tmp/logfile` file.

```
#include <stdio.h>
...
FILE *fp;
...
fp = freopen ("/tmp/logfile", "a+", stdout);
...
```

**APPLICATION USAGE**

The *freopen()* function is typically used to attach the preopened *streams* associated with *stdin*, *stdout*, and *stderr* to other files.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

*fclose()*, *fdopen()*, *fmemopen()*, *fopen()*, *mbsinit()*, *open()*, *open\_memstream()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**stdio.h**>

**CHANGE HISTORY**

First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 5**

The DESCRIPTION is updated to indicate that the orientation of the stream is cleared and the conversion state of the stream is set to an initial conversion state by a successful call to the *freopen()* function.

Large File Summit extensions are added.

**Issue 6**

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file description. This change is to support large files.
- In the ERRORS section, the [Eoverflow] condition is added. This change is to support large files.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.
- The [EINVAL], [ENOMEM], [ENXIO], and [ETXTBSY] optional error conditions are added.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The *freopen()* prototype is updated.
- The DESCRIPTION is updated.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

The DESCRIPTION is updated regarding failure to close, changing the “file” to “file descriptor”.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/40 is applied, adding the following sentence to the DESCRIPTION: “In this case, the file descriptor associated with the stream need not be closed if the call to *freopen()* succeeds.”.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/41 is applied, adding an mandatory [EBADF] error, and an optional [EBADF] error to the ERRORS section.

**Issue 7**

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

Austin Group Interpretation 1003.1-2001 #043 is applied, clarifying that the *freopen()* function allocates a file descriptor as per *open()*.



15105 **NAME**  
 15106 `frexp, frexpf, frexpl` — extract mantissa and exponent from a double precision number

15107 **SYNOPSIS**  
 15108 `#include <math.h>`  
 15109 `double frexp(double num, int *exp);`  
 15110 `float frexpf(float num, int *exp);`  
 15111 `long double frexpl(long double num, int *exp);`

15112 **DESCRIPTION**  
 15113 CX The functionality described on this reference page is aligned with the ISO C standard. Any  
 15114 conflict between the requirements described here and the ISO C standard is unintentional. This  
 15115 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

15116 These functions shall break a floating-point number *num* into a normalized fraction and an  
 15117 integral power of 2. The integer exponent shall be stored in the **int** object pointed to by *exp*.

15118 **RETURN VALUE**  
 15119 For finite arguments, these functions shall return the value *x*, such that *x* has a magnitude in the  
 15120 interval  $[\frac{1}{2}, 1)$  or 0, and *num* equals *x* times 2 raised to the power *\*exp*.

15121 MX If *num* is NaN, a NaN shall be returned, and the value of *\*exp* is unspecified.  
 15122 If *num* is  $\pm 0$ ,  $\pm 0$  shall be returned, and the value of *\*exp* shall be 0.  
 15123 If *num* is  $\pm \text{Inf}$ , *num* shall be returned, and the value of *\*exp* is unspecified.

15124 **ERRORS**  
 15125 No errors are defined.

15126 **EXAMPLES**  
 15127 None.

15128 **APPLICATION USAGE**  
 15129 None.

15130 **RATIONALE**  
 15131 None.

15132 **FUTURE DIRECTIONS**  
 15133 None.

15134 **SEE ALSO**  
 15135 `isnan()`, `ldexp()`, `modf()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<math.h>`

15136 **CHANGE HISTORY**  
 15137 First released in Issue 1. Derived from Issue 1 of the SVID.

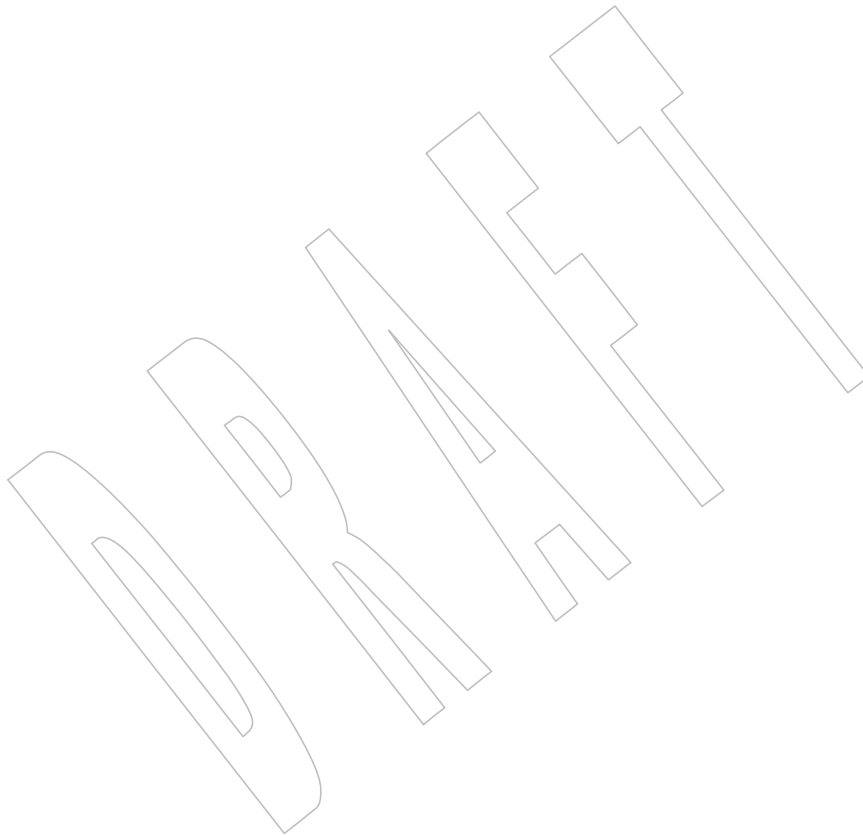
15138 **Issue 5**  
 15139 The DESCRIPTION is updated to indicate how an application should check for an error. This  
 15140 text was previously published in the APPLICATION USAGE section.

15141  
15142  
15143  
15144  
15145  
15146  
15147**Issue 6**

The *frexpf()* and *frexpl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.





15148 **NAME**  
 15149 fscanf, scanf, sscanf — convert formatted input

15150 **SYNOPSIS**

```
15151 #include <stdio.h>

15152 int fscanf(FILE *restrict stream, const char *restrict format, ...);
15153 int scanf(const char *restrict format, ...);
15154 int sscanf(const char *restrict s, const char *restrict format, ...);
```

15155 **DESCRIPTION**

15156 CX The functionality described on this reference page is aligned with the ISO C standard. Any  
 15157 conflict between the requirements described here and the ISO C standard is unintentional. This  
 15158 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

15159 The *fscanf()* function shall read from the named input *stream*. The *scanf()* function shall read  
 15160 from the standard input stream *stdin*. The *sscanf()* function shall read from the string *s*. Each  
 15161 function reads bytes, interprets them according to a *format*, and stores the results in its  
 15162 arguments. Each expects, as arguments, a control string *format* described below, and a set of  
 15163 *pointer* arguments indicating where the converted input should be stored. The result is  
 15164 undefined if there are insufficient arguments for the *format*. If the *format* is exhausted while  
 15165 arguments remain, the excess arguments shall be evaluated but otherwise ignored.

15166 CX Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than  
 15167 to the next unused argument. In this case, the conversion specifier character % (see below) is  
 15168 replaced by the sequence "%n\$", where *n* is a decimal integer in the range [1,{NL\_ARGMAX}].  
 15169 This feature provides for the definition of format strings that select arguments in an order  
 15170 appropriate to specific languages. In format strings containing the "%n\$" form of conversion  
 15171 specifications, it is unspecified whether numbered arguments in the argument list can be  
 15172 referenced from the format string more than once.

15173 The *format* can contain either form of a conversion specification—that is, % or "%n\$"—but the  
 15174 two forms cannot be mixed within a single *format* string. The only exception to this is that %% or  
 15175 %\* can be mixed with the "%n\$" form. When numbered argument specifications are used,  
 15176 specifying the *N*th argument requires that all the leading arguments, from the first to the  
 15177 (*N*–1)th, are pointers.

15178 The *fscanf()* function in all its forms shall allow detection of a language-dependent radix  
 15179 character in the input string. The radix character is defined in the locale of the process (category  
 15180 LC\_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the  
 15181 radix character shall default to a period ('.').

15182 The *format* is a character string, beginning and ending in its initial shift state, if any, composed  
 15183 of zero or more directives. Each directive is composed of one of the following: one or more  
 15184 white-space characters (<space>s, <tab>s, <newline>s, <vertical-tab>s, or <form-feed>s); an  
 15185 ordinary character (neither '%' nor a white-space character); or a conversion specification. Each  
 15186 CX conversion specification is introduced by the character '%' or the character sequence "%n\$",  
 15187 after which the following appear in sequence:

- 15188 • An optional assignment-suppressing character '\*'.
- 15189 • An optional non-zero decimal integer that specifies the maximum field width.
- 15190 • An option length modifier that specifies the size of the receiving object.
- 15191 • A *conversion specifier* character that specifies the type of conversion to be applied. The valid  
 15192 conversion specifiers are described below.

15193 The *fscanf()* functions shall execute each directive of the format in turn. If a directive fails, as  
 15194 detailed below, the function shall return. Failures are described as input failures (due to the  
 15195 unavailability of input bytes) or matching failures (due to inappropriate input).

15196 A directive composed of one or more white-space characters shall be executed by reading input  
 15197 until no more valid input can be read, or up to the first byte which is not a white-space character,  
 15198 which remains unread.

15199 A directive that is an ordinary character shall be executed as follows: the next byte shall be read  
 15200 from the input and compared with the byte that comprises the directive; if the comparison  
 15201 shows that they are not equivalent, the directive shall fail, and the differing and subsequent  
 15202 bytes shall remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a  
 15203 character from being read, the directive shall fail.

15204 A directive that is a conversion specification defines a set of matching input sequences, as  
 15205 described below for each conversion character. A conversion specification shall be executed in  
 15206 the following steps.

15207 Input white-space characters (as specified by *isspace()*) shall be skipped, unless the conversion  
 15208 specification includes a `[`, `c`, `C`, or `n` conversion specifier.

15209 An item shall be read from the input, unless the conversion specification includes an `n`  
 15210 conversion specifier. An input item shall be defined as the longest sequence of input bytes (up to  
 15211 any specified maximum field width, which may be measured in characters or bytes dependent  
 15212 on the conversion specifier) which is an initial subsequence of a matching sequence. The first  
 15213 byte, if any, after the input item shall remain unread. If the length of the input item is 0, the  
 15214 execution of the conversion specification shall fail; this condition is a matching failure, unless  
 15215 end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is  
 15216 an input failure.

15217 Except in the case of a `%` conversion specifier, the input item (or, in the case of a `%n` conversion  
 15218 specification, the count of input bytes) shall be converted to a type appropriate to the conversion  
 15219 character. If the input item is not a matching sequence, the execution of the conversion  
 15220 specification fails; this condition is a matching failure. Unless assignment suppression was  
 15221 indicated by a `'*'`, the result of the conversion shall be placed in the object pointed to by the  
 15222 first argument following the *format* argument that has not already received a conversion result if  
 15223 the conversion specification is introduced by `%`, or in the *n*th argument if introduced by the  
 15224 character sequence `"%n$"`. If this object does not have an appropriate type, or if the result of the  
 15225 conversion cannot be represented in the space provided, the behavior is undefined.

15226 The length modifiers and their meanings are:

15227 `hh` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an  
 15228 argument with type pointer to **signed char** or **unsigned char**.

15229 `h` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an  
 15230 argument with type pointer to **short** or **unsigned short**.

15231 `l (ell)` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an  
 15232 argument with type pointer to **long** or **unsigned long**; that a following `a`, `A`, `e`, `E`, `f`, `F`,  
 15233 `g`, or `G` conversion specifier applies to an argument with type pointer to **double**; or that  
 15234 a following `c`, `s`, or `[` conversion specifier applies to an argument with type pointer to  
 15235 **wchar\_t**.

15236 `ll (ell-ell)`  
 15237 Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an  
 15238 argument with type pointer to **long long** or **unsigned long long**.

- 15239           j       Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an  
15240           argument with type pointer to **intmax\_t** or **uintmax\_t**.
- 15241           z       Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an  
15242           argument with type pointer to **size\_t** or the corresponding signed integer type.
- 15243           t       Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an  
15244           argument with type pointer to **ptrdiff\_t** or the corresponding **unsigned** type.
- 15245           L       Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an  
15246           argument with type pointer to **long double**.
- 15247           If a length modifier appears with any conversion specifier other than as specified above, the  
15248           behavior is undefined.
- 15249           The following conversion specifiers are valid:
- 15250           d       Matches an optionally signed decimal integer, whose format is the same as expected for  
15251           the subject sequence of *strtol()* with the value 10 for the *base* argument. In the absence  
15252           of a size modifier, the application shall ensure that the corresponding argument is a  
15253           pointer to **int**.
- 15254           i       Matches an optionally signed integer, whose format is the same as expected for the  
15255           subject sequence of *strtol()* with 0 for the *base* argument. In the absence of a size  
15256           modifier, the application shall ensure that the corresponding argument is a pointer to  
15257           **int**.
- 15258           o       Matches an optionally signed octal integer, whose format is the same as expected for  
15259           the subject sequence of *strtoul()* with the value 8 for the *base* argument. In the absence  
15260           of a size modifier, the application shall ensure that the corresponding argument is a  
15261           pointer to **unsigned**.
- 15262           u       Matches an optionally signed decimal integer, whose format is the same as expected for  
15263           the subject sequence of *strtoul()* with the value 10 for the *base* argument. In the absence  
15264           of a size modifier, the application shall ensure that the corresponding argument is a  
15265           pointer to **unsigned**.
- 15266           x       Matches an optionally signed hexadecimal integer, whose format is the same as  
15267           expected for the subject sequence of *strtoul()* with the value 16 for the *base* argument. In  
15268           the absence of a size modifier, the application shall ensure that the corresponding  
15269           argument is a pointer to **unsigned**.
- 15270           a, e, f, g       Matches an optionally signed floating-point number, infinity, or NaN, whose format is  
15271           the same as expected for the subject sequence of *strtod()*. In the absence of a size  
15272           modifier, the application shall ensure that the corresponding argument is a pointer to  
15273           **float**.  
15274
- 15275           If the *fprintf()* family of functions generates character string representations for infinity  
15276           and NaN (a symbolic entity encoded in floating-point format) to support  
15277           IEEE Std 754-1985, the *fscanf()* family of functions shall recognize them as input.
- 15278           s       Matches a sequence of bytes that are not white-space characters. The application shall  
15279           ensure that the corresponding argument is a pointer to the initial byte of an array of  
15280           **char**, **signed char**, or **unsigned char** large enough to accept the sequence and a  
15281           terminating null character code, which shall be added automatically.
- 15282           If an l (ell) qualifier is present, the input is a sequence of characters that begins in the  
15283           initial shift state. Each character shall be converted to a wide character as if by a call to  
15284           the *mbrtowc()* function, with the conversion state described by an **mbstate\_t** object  
15285           initialized to zero before the first character is converted. The application shall ensure

- 15286 that the corresponding argument is a pointer to an array of **wchar\_t** large enough to  
 15287 accept the sequence and the terminating null wide character, which shall be added  
 15288 automatically.
- 15289 [ Matches a non-empty sequence of bytes from a set of expected bytes (the *scanset*). The  
 15290 normal skip over white-space characters shall be suppressed in this case. The  
 15291 application shall ensure that the corresponding argument is a pointer to the initial byte  
 15292 of an array of **char**, **signed char**, or **unsigned char** large enough to accept the sequence  
 15293 and a terminating null byte, which shall be added automatically.
- 15294 If an **l** (ell) qualifier is present, the input is a sequence of characters that begins in the  
 15295 initial shift state. Each character in the sequence shall be converted to a wide character  
 15296 as if by a call to the *mbrtowc()* function, with the conversion state described by an  
 15297 **mbstate\_t** object initialized to zero before the first character is converted. The  
 15298 application shall ensure that the corresponding argument is a pointer to an array of  
 15299 **wchar\_t** large enough to accept the sequence and the terminating null wide character,  
 15300 which shall be added automatically.
- 15301 The conversion specification includes all subsequent bytes in the *format* string up to  
 15302 and including the matching right square bracket (']'). The bytes between the square  
 15303 brackets (the *scanlist*) comprise the *scanset*, unless the byte after the left square bracket  
 15304 is a circumflex ('^'), in which case the *scanset* contains all bytes that do not appear in  
 15305 the *scanlist* between the circumflex and the right square bracket. If the conversion  
 15306 specification begins with "[ ]" or "[^]", the right square bracket is included in the  
 15307 *scanlist* and the next right square bracket is the matching right square bracket that ends  
 15308 the conversion specification; otherwise, the first right square bracket is the one that  
 15309 ends the conversion specification. If a '-' is in the *scanlist* and is not the first character,  
 15310 nor the second where the first character is a '^', nor the last character, the behavior is  
 15311 implementation-defined.
- 15312 c Matches a sequence of bytes of the number specified by the field width (1 if no field  
 15313 width is present in the conversion specification). The application shall ensure that the  
 15314 corresponding argument is a pointer to the initial byte of an array of **char**, **signed char**,  
 15315 or **unsigned char** large enough to accept the sequence. No null byte is added. The  
 15316 normal skip over white-space characters shall be suppressed in this case.
- 15317 If an **l** (ell) qualifier is present, the input shall be a sequence of characters that begins in  
 15318 the initial shift state. Each character in the sequence is converted to a wide character as  
 15319 if by a call to the *mbrtowc()* function, with the conversion state described by an  
 15320 **mbstate\_t** object initialized to zero before the first character is converted. The  
 15321 application shall ensure that the corresponding argument is a pointer to an array of  
 15322 **wchar\_t** large enough to accept the resulting sequence of wide characters. No null wide  
 15323 character is added.
- 15324 p Matches an implementation-defined set of sequences, which shall be the same as the set  
 15325 of sequences that is produced by the %p conversion specification of the corresponding  
 15326 *fprintf()* functions. The application shall ensure that the corresponding argument is a  
 15327 pointer to a pointer to **void**. The interpretation of the input item is implementation-  
 15328 defined. If the input item is a value converted earlier during the same program  
 15329 execution, the pointer that results shall compare equal to that value; otherwise, the  
 15330 behavior of the %p conversion specification is undefined.
- 15331 n No input is consumed. The application shall ensure that the corresponding argument is  
 15332 a pointer to the integer into which shall be written the number of bytes read from the  
 15333 input so far by this call to the *fscanf()* functions. Execution of a %n conversion  
 15334 specification shall not increment the assignment count returned at the completion of  
 15335 execution of the function. No argument shall be converted, but one shall be consumed.  
 15336 If the conversion specification includes an assignment-suppressing character or a field

|       |     |          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------|-----|----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 15337 |     |          | width, the behavior is undefined.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 15338 | XSI | <b>C</b> | Equivalent to <code>lc</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15339 | XSI | <b>S</b> | Equivalent to <code>ls</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15340 |     | <b>%</b> | Matches a single <code>'%'</code> character; no conversion or assignment occurs. The complete conversion specification shall be <code>%%</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 15341 |     |          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15342 |     |          | If a conversion specification is invalid, the behavior is undefined.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| 15343 |     |          | The conversion specifiers <code>A</code> , <code>E</code> , <code>F</code> , <code>G</code> , and <code>X</code> are also valid and shall be equivalent to <code>a</code> , <code>e</code> , <code>f</code> , <code>g</code> , and <code>x</code> , respectively.                                                                                                                                                                                                                                                                                                                       |
| 15344 |     |          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15345 |     |          | If end-of-file is encountered during input, conversion shall be terminated. If end-of-file occurs before any bytes matching the current conversion specification (except for <code>%n</code> ) have been read (other than leading white-space characters, where permitted), execution of the current conversion specification shall terminate with an input failure. Otherwise, unless execution of the current conversion specification is terminated with a matching failure, execution of the following conversion specification (if any) shall be terminated with an input failure. |
| 15346 |     |          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15347 |     |          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15348 |     |          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15349 |     |          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15350 |     |          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15351 |     |          | Reaching the end of the string in <code>sscanf()</code> shall be equivalent to encountering end-of-file for <code>fscanf()</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| 15352 |     |          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15353 |     |          | If conversion terminates on a conflicting input, the offending input is left unread in the input. Any trailing white space (including <code>&lt;newline&gt;</code> s) shall be left unread unless matched by a conversion specification. The success of literal matches and suppressed assignments is only directly determinable via the <code>%n</code> conversion specification.                                                                                                                                                                                                      |
| 15354 |     |          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15355 |     |          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15356 |     |          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15357 | CX  |          | The <code>fscanf()</code> and <code>scanf()</code> functions may mark the <code>st_atime</code> field of the file associated with <code>stream</code> for update. The <code>st_atime</code> field shall be marked for update by the first successful execution of <code>fgetc()</code> , <code>fgets()</code> , <code>fread()</code> , <code>getc()</code> , <code>getchar()</code> , <code>gets()</code> , <code>fscanf()</code> , or <code>scanf()</code> using <code>stream</code> that returns data not supplied by a prior call to <code>ungetc()</code> .                         |
| 15358 |     |          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15359 |     |          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15360 |     |          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

**RETURN VALUE**

|       |    |  |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------|----|--|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 15361 |    |  | Upon successful completion, these functions shall return the number of successfully matched and assigned input items; this number can be zero in the event of an early matching failure. If the input ends before the first matching failure or conversion, EOF shall be returned. If a read error occurs, the error indicator for the stream is set, EOF shall be returned, and <code>errno</code> shall be set to indicate the error. |
| 15362 |    |  |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15363 |    |  |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15364 |    |  |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15365 | CX |  |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 15366 |    |  |                                                                                                                                                                                                                                                                                                                                                                                                                                         |

**ERRORS**

|       |  |  |                                                                                                                                                |
|-------|--|--|------------------------------------------------------------------------------------------------------------------------------------------------|
| 15367 |  |  | For the conditions under which the <code>fscanf()</code> functions fail and may fail, refer to <code>fgetc()</code> or <code>fgetwc()</code> . |
| 15368 |  |  |                                                                                                                                                |
| 15369 |  |  |                                                                                                                                                |
| 15370 |  |  | In addition, <code>fscanf()</code> may fail if:                                                                                                |

|       |    |                 |                                                      |
|-------|----|-----------------|------------------------------------------------------|
| 15371 | CX | <b>[EILSEQ]</b> | Input byte sequence does not form a valid character. |
| 15372 | CX | <b>[EINVAL]</b> | There are insufficient arguments.                    |

**EXAMPLES**

|       |  |  |                                                                                                                                                                |
|-------|--|--|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 15373 |  |  | The call:                                                                                                                                                      |
| 15374 |  |  |                                                                                                                                                                |
| 15375 |  |  | <code>int i, n; float x; char name[50];</code>                                                                                                                 |
| 15376 |  |  | <code>n = scanf("%d%f%s", &amp;i, &amp;x, name);</code>                                                                                                        |
| 15377 |  |  | with the input line:                                                                                                                                           |
| 15378 |  |  | <code>25 54.32E-1 Hamster</code>                                                                                                                               |
| 15379 |  |  | assigns to <code>n</code> the value 3, to <code>i</code> the value 25, to <code>x</code> the value 5.432, and <code>name</code> contains the string "Hamster". |
| 15380 |  |  |                                                                                                                                                                |

15381 The call:  
 15382 `int i; float x; char name[50];`  
 15383 `(void) scanf("%2d%f*d %[0123456789]", &i, &x, name);`

15384 with input:

15385 56789 0123 56a72

15386 assigns 56 to *i*, 789.0 to *x*, skips 0123, and places the string "56\0" in *name*. The next call to  
 15387 `getchar()` shall return the character 'a'.

### 15388 Reading Data into an Array

15389 The following call uses `fscanf()` to read three floating-point numbers from standard input into  
 15390 the *input* array.

15391 `float input[3]; fscanf (stdin, "%f %f %f", input, input+1, input+2);`

### 15392 APPLICATION USAGE

15393 If the application calling `fscanf()` has any objects of type `wint_t` or `wchar_t`, it must also include  
 15394 the `<wchar.h>` header to have these objects defined.

### 15395 RATIONALE

15396 This function is aligned with the ISO/IEC 9899:1999 standard, and in doing so a few "obvious"  
 15397 things were not included. Specifically, the set of characters allowed in a scanset is limited to  
 15398 single-byte characters. In other similar places, multi-byte characters have been permitted, but  
 15399 for alignment with the ISO/IEC 9899:1999 standard, it has not been done here. Applications  
 15400 needing this could use the corresponding wide-character functions to achieve the desired  
 15401 results.

### 15402 FUTURE DIRECTIONS

15403 None.

### 15404 SEE ALSO

15405 `getc()`, `printf()`, `setlocale()`, `strtod()`, `strtol()`, `strtoul()`, `wcrtomb()`, the Base Definitions volume of  
 15406 IEEE Std 1003.1-200x, Chapter 7, Locale, `<langinfo.h>`, `<stdio.h>`, `<wchar.h>`

### 15407 CHANGE HISTORY

15408 First released in Issue 1. Derived from Issue 1 of the SVID.

#### 15409 Issue 5

15410 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the `l` (ell) qualifier is  
 15411 now defined for the `c`, `s`, and `[` conversion specifiers.

15412 The DESCRIPTION is updated to indicate that if infinity and NaN can be generated by the  
 15413 `fprintf()` family of functions, then they are recognized by the `fscanf()` family.

#### 15414 Issue 6

15415 The Open Group Corrigenda U021/7 and U028/10 are applied. These correct several  
 15416 occurrences of "characters" in the text which have been replaced with the term "bytes".

15417 The normative text is updated to avoid use of the term "must" for application requirements.

15418 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 15419 • The prototypes for `fscanf()`, `scanf()`, and `sscanf()` are updated.
- 15420 • The DESCRIPTION is updated.
- 15421 • The `hh`, `ll`, `j`, `t`, and `z` length modifiers are added.



15422

- The a, A, and F conversion characters are added.

15423

15424

The DESCRIPTION is updated to use the terms “conversion specifier” and “conversion specification” consistently.

15425

**Issue 7**

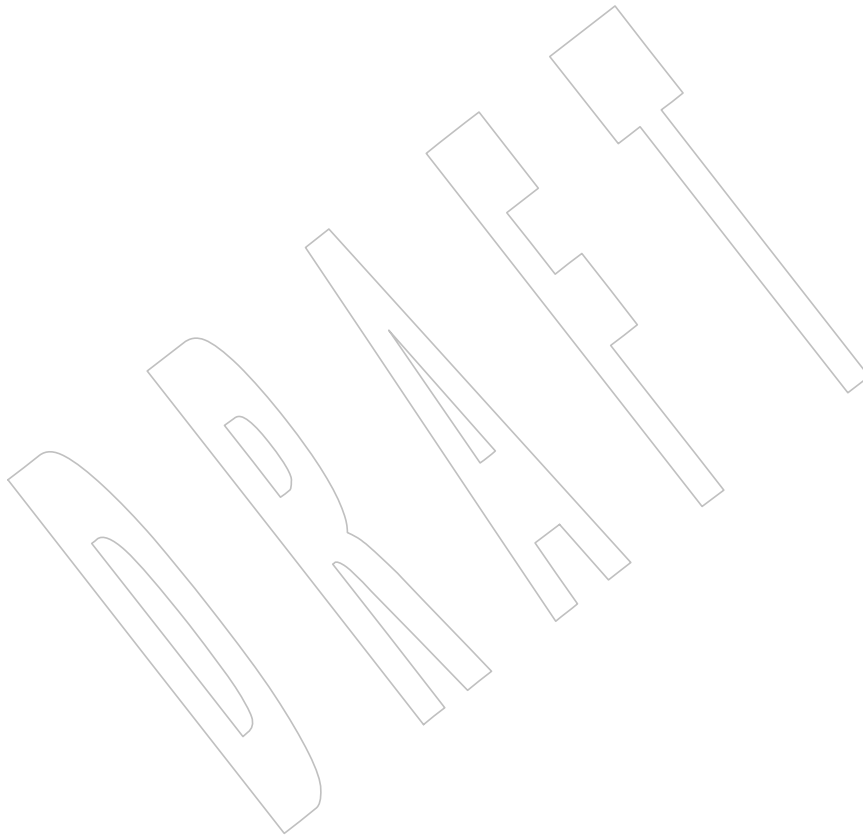
15426

XD5-XSH-ERN-9 is applied, correcting *fscanf()* to *scanf()* in the DESCRIPTION.

15427

15428

Functionality relating to the %n\$ form of conversion specification is moved from the XSI option to the Base.



15429 **NAME**15430 `fseek, fseeko` — reposition a file-position indicator in a stream15431 **SYNOPSIS**15432 `#include <stdio.h>`15433 `int fseek(FILE *stream, long offset, int whence);`15434 CX `int fseeko(FILE *stream, off_t offset, int whence);`15435 **DESCRIPTION**15436 CX The functionality described on this reference page is aligned with the ISO C standard. Any  
15437 conflict between the requirements described here and the ISO C standard is unintentional. This  
15438 volume of IEEE Std 1003.1-200x defers to the ISO C standard.15439 The `fseek()` function shall set the file-position indicator for the stream pointed to by `stream`. If a  
15440 read or write error occurs, the error indicator for the stream shall be set and `fseek()` fails.15441 The new position, measured in bytes from the beginning of the file, shall be obtained by adding  
15442 `offset` to the position specified by `whence`. The specified point is the beginning of the file for  
15443 `SEEK_SET`, the current value of the file-position indicator for `SEEK_CUR`, or end-of-file for  
15444 `SEEK_END`.15445 If the stream is to be used with wide-character input/output functions, the application shall  
15446 ensure that `offset` is either 0 or a value returned by an earlier call to `ftell()` on the same stream and  
15447 `whence` is `SEEK_SET`.15448 A successful call to `fseek()` shall clear the end-of-file indicator for the stream and undo any effects  
15449 of `ungetc()` and `ungetwc()` on the same stream. After an `fseek()` call, the next operation on an  
15450 update stream may be either input or output.15451 CX If the most recent operation, other than `ftell()`, on a given stream is `fflush()`, the file offset in the  
15452 underlying open file description shall be adjusted to reflect the location specified by `fseek()`.15453 The `fseek()` function shall allow the file-position indicator to be set beyond the end of existing  
15454 data in the file. If data is later written at this point, subsequent reads of data in the gap shall  
15455 return bytes with the value 0 until data is actually written into the gap.15456 The behavior of `fseek()` on devices which are incapable of seeking is implementation-defined.  
15457 The value of the file offset associated with such a device is undefined.15458 If the stream is writable and buffered data had not been written to the underlying file, `fseek()`  
15459 shall cause the unwritten data to be written to the file and shall mark the `st_ctime` and `st_mtime`  
15460 fields of the file for update.15461 In a locale with state-dependent encoding, whether `fseek()` restores the stream's shift state is  
15462 implementation-defined.15463 The `fseeko()` function shall be equivalent to the `fseek()` function except that the `offset` argument is  
15464 of type `off_t`.15465 **RETURN VALUE**15466 CX The `fseek()` and `fseeko()` functions shall return 0 if they succeed.15467 CX Otherwise, they shall return `-1` and set `errno` to indicate the error.



**fseek()****ERRORS**

- 15468
- 15469 CX The *fseek()* and *fseeko()* functions shall fail if, either the *stream* is unbuffered or the *stream's*
- 15470 buffer needed to be flushed, and the call to *fseek()* or *fseeko()* causes an underlying *lseek()* or
- 15471 *write()* to be invoked, and:
- 15472 CX [EAGAIN] The O\_NONBLOCK flag is set for the file descriptor and the thread would be
- 15473 delayed in the write operation.
- 15474 CX [EBADF] The file descriptor underlying the stream file is not open for writing or the
- 15475 stream's buffer needed to be flushed and the file is not open.
- 15476 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size.
- 15477 XSI [EFBIG] An attempt was made to write a file that exceeds the file size limit of the
- 15478 process.
- 15479 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
- 15480 offset maximum associated with the corresponding stream.
- 15481 CX [EINTR] The write operation was terminated due to the receipt of a signal, and no data
- 15482 was transferred.
- 15483 CX [EINVAL] The *whence* argument is invalid. The resulting file-position indicator would be
- 15484 set to a negative value.
- 15485 CX [EIO] A physical I/O error has occurred, or the process is a member of a background
- 15486 process group attempting to perform a *write()* to its controlling terminal,
- 15487 TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the
- 15488 process group of the process is orphaned. This error may also be returned
- 15489 under implementation-defined conditions.
- 15490 CX [ENOSPC] There was no free space remaining on the device containing the file.
- 15491 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
- 15492 capabilities of the device.
- 15493 CX [EOVERFLOW] For *fseek()*, the resulting file offset would be a value which cannot be
- 15494 represented correctly in an object of type **long**.
- 15495 CX [EOVERFLOW] For *fseeko()*, the resulting file offset would be a value which cannot be
- 15496 represented correctly in an object of type **off\_t**.
- 15497 CX [EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading
- 15498 by any process; a SIGPIPE signal shall also be sent to the thread.
- 15499 CX [ESPIPE] The file descriptor underlying *stream* is associated with a pipe or FIFO.

**EXAMPLES**

None.

**APPLICATION USAGE**

None.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

*open()*, *fsetpos()*, *ftell()*, *getrlimit()*, *lseek()*, *rewind()*, *ulimit()*, *ungetc()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

15511  
15512  
  
15513  
15514  
15515  
  
15516  
  
15517  
15518  
  
15519  
15520  
  
15521  
15522  
  
15523  
  
15524  
15525  
  
15526  
  
15527  
15528  
15529  
  
15530  
15531  
15532

## CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

### Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

Large File Summit extensions are added.

### Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The *fseeko()* function is added.
- The [EFBIG], [EOVERFLOW], and [ENXIO] mandatory error conditions are added.

The following change is incorporated for alignment with the FIPS requirements:

- The [EINTR] error is no longer an indication that the implementation does not report partial transfers.

The normative text is updated to avoid use of the term “must” for application requirements.

The DESCRIPTION is updated to explicitly state that *fseek()* sets the file-position indicator, and then on error the error indicator is set and *fseek()* fails. This is for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/42 is applied, updating the [EAGAIN] error in the ERRORS section from “the process would be delayed” to “the thread would be delayed”.

15533 **NAME**15534 `fsetpos` — set current file position15535 **SYNOPSIS**15536 `#include <stdio.h>`15537 `int fsetpos(FILE *stream, const fpos_t *pos);`15538 **DESCRIPTION**

15539 CX The functionality described on this reference page is aligned with the ISO C standard. Any  
 15540 conflict between the requirements described here and the ISO C standard is unintentional. This  
 15541 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

15542 The `fsetpos()` function shall set the file position and state indicators for the stream pointed to by  
 15543 `stream` according to the value of the object pointed to by `pos`, which the application shall ensure is  
 15544 a value obtained from an earlier call to `fgetpos()` on the same stream. If a read or write error  
 15545 occurs, the error indicator for the stream shall be set and `fsetpos()` fails.

15546 A successful call to the `fsetpos()` function shall clear the end-of-file indicator for the stream and  
 15547 undo any effects of `ungetc()` on the same stream. After an `fsetpos()` call, the next operation on an  
 15548 update stream may be either input or output.

15549 CX The behavior of `fsetpos()` on devices which are incapable of seeking is implementation-defined.  
 15550 The value of the file offset associated with such a device is undefined.

15551 **RETURN VALUE**

15552 The `fsetpos()` function shall return 0 if it succeeds; otherwise, it shall return a non-zero value and  
 15553 set `errno` to indicate the error.

15554 **ERRORS**

15555 CX The `fsetpos()` function shall fail if, either the `stream` is unbuffered or the `stream`'s buffer needed to  
 15556 be flushed, and the call to `fsetpos()` causes an underlying `lseek()` or `write()` to be invoked, and:

15557 CX **[EAGAIN]** The `O_NONBLOCK` flag is set for the file descriptor and the thread would be  
 15558 delayed in the write operation.

15559 CX **[EBADF]** The file descriptor underlying the stream file is not open for writing or the  
 15560 stream's buffer needed to be flushed and the file is not open.

15561 CX **[EFBIG]** An attempt was made to write a file that exceeds the maximum file size.

15562 XSI **[EFBIG]** An attempt was made to write a file that exceeds the file size limit of the  
 15563 process.

15564 CX **[EFBIG]** The file is a regular file and an attempt was made to write at or beyond the  
 15565 offset maximum associated with the corresponding stream.

15566 CX **[EINTR]** The write operation was terminated due to the receipt of a signal, and no data  
 15567 was transferred.

15568 CX **[EIO]** A physical I/O error has occurred, or the process is a member of a background  
 15569 process group attempting to perform a `write()` to its controlling terminal,  
 15570 TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the  
 15571 process group of the process is orphaned. This error may also be returned  
 15572 under implementation-defined conditions.

15573 CX **[ENOSPC]** There was no free space remaining on the device containing the file.

- 15574 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the  
15575 capabilities of the device.
- 15576 CX [EPIPE] The file descriptor underlying *stream* is associated with a pipe or FIFO.
- 15577 CX [EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading  
15578 by any process; a SIGPIPE signal shall also be sent to the thread.

**EXAMPLES**

None.

**APPLICATION USAGE**

None.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

*fopen()*, *ftell()*, *lseek()*, *rewind()*, *ungetc()*, *write()*, the Base Definitions volume of  
IEEE Std 1003.1-200x, <stdio.h>

**CHANGE HISTORY**

First released in Issue 4. Derived from the ISO C standard.

**Issue 6**

Extensions beyond the ISO C standard are marked.

An additional [ESPIPE] error condition is added for sockets.

The normative text is updated to avoid use of the term “must” for application requirements.

The DESCRIPTION is updated to clarify that the error indicator is set for the stream on a read or write error. This is for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/21 is applied, deleting an erroneous [EINVAL] error case from the ERRORS section.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/43 is applied, updating the [EAGAIN] error in the ERRORS section from “the process would be delayed” to “the thread would be delayed”.

15603 **NAME**15604 `fstat` — get file status15605 **SYNOPSIS**15606 `#include <sys/stat.h>`15607 `int fstat(int fildev, struct stat *buf);`15608 **DESCRIPTION**15609 The `fstat()` function shall obtain information about an open file associated with the file  
15610 descriptor *fildev*, and shall write it to the area pointed to by *buf*.15611 SHM If *fildev* references a shared memory object, the implementation shall update in the **stat** structure  
15612 pointed to by the *buf* argument the *st\_uid*, *st\_gid*, *st\_size*, and *st\_mode* fields, and only the  
15613 S\_IRUSR, S\_IWUSR, S\_IRGRP, S\_IWGRP, S\_IROTH, and S\_IWOTH file permission bits need be  
15614 valid. The implementation may update other fields and flags.15615 TYM If *fildev* references a typed memory object, the implementation shall update in the **stat** structure  
15616 pointed to by the *buf* argument the *st\_uid*, *st\_gid*, *st\_size*, and *st\_mode* fields, and only the  
15617 S\_IRUSR, S\_IWUSR, S\_IRGRP, S\_IWGRP, S\_IROTH, and S\_IWOTH file permission bits need be  
15618 valid. The implementation may update other fields and flags.15619 The *buf* argument is a pointer to a **stat** structure, as defined in `<sys/stat.h>`, into which  
15620 information is placed concerning the file.15621 For all other file types defined in this volume of IEEE Std 1003.1-200x, the structure members  
15622 *st\_mode*, *st\_ino*, *st\_dev*, *st\_uid*, *st\_gid*, *st\_atime*, *st\_ctime*, and *st\_mtime* shall have meaningful  
15623 values and the value of the *st\_nlink* member shall be set to the number of links to the file.15624 An implementation that provides additional or alternative file access control mechanisms may,  
15625 under implementation-defined conditions, cause `fstat()` to fail.15626 The `fstat()` function shall update any time-related fields as described in the Base Definitions  
15627 volume of IEEE Std 1003.1-200x, Section 4.7, File Times Update, before writing into the **stat**  
15628 structure.15629 **RETURN VALUE**15630 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to  
15631 indicate the error.15632 **ERRORS**15633 The `fstat()` function shall fail if:15634 [EBADF] The *fildev* argument is not a valid file descriptor.

15635 [EIO] An I/O error occurred while reading from the file system.

15636 [EOVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file  
15637 serial number cannot be represented correctly in the structure pointed to by  
15638 *buf*.15639 The `fstat()` function may fail if:15640 [EOVERFLOW] One of the values is too large to store into the structure pointed to by the *buf*  
15641 argument.

15642

**EXAMPLES**

15643

**Obtaining File Status Information**

15644

15645

15646

15647

The following example shows how to obtain file status information for a file named `/home/cnd/mod1`. The structure variable `buffer` is defined for the `stat` structure. The `/home/cnd/mod1` file is opened with read/write privileges and is passed to the open file descriptor `fildes`.

15648

```
#include <sys/types.h>
```

15649

```
#include <sys/stat.h>
```

15650

```
#include <fcntl.h>
```

15651

```
struct stat buffer;
```

15652

```
int status;
```

15653

```
...
```

15654

```
fildes = open("/home/cnd/mod1", O_RDWR);
```

15655

```
status = fstat(fildes, &buffer);
```

15656

**APPLICATION USAGE**

15657

None.

15658

**RATIONALE**

15659

None.

15660

**FUTURE DIRECTIONS**

15661

None.

15662

**SEE ALSO**

15663

[\*fstatat\(\)\*](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<sys/stat.h>`, `<sys/types.h>`

15664

**CHANGE HISTORY**

15665

First released in Issue 1. Derived from Issue 1 of the SVID.

15666

**Issue 5**

15667

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

15668

Large File Summit extensions are added.

15669

**Issue 6**

15670

In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

15671

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

15672

15673

- The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

15674

15675

15676

- The [EIO] mandatory error condition is added.

15677

- The [EOVERFLOW] mandatory error condition is added. This change is to support large files.

15678

15679

- The [EOVERFLOW] optional error condition is added.

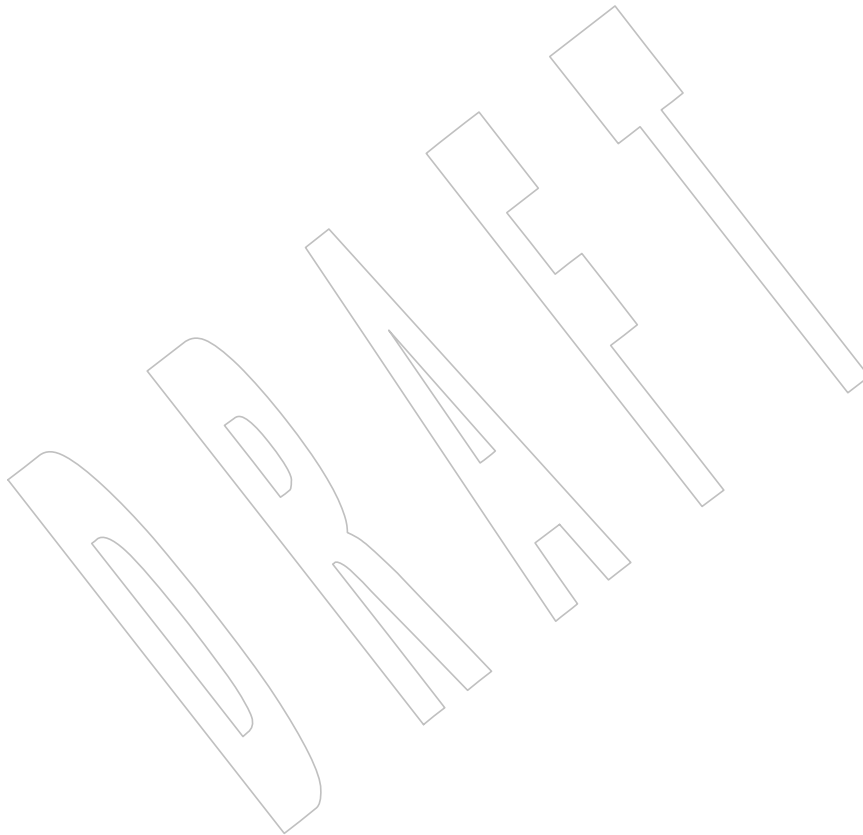
15680

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that shared memory object semantics apply to typed memory objects.

15681

15682  
15683  
15684**Issue 7**

XSH-SD5-ERN-161 is applied, updating the DESCRIPTION to clarify which file types *st\_nlink* applies to.



15685 **NAME**  
 15686 `fstatat, lstat, stat` — get file status

15687 **SYNOPSIS**  
 15688 `#include <sys/stat.h>`  
 15689 `int fstatat(int fd, const char *restrict path,`  
 15690 `struct stat *restrict buf);`  
 15691 `int lstat(const char *restrict path, struct stat *restrict buf);`  
 15692 `int stat(const char *restrict path, struct stat *restrict buf);`

15693 **DESCRIPTION**  
 15694 The `stat()` function shall obtain information about the named file and write it to the area pointed  
 15695 to by the `buf` argument. The `path` argument points to a pathname naming a file. Read, write, or  
 15696 execute permission of the named file is not required. An implementation that provides  
 15697 additional or alternate file access control mechanisms may, under implementation-defined  
 15698 conditions, cause `stat()` to fail. In particular, the system may deny the existence of the file  
 15699 specified by `path`.

15700 If the named file is a symbolic link, the `stat()` function shall continue pathname resolution using  
 15701 the contents of the symbolic link, and shall return information pertaining to the resulting file if  
 15702 the file exists.

15703 The `buf` argument is a pointer to a **stat** structure, as defined in the `<sys/stat.h>` header, into  
 15704 which information is placed concerning the file.

15705 The `stat()` function shall update any time-related fields (as described in the Base Definitions  
 15706 volume of IEEE Std 1003.1-200x, Section 4.7, File Times Update), before writing into the **stat**  
 15707 structure.

15708 SHM If the named file is a shared memory object, the implementation shall update in the **stat** structure  
 15709 pointed to by the `buf` argument the `st_uid`, `st_gid`, `st_size`, and `st_mode` fields, and only the  
 15710 `S_IRUSR`, `S_IWUSR`, `S_IRGRP`, `S_IWGRP`, `S_IROTH`, and `S_IWOTH` file permission bits need be  
 15711 valid. The implementation may update other fields and flags.

15712 TYM If the named file is a typed memory object, the implementation shall update in the **stat** structure  
 15713 pointed to by the `buf` argument the `st_uid`, `st_gid`, `st_size`, and `st_mode` fields, and only the  
 15714 `S_IRUSR`, `S_IWUSR`, `S_IRGRP`, `S_IWGRP`, `S_IROTH`, and `S_IWOTH` file permission bits need be  
 15715 valid. The implementation may update other fields and flags.

15716 For all other file types defined in this volume of IEEE Std 1003.1-200x, the structure members  
 15717 `st_mode`, `st_ino`, `st_dev`, `st_uid`, `st_gid`, `st_atime`, `st_ctime`, and `st_mtime` shall have meaningful  
 15718 values and the value of the member `st_nlink` shall be set to the number of links to the file.

15719 The `lstat()` function shall be equivalent to `stat()`, except when `path` refers to a symbolic link. In  
 15720 that case `lstat()` shall return information about the link, while `stat()` shall return information  
 15721 about the file the link references.

15722 For symbolic links, the `st_mode` member shall contain meaningful information when used with  
 15723 the file type macros, and the `st_size` member shall contain the length of the pathname contained  
 15724 in the symbolic link. File mode bits and the contents of the remaining members of the **stat**  
 15725 structure are unspecified. The value returned in the `st_size` member is the length of the contents  
 15726 of the symbolic link, and does not count any trailing null.

15727 The `fstatat()` function shall be equivalent to the `stat()` or `lstat()` function, depending on the value  
 15728 of `flag` (see below), except in the case where `path` specifies a relative path. In this case the status  
 15729 shall be retrieved from a file relative to the directory associated with the file descriptor `fd` instead  
 15730 of the current working directory. It is unspecified whether directory searches are permitted



15731 based on whether the file was opened with search permission or on the current permissions of  
 15732 the directory underlying the file descriptor.

15733 Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined  
 15734 in **<fcntl.h>**:

15735 AT\_SYMLINK\_NOFOLLOW

15736 If *path* names a symbolic link, the status of the symbolic link is returned.

15737 If *fstatat()* is passed the special value AT\_FDCWD in the *fd* parameter, the current working  
 15738 directory is used and the behavior shall be identical to a call to *stat()* or *lstat()* respectively,  
 15739 depending on whether or not the AT\_SYMLINK\_NOFOLLOW bit is set in *flag*.

#### 15740 RETURN VALUE

15741 Upon successful completion, these functions shall return 0. Otherwise, these functions shall  
 15742 return  $-1$  and set *errno* to indicate the error.

#### 15743 ERRORS

15744 These functions shall fail if:

15745 [EACCES] Search permission is denied for a component of the path prefix.

15746 [EIO] An error occurred while reading from the file system.

15747 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*  
 15748 argument.

15749 [ENAMETOOLONG]

15750 The length of the *path* argument exceeds {PATH\_MAX} or a pathname  
 15751 component is longer than {NAME\_MAX}.

15752 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

15753 [ENOTDIR] A component of the path prefix is not a directory.

15754 [EOVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file  
 15755 serial number cannot be represented correctly in the structure pointed to by  
 15756 *buf*.

15757 The *fstatat()* function shall fail if:

15758 [EBADF] The *path* argument does not specify an absolute path and the *fd* argument is  
 15759 neither AT\_FDCWD nor a valid file descriptor open for searching.

15760 These functions may fail if:

15761 [ELOOP] More than {SYMLOOP\_MAX} symbolic links were encountered during  
 15762 resolution of the *path* argument.

15763 [ENAMETOOLONG]

15764 As a result of encountering a symbolic link in resolution of the *path* argument,  
 15765 the length of the substituted pathname string exceeded {PATH\_MAX}.

15766 [EOVERFLOW] A value to be stored would overflow one of the members of the **stat** structure.

15767 The *fstatat()* function may fail if:

15768 [EINVAL] The value of the *flag* argument is not valid.

15769 [ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT\_FDCWD nor a  
 15770 file descriptor associated with a directory.

## EXAMPLES

**Obtaining File Status Information**

The following example shows how to obtain file status information for a file named `/home/cnd/mod1`. The structure variable `buffer` is defined for the `stat` structure.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

struct stat buffer;
int status;
...
status = stat("/home/cnd/mod1", &buffer);
```

**Getting Directory Information**

The following example fragment gets status information for each entry in a directory. The call to the `stat()` function stores file information in the `stat` structure pointed to by `statbuf`. The lines that follow the `stat()` call format the fields in the `stat` structure for presentation to the user of the program.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
#include <locale.h>
#include <langinfo.h>
#include <stdio.h>
#include <stdint.h>

struct dirent *dp;
struct stat statbuf;
struct passwd *pwd;
struct group *grp;
struct tm *tm;
char datestring[256];
...
/* Loop through directory entries. */
while ((dp = readdir(dir)) != NULL) {
```

```

15818 if ((grp = getgrgid(statbuf.st_gid)) != NULL)
15819 printf(" %-8.8s", grp->gr_name);
15820 else
15821 printf(" %-8d", statbuf.st_gid);
15822
15822 /* Print size of file. */
15823 printf(" %9jd", (intmax_t)statbuf.st_size);
15824
15824 tm = localtime(&statbuf.st_mtime);
15825
15825 /* Get localized date string. */
15826 strftime(datestring, sizeof(datestring), nl_langinfo(D_T_FMT), tm);
15827 printf(" %s %s\n", datestring, dp->d_name);
15828 }

```

### 15829 Obtaining Symbolic Link Status Information

15830 The following example shows how to obtain status information for a symbolic link named  
 15831 **/modules/pass1**. The structure variable *buffer* is defined for the **stat** structure. If the *path*  
 15832 argument specified the filename for the file pointed to by the symbolic link (**/home/cnd/mod1**),  
 15833 the results of calling the function would be the same as those returned by a call to the *stat()*  
 15834 function.

```

15835 #include <sys/stat.h>
15836
15836 struct stat buffer;
15837 int status;
15838 ...
15839 status = lstat("/modules/pass1", &buffer);

```

### 15840 APPLICATION USAGE

15841 None.

### 15842 RATIONALE

15843 The intent of the paragraph describing “additional or alternate file access control mechanisms”  
 15844 is to allow a secure implementation where a process with a label that does not dominate the  
 15845 file’s label cannot perform a *stat()* function. This is not related to read permission; a process with  
 15846 a label that dominates the file’s label does not need read permission. An implementation that  
 15847 supports write-up operations could fail *fstat()* function calls even though it has a valid file  
 15848 descriptor open for writing.

15849 The *lstat()* function is not required to update the time-related fields if the named file is not a  
 15850 symbolic link. While the *st\_uid*, *st\_gid*, *st\_atime*, *st\_mtime*, and *st\_ctime* members of the **stat**  
 15851 structure may apply to a symbolic link, they are not required to do so. No functions in  
 15852 IEEE Std 1003.1-200x are required to maintain any of these time fields.

15853 The purpose of the *fstatat()* function is to obtain the status of files in directories other than the  
 15854 current working directory without exposure to race conditions. Any part of the path of a file  
 15855 could be changed in parallel to a call to *stat()*, resulting in unspecified behavior. By opening a  
 15856 file descriptor for the target directory and using the *fstatat()* function it can be guaranteed that  
 15857 the file for which status is returned is located relative to the desired directory.

### 15858 FUTURE DIRECTIONS

15859 None.

15860  
15861  
15862  
15863  
15864  
15865  
15866  
15867  
15868  
15869  
15870  
15871  
15872  
15873  
15874  
15875  
15876  
15877  
15878  
15879  
15880  
15881  
15882  
15883  
15884  
15885  
15886  
15887  
15888  
15889  
15890

**SEE ALSO**

*access()*, *chmod()*, *fchmoddir()*, *fstat()*, *mknod()*, *readlink()*, *symlink()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<fcntl.h>`, `<sys/stat.h>`, `<sys/types.h>`

**CHANGE HISTORY**

First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 5**

Large File Summit extensions are added.

**Issue 6**

In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The [EIO] mandatory error condition is added.
- The [ELOOP] mandatory error condition is added.
- The [E\_OVERFLOW] mandatory error condition is added. This change is to support large files.
- The [ENAMETOOLONG] and the second [E\_OVERFLOW] optional error conditions are added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Details are added regarding the treatment of symbolic links.
- The [ELOOP] optional error condition is added.

The normative text is updated to avoid use of the term “must” for application requirements.

The **restrict** keyword is added to the *stat()* prototype for alignment with the ISO/IEC 9899:1999 standard.

**Issue 7**

The *fstatat()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

XSH-SD5-ERN-161 is applied, updating the DESCRIPTION to clarify which file types *st\_nlink* applies to.

15891 **NAME**15892 `fstatvfs, statvfs` — get file system information15893 **SYNOPSIS**15894 `#include <sys/statvfs.h>`15895 `int fstatvfs(int fildev, struct statvfs *buf);`15896 `int statvfs(const char *restrict path, struct statvfs *restrict buf);`15897 **DESCRIPTION**15898 The `fstatvfs()` function shall obtain information about the file system containing the file  
15899 referenced by *fildev*.15900 The `statvfs()` function shall obtain information about the file system containing the file named by  
15901 *path*.15902 For both functions, the *buf* argument is a pointer to a **statvfs** structure that shall be filled. Read,  
15903 write, or execute permission of the named file is not required.15904 The following flags can be returned in the *f\_flag* member:15905 **ST\_RDONLY** Read-only file system.15906 **ST\_NOSUID** Setuid/setgid bits ignored by *exec*.15907 It is unspecified whether all members of the **statvfs** structure have meaningful values on all file  
15908 systems.15909 **RETURN VALUE**15910 Upon successful completion, `statvfs()` shall return 0. Otherwise, it shall return `-1` and set *errno* to  
15911 indicate the error.15912 **ERRORS**15913 The `fstatvfs()` and `statvfs()` functions shall fail if:15914 **[EIO]** An I/O error occurred while reading the file system.15915 **[EINTR]** A signal was caught during execution of the function.15916 **[EOVERFLOW]** One of the values to be returned cannot be represented correctly in the  
15917 structure pointed to by *buf*.15918 The `fstatvfs()` function shall fail if:15919 **[EBADF]** The *fildev* argument is not an open file descriptor.15920 The `statvfs()` function shall fail if:15921 **[EACCES]** Search permission is denied on a component of the path prefix.15922 **[ELOOP]** A loop exists in symbolic links encountered during resolution of the *path*  
15923 argument.15924 **[ENAMETOOLONG]**15925 The length of a pathname exceeds `{PATH_MAX}` or a pathname component is  
15926 longer than `{NAME_MAX}`.15927 **[ENOENT]** A component of *path* does not name an existing file or *path* is an empty string.15928 **[ENOTDIR]** A component of the path prefix of *path* is not a directory.

15929 The *statvfs()* function may fail if:

15930 [ELOOP] More than {SYMLOOP\_MAX} symbolic links were encountered during  
15931 resolution of the *path* argument.

15932 [ENAMETOOLONG]  
15933 Pathname resolution of a symbolic link produced an intermediate result  
15934 whose length exceeds {PATH\_MAX}.

**EXAMPLES****Obtaining File System Information Using *fstatvfs()***

15936 The following example shows how to obtain file system information for the file system upon  
15937 which the file named */home/cnd/mod1* resides, using the *fstatvfs()* function. The  
15938 */home/cnd/mod1* file is opened with read/write privileges and the open file descriptor is passed  
15939 to the *fstatvfs()* function.  
15940

```
15941 #include <sys/statvfs.h>
15942 #include <fcntl.h>
15943
15944 struct statvfs buffer;
15945 int status;
15946 ...
15947 fildes = open("/home/cnd/mod1", O_RDWR);
15948 status = fstatvfs(fildes, &buffer);
```

**Obtaining File System Information Using *statvfs()***

15949 The following example shows how to obtain file system information for the file system upon  
15950 which the file named */home/cnd/mod1* resides, using the *statvfs()* function.

```
15951 #include <sys/statvfs.h>
15952
15953 struct statvfs buffer;
15954 int status;
15955 ...
15956 status = statvfs("/home/cnd/mod1", &buffer);
```

**APPLICATION USAGE**

15957 None.

**RATIONALE**

15958 None.

**FUTURE DIRECTIONS**

15960 None.

**SEE ALSO**

15962 *chmod()*, *chown()*, *creat()*, *dup()*, *exec*, *fcntl()*, *link()*, *mknod()*, *open()*, *pipe()*, *read()*, *time()*,  
15963 *unlink()*, *utime()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/statvfs.h>**

**CHANGE HISTORY**

15965 First released in Issue 4, Version 2.

**Issue 5**

15967 Moved from X/OPEN UNIX extension to BASE.

15969 Large File Summit extensions are added.

**fstatvfs()**

15970

**Issue 6**

15971

The normative text is updated to avoid use of the term “must” for application requirements.

15972

15973

The **restrict** keyword is added to the *statvfs()* prototype for alignment with the ISO/IEC 9899:1999 standard.

15974

15975

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

15976

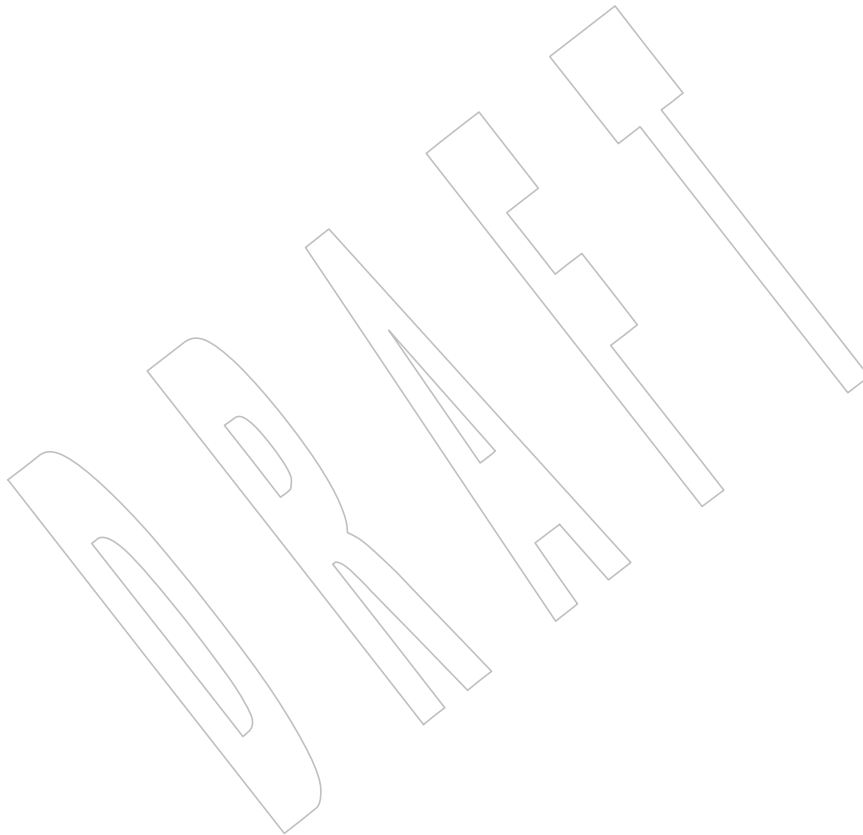
**Issue 7**

15977

The *fstatvfs()* and *statvfs()* functions are moved from the XSI option to the Base.

15978

SD5-XSH-ERN-68 is applied, correcting the EXAMPLES section.



15979 **NAME**

15980 fsync — synchronize changes to a file

15981 **SYNOPSIS**

```
15982 FSC #include <unistd.h>
15983 int fsync(int fildes);
```

15984 **DESCRIPTION**

15985 The *fsync()* function shall request that all data for the open file descriptor named by *fil*des is to be  
 15986 transferred to the storage device associated with the file described by *fil*des. The nature of the  
 15987 transfer is implementation-defined. The *fsync()* function shall not return until the system has  
 15988 completed that action or until an error is detected.

15989 SIO If `_POSIX_SYNCHRONIZED_IO` is defined, the *fsync()* function shall force all currently queued  
 15990 I/O operations associated with the file indicated by file descriptor *fil*des to the synchronized I/O  
 15991 completion state. All I/O operations shall be completed as defined for synchronized I/O file  
 15992 integrity completion.

15993 **RETURN VALUE**

15994 Upon successful completion, *fsync()* shall return 0. Otherwise, `-1` shall be returned and *errno* set  
 15995 to indicate the error. If the *fsync()* function fails, outstanding I/O operations are not guaranteed  
 15996 to have been completed.

15997 **ERRORS**15998 The *fsync()* function shall fail if:

- |       |          |                                                                                           |
|-------|----------|-------------------------------------------------------------------------------------------|
| 15999 | [EBADF]  | The <i>fil</i> des argument is not a valid descriptor.                                    |
| 16000 | [EINTR]  | The <i>fsync()</i> function was interrupted by a signal.                                  |
| 16001 | [EINVAL] | The <i>fil</i> des argument does not refer to a file on which this operation is possible. |
| 16002 | [EIO]    | An I/O error occurred while reading from or writing to the file system.                   |

16003 In the event that any of the queued I/O operations fail, *fsync()* shall return the error conditions  
 16004 defined for *read()* and *write()*.

16005 **EXAMPLES**

16006 None.

16007 **APPLICATION USAGE**

16008 The *fsync()* function should be used by programs which require modifications to a file to be  
 16009 completed before continuing; for example, a program which contains a simple transaction  
 16010 facility might use it to ensure that all modifications to a file or files caused by a transaction are  
 16011 recorded.

16012 **RATIONALE**

16013 The *fsync()* function is intended to force a physical write of data from the buffer cache, and to  
 16014 assure that after a system crash or other failure that all data up to the time of the *fsync()* call is  
 16015 recorded on the disk. Since the concepts of “buffer cache”, “system crash”, “physical write”, and  
 16016 “non-volatile storage” are not defined here, the wording has to be more abstract.

16017 If `_POSIX_SYNCHRONIZED_IO` is not defined, the wording relies heavily on the conformance  
 16018 document to tell the user what can be expected from the system. It is explicitly intended that a  
 16019 null implementation is permitted. This could be valid in the case where the system cannot assure  
 16020 non-volatile storage under any circumstances or when the system is highly fault-tolerant and the  
 16021 functionality is not required. In the middle ground between these extremes, *fsync()* might or



16022 might not actually cause data to be written where it is safe from a power failure. The  
 16023 conformance document should identify at least that one configuration exists (and how to obtain  
 16024 that configuration) where this can be assured for at least some files that the user can select to use  
 16025 for critical data. It is not intended that an exhaustive list is required, but rather sufficient  
 16026 information is provided so that if critical data needs to be saved, the user can determine how the  
 16027 system is to be configured to allow the data to be written to non-volatile storage.

16028 It is reasonable to assert that the key aspects of *fsync()* are unreasonable to test in a test suite.  
 16029 That does not make the function any less valuable, just more difficult to test. A formal  
 16030 conformance test should probably force a system crash (power shutdown) during the test for  
 16031 this condition, but it needs to be done in such a way that automated testing does not require this  
 16032 to be done except when a formal record of the results is being made. It would also not be  
 16033 unreasonable to omit testing for *fsync()*, allowing it to be treated as a quality-of-implementation  
 16034 issue.

#### 16035 **FUTURE DIRECTIONS**

16036 None.

#### 16037 **SEE ALSO**

16038 *sync()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**unistd.h**>

#### 16039 **CHANGE HISTORY**

16040 First released in Issue 3.

#### 16041 **Issue 5**

16042 Aligned with *fsync()* in the POSIX Realtime Extension. Specifically, the DESCRIPTION and  
 16043 RETURN VALUE sections are much expanded, and the ERRORS section is updated to indicate  
 16044 that *fsync()* can return the error conditions defined for *read()* and *write()*.

#### 16045 **Issue 6**

16046 This function is marked as part of the File Synchronization option.

16047 The following new requirements on POSIX implementations derive from alignment with the  
 16048 Single UNIX Specification:

- The [EINVAL] and [EIO] mandatory error conditions are added.

16050 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/44 is applied, applying an editorial  
 16051 rewording of the DESCRIPTION. No change in meaning is intended.

**NAME**

ftell, ftello — return a file offset in a stream

**SYNOPSIS**

```
#include <stdio.h>

long ftell(FILE *stream);
CX off_t ftello(FILE *stream);
```

**DESCRIPTION**

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *ftell()* function shall obtain the current value of the file-position indicator for the stream pointed to by *stream*.

CX The *ftello()* function shall be equivalent to *ftell()*, except that the return value is of type **off\_t**.

**RETURN VALUE**

CX Upon successful completion, *ftell()* and *ftello()* shall return the current value of the file-position

16090  
16091  
16092  
16093  
16094  
16095  
16096  
16097  
16098  
16099  
16100

**CHANGE HISTORY**

First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 5**

Large File Summit extensions are added.

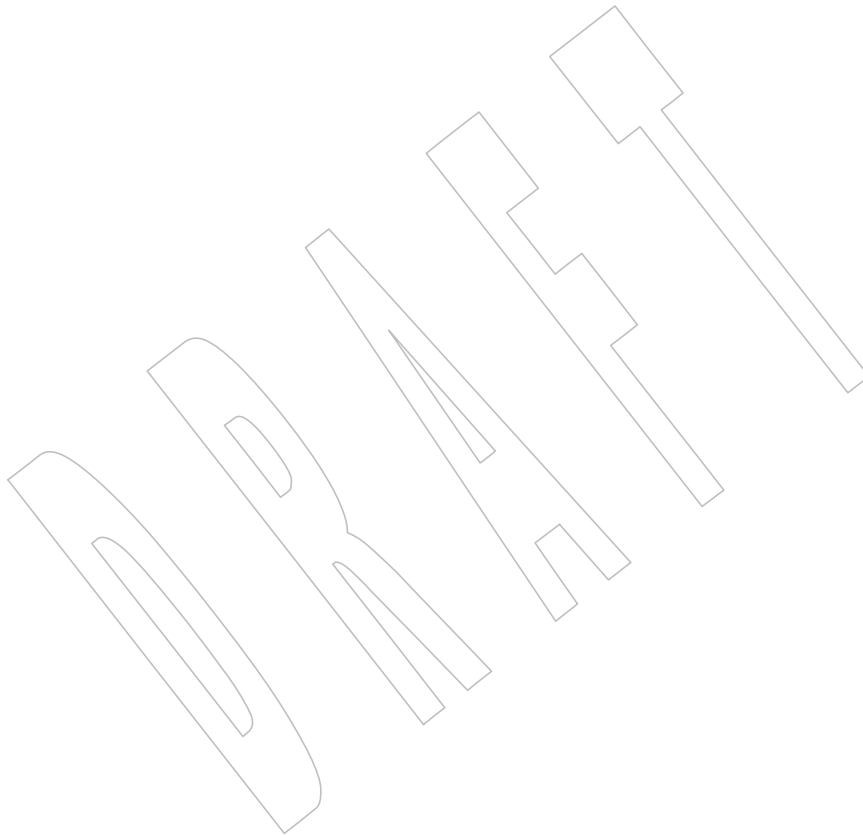
**Issue 6**

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The *ftello()* function is added.
- The [Eoverflow] error conditions are added.

An additional [ESPIPE] error condition is added for sockets.



16101 **NAME**

16102 ftok — generate an IPC key

16103 **SYNOPSIS**

```
16104 xSI #include <sys/ipc.h>
16105 key_t ftok(const char *path, int id);
```

16106 **DESCRIPTION**

16107 The *ftok()* function shall return a key based on *path* and *id* that is usable in subsequent calls to  
 16108 *msgget()*, *semget()*, and *shmget()*. The application shall ensure that the *path* argument is the  
 16109 pathname of an existing file that the process is able to *stat()*.

16110 The *ftok()* function shall return the same key value for all paths that name the same file, when  
 16111 called with the same *id* value, and return different key values when called with different *id*  
 16112 values or with paths that name different files existing on the same file system at the same time. It  
 16113 is unspecified whether *ftok()* shall return the same key value when called again after the file  
 16114 named by *path* is removed and recreated with the same name.

16115 Only the low-order 8-bits of *id* are significant. The behavior of *ftok()* is unspecified if these bits  
 16116 are 0.

16117 **RETURN VALUE**

16118 Upon successful completion, *ftok()* shall return a key. Otherwise, *ftok()* shall return (**key\_t**)-1  
 16119 and set *errno* to indicate the error.

16120 **ERRORS**

16121 The *ftok()* function shall fail if:

- |       |                |                                                                                              |
|-------|----------------|----------------------------------------------------------------------------------------------|
| 16122 | [EACCES]       | Search permission is denied for a component of the path prefix.                              |
| 16123 | [ELOOP]        | A loop exists in symbolic links encountered during resolution of the <i>path</i>             |
| 16124 |                | argument.                                                                                    |
| 16125 | [ENAMETOOLONG] |                                                                                              |
| 16126 |                | The length of the <i>path</i> argument exceeds {PATH_MAX} or a pathname                      |
| 16127 |                | component is longer than {NAME_MAX}.                                                         |
| 16128 | [ENOENT]       | A component of <i>path</i> does not name an existing file or <i>path</i> is an empty string. |
| 16129 | [ENOTDIR]      | A component of the path prefix is not a directory.                                           |
| 16130 |                | The <i>ftok()</i> function may fail if:                                                      |
| 16131 | [ELOOP]        | More than {SYMLOOP_MAX} symbolic links were encountered during                               |
| 16132 |                | resolution of the <i>path</i> argument.                                                      |
| 16133 | [ENAMETOOLONG] |                                                                                              |
| 16134 |                | Pathname resolution of a symbolic link produced an intermediate result                       |
| 16135 |                | whose length exceeds {PATH_MAX}.                                                             |

**EXAMPLES****Getting an IPC Key**

The following example gets a unique key that can be used by the IPC functions *semget()*, *msgget()*, and *shmget()*. The key returned by *ftok()* for this example is based on the ID value *S* and the pathname **/tmp**.

```
16141 #include <sys/ipc.h>
16142 ...
16143 key_t key;
16144 char *path = "/tmp";
16145 int id = 'S';
16146
16147 key = ftok(path, id);
```

**Saving an IPC Key**

The following example gets a unique key based on the pathname **/tmp** and the ID value *a*. It also assigns the value of the resulting key to the *semkey* variable so that it will be available to a later call to *semget()*, *msgget()*, or *shmget()*.

```
16151 #include <sys/ipc.h>
16152 ...
16153 key_t semkey;
16154 if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
16155 perror("IPC error: ftok"); exit(1);
16156 }
```

**APPLICATION USAGE**

For maximum portability, *id* should be a single-byte character.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

*msgget()*, *semget()*, *shmget()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/ipc.h>**

**CHANGE HISTORY**

First released in Issue 4, Version 2.

**Issue 5**

Moved from X/OPEN UNIX extension to BASE.

**Issue 6**

The normative text is updated to avoid use of the term “must” for application requirements.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

16173 **NAME**16174 `ftruncate` — truncate a file to a specified length16175 **SYNOPSIS**16176 `#include <unistd.h>`16177 `int ftruncate(int fildes, off_t length);`16178 **DESCRIPTION**16179 If *filde*s is not a valid file descriptor open for writing, the *ftruncate()* function shall fail.

16180 If *filde*s refers to a regular file, the *ftruncate()* function shall cause the size of the file to be  
 16181 truncated to *length*. If the size of the file previously exceeded *length*, the extra data shall no  
 16182 longer be available to reads on the file. If the file previously was smaller than this size,  
 16183 *ftruncate()* shall increase the size of the file. If the file size is increased, the extended area shall  
 16184 appear as if it were zero-filled. The value of the seek pointer shall not be modified by a call to  
 16185 *ftruncate()*.

16186 Upon successful completion, if *filde*s refers to a regular file, the *ftruncate()* function shall mark  
 16187 for update the *st\_ctime* and *st\_mtime* fields of the file and the S\_ISUID and S\_ISGID bits of the  
 16188 file mode may be cleared. If the *ftruncate()* function is unsuccessful, the file is unaffected.

16189 XSI If the request would cause the file size to exceed the soft file size limit for the process, the  
 16190 request shall fail and the implementation shall generate the SIGXFSZ signal for the thread.

16191 If *filde*s refers to a directory, *ftruncate()* shall fail.16192 If *filde*s refers to any other file type, except a shared memory object, the result is unspecified.

16193 SHM If *filde*s refers to a shared memory object, *ftruncate()* shall set the size of the shared memory  
 16194 object to *length*.

16195 SHM If the effect of *ftruncate()* is to decrease the size of a memory mapped file or a shared memory  
 16196 object and whole pages beyond the new end were previously mapped, then the whole pages  
 16197 beyond the new end shall be discarded.

16198 References to discarded pages shall result in the generation of a SIGBUS signal.

16199 If the effect of *ftruncate()* is to increase the size of a memory object, it is unspecified whether the  
 16200 contents of any mapped pages between the old end-of-file and the new are flushed to the  
 16201 underlying object.

16202 **RETURN VALUE**

16203 Upon successful completion, *ftruncate()* shall return 0; otherwise,  $-1$  shall be returned and *errno*  
 16204 set to indicate the error.

16205 **ERRORS**16206 The *ftruncate()* function shall fail if:

16207 [EINTR] A signal was caught during execution.

16208 [EINVAL] The *length* argument was less than 0.

16209 [EFBIG] or [EINVAL]

16210 The *length* argument was greater than the maximum file size.

16211 [EFBIG] The file is a regular file and *length* is greater than the offset maximum  
 16212 established in the open file description associated with *filde*s.

**ftruncate()**

16213 [EIO] An I/O error occurred while reading from or writing to a file system.  
 16214 [EBADF] or [EINVAL]  
 16215 The *filides* argument is not a file descriptor open for writing.

**EXAMPLES**

16216 None.  
 16217

**APPLICATION USAGE**

16218 None.  
 16219

**RATIONALE**

16220 None.  
 16221

**FUTURE DIRECTIONS**

16222 None.  
 16223

**SEE ALSO**

16224 *open()*, *truncate()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**unistd.h**>  
 16225

**CHANGE HISTORY**

16226 First released in Issue 4, Version 2.  
 16227

**Issue 5**

16228 Moved from X/OPEN UNIX extension to BASE and aligned with *ftruncate()* in the POSIX  
 16229 Realtime Extension. Specifically, the DESCRIPTION is extensively reworded and [EROFS] is  
 16230 added to the list of mandatory errors that can be returned by *ftruncate()*.  
 16231

16232 Large File Summit extensions are added.

**Issue 6**

16233 The *truncate()* function is split out into a separate reference page.  
 16234

16235 The following new requirements on POSIX implementations derive from alignment with the  
 16236 Single UNIX Specification:

- The DESCRIPTION is changed to indicate that if the file size is changed, and if the file is a regular file, the S\_ISUID and S\_ISGID bits in the file mode may be cleared.

16237 The following changes were made to align with the IEEE P1003.1a draft standard:  
 16238

- The DESCRIPTION text is updated.

16239 XSI-conformant systems are required to increase the size of the file if the file was previously  
 16240 smaller than the size requested.  
 16241

**Issue 7**

16242 Austin Group Interpretation 1003.1-2001 #056 is applied, revising the ERRORS section (although  
 16243 the [EINVAL] “may fail” error was subsequently removed during review of the XSI option).  
 16244

16245 Functionality relating to the Memory Protection and Memory Mapped Files options is moved to  
 16246 the Base.  
 16247

16248 The DESCRIPTION is updated so that a call to *ftruncate()* when the file is smaller than the size  
 16249 requested will increase the size of the file. Previously, non-XSI-conforming implementations  
 16250 were allowed to increase the size of the file or fail.

16251 **NAME**  
16252 `ftrylockfile` — stdio locking functions

16253 **SYNOPSIS**

16254 CX `#include <stdio.h>`  
16255 `int ftrylockfile(FILE *file);`

16256 **DESCRIPTION**

16257 Refer to [flockfile\(\)](#).



**NAME**

ftw — traverse (walk) a file tree

**SYNOPSIS**

```
OB XSI #include <ftw.h>

int ftw(const char *path, int (*fn)(const char *,
 const struct stat *ptr, int flag), int ndirs);
```

**DESCRIPTION**

The *ftw()* function shall recursively descend the directory hierarchy rooted in *path*. For each object in the hierarchy, *ftw()* shall call the function pointed to by *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure containing information about the object, filled in as if *stat()* or *lstat()* had been called to retrieve the information. Possible values of the integer, defined in the **<ftw.h>** header, are:

FTW\_D For a directory.

FTW\_DNR For a directory that cannot be read.

FTW\_F For a file.

FTW\_SL For a symbolic link (but see also FTW\_NS below).

FTW\_NS For an object other than a symbolic link on which *stat()* could not successfully be executed. If the object is a symbolic link and *stat()* failed, it is unspecified whether *ftw()* passes FTW\_SL or FTW\_NS to the user-supplied function.

If the integer is FTW\_DNR, descendants of that directory shall not be processed. If the integer is FTW\_NS, the **stat** structure contains undefined values. An example of an object that would cause FTW\_NS to be passed to the function pointed to by *fn* would be a file in a directory with read but without execute (search) permission.

The *ftw()* function shall visit a directory before visiting any of its descendants.

The *ftw()* function shall use at most one file descriptor for each level in the tree.

The argument *ndirs* should be in the range [1,{OPEN\_MAX}].

The tree traversal shall continue until either the tree is exhausted, an invocation of *fn* returns a non-zero value, or some error, other than [EACCES], is detected within *ftw()*.

The *ndirs* argument shall specify the maximum number of directory streams or file descriptors or both available for use by *ftw()* while traversing the tree. When *ftw()* returns it shall close any directory streams and file descriptors it uses not counting any opened by the application-supplied *fn* function.

The results are unspecified if the application-supplied *fn* function does not preserve the current working directory.

The *ftw()* function need not be thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

**RETURN VALUE**

If the tree is exhausted, *ftw()* shall return 0. If the function pointed to by *fn* returns a non-zero value, *ftw()* shall stop its tree traversal and return whatever value was returned by the function pointed to by *fn*. If *ftw()* detects an error, it shall return -1 and set *errno* to indicate the error.

If *ftw()* encounters an error other than [EACCES] (see FTW\_DNR and FTW\_NS above), it shall return -1 and set *errno* to indicate the error. The external variable *errno* may contain any error

16300 value that is possible when a directory is opened or when one of the *stat* functions is executed on  
 16301 a directory or file.

**ERRORS**

16302 The *ftw()* function shall fail if:

16304 [EACCES] Search permission is denied for any component of *path* or read permission is  
 16305 denied for *path*.

16306 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*  
 16307 argument.

16308 [ENAMETOOLONG]  
 16309 The length of the *path* argument exceeds {PATH\_MAX} or a pathname  
 16310 component is longer than {NAME\_MAX}.

16311 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

16312 [ENOTDIR] A component of *path* is not a directory.

16313 [EOVERFLOW] A field in the **stat** structure cannot be represented correctly in the current  
 16314 programming environment for one or more files found in the file hierarchy.

16315 The *ftw()* function may fail if:

16316 [EINVAL] The value of the *ndirs* argument is invalid.

16317 [ELOOP] More than {SYMLOOP\_MAX} symbolic links were encountered during  
 16318 resolution of the *path* argument.

16319 [ENAMETOOLONG]  
 16320 Pathname resolution of a symbolic link produced an intermediate result  
 16321 whose length exceeds {PATH\_MAX}.

16322 In addition, if the function pointed to by *fn* encounters system errors, *errno* may be set  
 16323 accordingly.

**EXAMPLES****Walking a Directory Structure**

16325 The following example walks the current directory structure, calling the *fn* function for every  
 16326 directory entry, using at most 10 file descriptors:

```
16328 #include <ftw.h>
16329 ...
16330 if (ftw(".", fn, 10) != 0) {
16331 perror("ftw"); exit(2);
16332 }
```

**APPLICATION USAGE**

16334 The *ftw()* function may allocate dynamic storage during its operation. If *ftw()* is forcibly  
 16335 terminated, such as by *longjmp()* or *siglongjmp()* being executed by the function pointed to by *fn*  
 16336 or an interrupt routine, *ftw()* does not have a chance to free that storage, so it remains  
 16337 permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has  
 16338 occurred, and arrange to have the function pointed to by *fn* return a non-zero value at its next  
 16339 invocation.

16340 Applications should use the *nftw()* function instead of the obsolescent *ftw()* function.

16341

**RATIONALE**

16342

None.

16343

**FUTURE DIRECTIONS**

16344

The *ftw()* function may be removed in a future version.

16345

**SEE ALSO**

16346

*fdopendir()*, *fstatat()*, *longjmp()*, *malloc()*, *nftw()*, *siglongjmp()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<ftw.h>**, **<sys/stat.h>**

16347

16348

**CHANGE HISTORY**

16349

First released in Issue 1. Derived from Issue 1 of the SVID.

16350

**Issue 5**

16351

UX codings in the DESCRIPTION, RETURN VALUE, and ERRORS sections are changed to EX.

16352

**Issue 6**

16353

The ERRORS section is updated as follows:

16354

- The wording of the mandatory [ELOOP] error condition is updated.

16355

- A second optional [ELOOP] error condition is added.

16356

- The [EOVERFLOW] mandatory error condition is added.

16357

A note is added to the DESCRIPTION indicating that this function need not be reentrant, and that the results are unspecified if the application-supplied *fn* function does not preserve the current working directory.

16358

16359

16360

**Issue 7**

16361

SD5-XBD-ERN-61 is applied.

16362

The *ftw()* function is marked obsolescent.

16363 **NAME**  
16364 funlockfile — stdio locking functions

16365 **SYNOPSIS**

```
16366 CX #include <stdio.h>
16367 void funlockfile(FILE *file);
```

16368 **DESCRIPTION**

16369 Refer to *flockfile()*.

**futimesat()**

16370 **NAME**  
16371 futimesat — set file access and modification times relative to directory file descriptor

**SYNOPSIS**

16372 XSI `#include <sys/time.h>`  
16373 `int futimesat(int fd, const char *path, const struct timeval times[2]);`  
16374

**DESCRIPTION**

16375 Refer to *utimes()*.  
16376

16377 **NAME**16378 `fwide` — set stream orientation16379 **SYNOPSIS**16380 `#include <stdio.h>`16381 `#include <wchar.h>`16382 `int fwide(FILE *stream, int mode);`16383 **DESCRIPTION**

16384 CX The functionality described on this reference page is aligned with the ISO C standard. Any  
 16385 conflict between the requirements described here and the ISO C standard is unintentional. This  
 16386 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

16387 The `fwide()` function shall determine the orientation of the stream pointed to by `stream`. If `mode` is  
 16388 greater than zero, the function first attempts to make the stream wide-oriented. If `mode` is less  
 16389 than zero, the function first attempts to make the stream byte-oriented. Otherwise, `mode` is zero  
 16390 and the function does not alter the orientation of the stream.

16391 If the orientation of the stream has already been determined, `fwide()` shall not change it.

16392 CX Since no return value is reserved to indicate an error, an application wishing to check for error  
 16393 situations should set `errno` to 0, then call `fwide()`, then check `errno`, and if it is non-zero, assume  
 16394 an error has occurred.

16395 **RETURN VALUE**

16396 The `fwide()` function shall return a value greater than zero if, after the call, the stream has wide-  
 16397 orientation, a value less than zero if the stream has byte-orientation, or zero if the stream has no  
 16398 orientation.

16399 **ERRORS**16400 The `fwide()` function may fail if:

16401 CX [EBADF] The `stream` argument is not a valid stream.

16402 **EXAMPLES**

16403 None.

16404 **APPLICATION USAGE**

16405 A call to `fwide()` with `mode` set to zero can be used to determine the current orientation of a  
 16406 stream.

16407 **RATIONALE**

16408 None.

16409 **FUTURE DIRECTIONS**

16410 None.

16411 **SEE ALSO**16412 The Base Definitions volume of IEEE Std 1003.1-200x, `<wchar.h>`16413 **CHANGE HISTORY**

16414 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995  
 16415 (E).

16416 **Issue 6**

16417 Extensions beyond the ISO C standard are marked.

16418 **NAME**

16419 fwprintf, swprintf, wprintf — print formatted wide-character output

16420 **SYNOPSIS**

16421 #include &lt;stdio.h&gt;

16422 #include &lt;wchar.h&gt;

16423 int fwprintf(FILE \*restrict stream, const wchar\_t \*restrict format, ...);

16424 int swprintf(wchar\_t \*restrict ws, size\_t n,

16425 const wchar\_t \*restrict format, ...);

16426 int wprintf(const wchar\_t \*restrict format, ...);

16427 **DESCRIPTION**

16428 CX The functionality described on this reference page is aligned with the ISO C standard. Any  
 16429 conflict between the requirements described here and the ISO C standard is unintentional. This  
 16430 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

16431 The *fwprintf()* function shall place output on the named output *stream*. The *wprintf()* function  
 16432 shall place output on the standard output stream *stdout*. The *swprintf()* function shall place  
 16433 output followed by the null wide character in consecutive wide characters starting at *\*ws*; no  
 16434 more than *n* wide characters shall be written, including a terminating null wide character, which  
 16435 is always added (unless *n* is zero).

16436 Each of these functions shall convert, format, and print its arguments under control of the *format*  
 16437 wide-character string. The *format* is composed of zero or more directives: *ordinary wide-characters*,  
 16438 which are simply copied to the output stream, and *conversion specifications*, each of which results  
 16439 in the fetching of zero or more arguments. The results are undefined if there are insufficient  
 16440 arguments for the *format*. If the *format* is exhausted while arguments remain, the excess  
 16441 arguments are evaluated but are otherwise ignored.

16442 CX Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than  
 16443 to the next unused argument. In this case, the conversion specifier wide character % (see below)  
 16444 is replaced by the sequence "%n\$", where *n* is a decimal integer in the range  
 16445 [1,{NL\_ARGMAX}], giving the position of the argument in the argument list. This feature  
 16446 provides for the definition of *format* wide-character strings that select arguments in an order  
 16447 appropriate to specific languages (see the EXAMPLES section).

16448 The *format* can contain either numbered argument specifications (that is, "%n\$" and "\*m\$"), or  
 16449 unnumbered argument conversion specifications (that is, % and \*), but not both. The only  
 16450 exception to this is that %% can be mixed with the "%n\$" form. The results of mixing numbered  
 16451 and unnumbered argument specifications in a *format* wide-character string are undefined. When  
 16452 numbered argument specifications are used, specifying the *N*th argument requires that all the  
 16453 leading arguments, from the first to the (*N*-1)th, are specified in the *format* wide-character string.

16454 In *format* wide-character strings containing the "%n\$" form of conversion specification,  
 16455 numbered arguments in the argument list can be referenced from the *format* wide-character  
 16456 string as many times as required.

16457 In *format* wide-character strings containing the % form of conversion specification, each  
 16458 argument in the argument list shall be used exactly once.

16459 CX All forms of the *fwprintf()* function allow for the insertion of a locale-dependent radix character  
 16460 in the output string, output as a wide-character value. The radix character is defined in the  
 16461 locale of the process (category *LC\_NUMERIC*). In the POSIX locale, or in a locale where the  
 16462 radix character is not defined, the radix character shall default to a period ('.').

- 16463 CX Each conversion specification is introduced by the '`%`' wide character or by the wide-character  
16464 sequence "`%n$`", after which the following appear in sequence:
- 16465 • Zero or more *flags* (in any order), which modify the meaning of the conversion  
16466 specification.
  - 16467 • An optional minimum *field width*. If the converted value has fewer wide characters than  
16468 the field width, it shall be padded with spaces by default on the left; it shall be padded on  
16469 the right, if the left-adjustment flag ('`-`'), described below, is given to the field width. The  
16470 field width takes the form of an asterisk ('`*`'), described below, or a decimal integer.
  - 16471 • An optional *precision* that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`,  
16472 `x`, and `X` conversion specifiers; the number of digits to appear after the radix character for  
16473 the `a`, `A`, `e`, `E`, `f`, and `F` conversion specifiers; the maximum number of significant digits for  
16474 the `g` and `G` conversion specifiers; or the maximum number of wide characters to be  
16475 printed from a string in the `s` conversion specifiers. The precision takes the form of a  
16476 period ('`.`') followed either by an asterisk ('`*`'), described below, or an optional decimal  
16477 digit string, where a null digit string is treated as 0. If a precision appears with any other  
16478 conversion wide character, the behavior is undefined.
  - 16479 • An optional length modifier that specifies the size of the argument.
  - 16480 • A *conversion specifier* wide character that indicates the type of conversion to be applied.
- 16481 A field width, or precision, or both, may be indicated by an asterisk ('`*`'). In this case an  
16482 argument of type `int` supplies the field width or precision. Applications shall ensure that  
16483 arguments specifying field width, or precision, or both appear in that order before the argument,  
16484 if any, to be converted. A negative field width is taken as a '`-`' flag followed by a positive field  
16485 width. A negative precision is taken as if the precision were omitted. In *format* wide-character  
16486 strings containing the "`%n$`" form of a conversion specification, a field width or precision may  
16487 be indicated by the sequence "`*m$`", where `m` is a decimal integer in the range  
16488 `[1, {NL_ARGMAX}]` giving the position in the argument list (after the *format* argument) of an  
16489 integer argument containing the field width or precision, for example:
- ```
16490 wprintf(L"%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```
- 16491 The flag wide characters and their meanings are:
- 16492 CX '`'` The integer portion of the result of a decimal conversion (`%i`, `%d`, `%u`, `%f`, `%F`, `%g`, or `%G`)
16493 shall be formatted with thousands' grouping wide characters. For other conversions,
16494 the behavior is undefined. The numeric grouping wide character is used.
 - 16495 `-` The result of the conversion shall be left-justified within the field. The conversion shall
16496 be right-justified if this flag is not specified.
 - 16497 `+` The result of a signed conversion shall always begin with a sign ('`+`' or '`-`'). The
16498 conversion shall begin with a sign only when a negative value is converted if this flag is
16499 not specified.
 - 16500 `<space>` If the first wide character of a signed conversion is not a sign, or if a signed conversion
16501 results in no wide characters, a `<space>` shall be prefixed to the result. This means that
16502 if the `<space>` and '`+`' flags both appear, the `<space>` flag shall be ignored.
 - 16503 `#` Specifies that the value is to be converted to an alternative form. For `o` conversion, it
16504 increases the precision (if necessary) to force the first digit of the result to be 0. For `x` or
16505 `X` conversion specifiers, a non-zero result shall have `0x` (or `0X`) prefixed to it. For `a`, `A`, `e`,
16506 `E`, `f`, `F`, `g`, and `G` conversion specifiers, the result shall always contain a radix character,
16507 even if no digits follow it. Without this flag, a radix character appears in the result of
16508 these conversions only if a digit follows it. For `g` and `G` conversion specifiers, trailing
16509 zeros shall *not* be removed from the result as they normally are. For other conversion
16510 specifiers, the behavior is undefined.

fwprintf()

16511 0 For **d**, **i**, **o**, **u**, **x**, **X**, **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversion specifiers, leading zeros
 16512 (following any indication of sign or base) are used to pad to the field width; no space
 16513 padding is performed. If the **'0'** and **'-'** flags both appear, the **'0'** flag shall be
 16514 ignored. For **d**, **i**, **o**, **u**, **x**, and **X** conversion specifiers, if a precision is specified, the **'0'**
 16515 **CX** flag shall be ignored. If the **'0'** and **''** flags both appear, the grouping wide
 16516 characters are inserted before zero padding. For other conversions, the behavior is
 16517 undefined.

16518 The length modifiers and their meanings are:

16519 **hh** Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **signed char**
 16520 or **unsigned char** argument (the argument will have been promoted according to the
 16521 integer promotions, but its value shall be converted to **signed char** or **unsigned char**
 16522 before printing); or that a following **n** conversion specifier applies to a pointer to a
 16523 **signed char** argument.

16524 **h** Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **short** or
 16525 **unsigned short** argument (the argument will have been promoted according to the
 16526 integer promotions, but its value shall be converted to **short** or **unsigned short** before
 16527 printing); or that a following **n** conversion specifier applies to a pointer to a **short**
 16528 argument.

16529 **l (ell)** Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **long** or
 16530 **unsigned long** argument; that a following **n** conversion specifier applies to a pointer to
 16531 a **long** argument; that a following **c** conversion specifier applies to a **wint_t** argument;
 16532 that a following **s** conversion specifier applies to a pointer to a **wchar_t** argument; or
 16533 has no effect on a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier.

16534 **ll (ell-ell)**
 16535 Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **long long**
 16536 or **unsigned long long** argument; or that a following **n** conversion specifier applies to a
 16537 pointer to a **long long** argument.

16538 **j** Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to an **intmax_t**
 16539 or **uintmax_t** argument; or that a following **n** conversion specifier applies to a pointer
 16540 to an **intmax_t** argument.

16541 **z** Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **size_t** or the
 16542 corresponding signed integer type argument; or that a following **n** conversion specifier
 16543 applies to a pointer to a signed integer type corresponding to a **size_t** argument.

16544 **t** Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **ptrdiff_t** or
 16545 the corresponding **unsigned** type argument; or that a following **n** conversion specifier
 16546 applies to a pointer to a **ptrdiff_t** argument.

16547 **L** Specifies that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to a **long**
 16548 **double** argument.

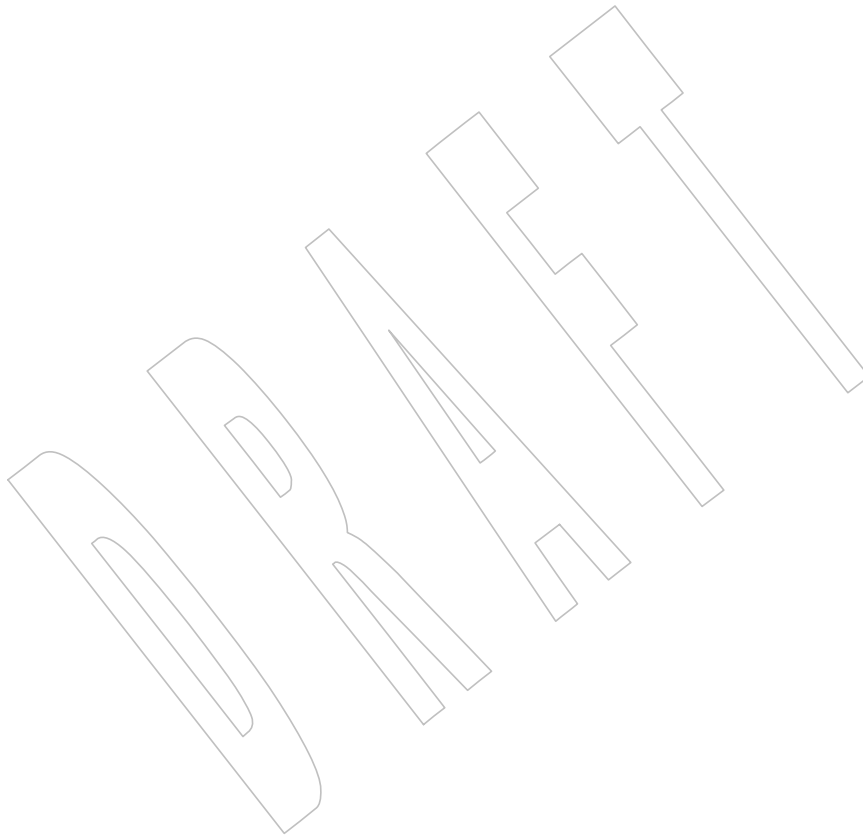
16549 If a length modifier appears with any conversion specifier other than as specified above, the
 16550 behavior is undefined.

16551 The conversion specifiers and their meanings are:

16552 **d, i** The **int** argument shall be converted to a signed decimal in the style "**[-]dddd**". The
 16553 precision specifies the minimum number of digits to appear; if the value being
 16554 converted can be represented in fewer digits, it shall be expanded with leading zeros.
 16555 The default precision shall be 1. The result of converting zero with an explicit precision
 16556 of zero shall be no wide characters.

- 16557 o The **unsigned** argument shall be converted to unsigned octal format in the style
16558 "dddd". The precision specifies the minimum number of digits to appear; if the value
16559 being converted can be represented in fewer digits, it shall be expanded with leading
16560 zeros. The default precision shall be 1. The result of converting zero with an explicit
16561 precision of zero shall be no wide characters.
- 16562 u The **unsigned** argument shall be converted to unsigned decimal format in the style
16563 "dddd". The precision specifies the minimum number of digits to appear; if the value
16564 being converted can be represented in fewer digits, it shall be expanded with leading
16565 zeros. The default precision shall be 1. The result of converting zero with an explicit
16566 precision of zero shall be no wide characters.
- 16567 x The **unsigned** argument shall be converted to unsigned hexadecimal format in the style
16568 "dddd"; the letters "abcdef" are used. The precision specifies the minimum number
16569 of digits to appear; if the value being converted can be represented in fewer digits, it
16570 shall be expanded with leading zeros. The default precision shall be 1. The result of
16571 converting zero with an explicit precision of zero shall be no wide characters.
- 16572 X Equivalent to the x conversion specifier, except that letters "ABCDEF" are used instead
16573 of "abcdef".
- 16574 f, F The **double** argument shall be converted to decimal notation in the style
16575 "[-]ddd.ddd", where the number of digits after the radix character shall be equal to
16576 the precision specification. If the precision is missing, it shall be taken as 6; if the
16577 precision is explicitly zero and no '#' flag is present, no radix character shall appear. If
16578 a radix character appears, at least one digit shall appear before it. The value shall be
16579 rounded in an implementation-defined manner to the appropriate number of digits.
- 16580 A **double** argument representing an infinity shall be converted in one of the styles
16581 "[-]inf" or "[-]infinity"; which style is implementation-defined. A **double**
16582 argument representing a NaN shall be converted in one of the styles "[-]nan" or
16583 "[-]nan(*n-char-sequence*)"; which style, and the meaning of any *n-char-sequence*,
16584 is implementation-defined. The F conversion specifier produces "INF", "INFINITY",
16585 or "NAN" instead of "inf", "infinity", or "nan", respectively.
- 16586 e, E The **double** argument shall be converted in the style "[-]d.ddde±dd", where there
16587 shall be one digit before the radix character (which is non-zero if the argument is non-
16588 zero) and the number of digits after it shall be equal to the precision; if the precision is
16589 missing, it shall be taken as 6; if the precision is zero and no '#' flag is present, no
16590 radix character shall appear. The value shall be rounded in an implementation-defined
16591 manner to the appropriate number of digits. The E conversion wide character shall
16592 produce a number with 'E' instead of 'e' introducing the exponent. The exponent
16593 shall always contain at least two digits. If the value is zero, the exponent shall be zero.
- 16594 A **double** argument representing an infinity or NaN shall be converted in the style of
16595 an f or F conversion specifier.
- 16596 g, G The **double** argument shall be converted in the style f or e (or in the style F or E in the
16597 case of a G conversion specifier), with the precision specifying the number of significant
16598 digits. If an explicit precision is zero, it shall be taken as 1. The style used depends on
16599 the value converted; style e (or E) shall be used only if the exponent resulting from
16600 such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros
16601 shall be removed from the fractional portion of the result; a radix character shall appear
16602 only if it is followed by a digit.
- 16603 A **double** argument representing an infinity or NaN shall be converted in the style of
16604 an f or F conversion specifier.

- a, A A **double** argument representing a floating-point number shall be converted in the style "[−]0xh.hhhhp±d", where there shall be one hexadecimal digit (which is non-zero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point wide character and the number of hexadecimal digits after it shall be equal to the precision; if the precision is missing and FLT_RADIX is a power of 2, then the precision shall be sufficient for an exact representation of the value; if the precision is missing and FLT_RADIX is not a power of 2, then the precision shall be sufficient to distinguish values of type **double**, except that trailing zeros may be omitted; if the precision is zero and the '#' flag is not specified, no decimal-point wide character shall appear. The letters "abcdef" are used for a conversion and the letters "ABCDEF" for A



16654 result. Characters generated by *fwprintf()* and *wprintf()* shall be printed as if *fputc()* had been
16655 called.

16656 For a and A conversions, if FLT_RADIX is not a power of 2 and the result is not exactly
16657 representable in the given precision, the result should be one of the two adjacent numbers in
16658 hexadecimal floating style with the given precision, with the extra stipulation that the error
16659 should have a correct sign for the current rounding direction.

16660 For e, E, f, F, g, and G conversion specifiers, if the number of significant decimal digits is at
16661 most DECIMAL_DIG, then the result should be correctly rounded. If the number of significant
16662 decimal digits is more than DECIMAL_DIG but the source value is exactly representable with
16663 DECIMAL_DIG digits, then the result should be an exact representation with trailing zeros.
16664 Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having
16665 DECIMAL_DIG significant digits; the value of the resultant decimal string D should satisfy $L \leq$
16666 $D \leq U$, with the extra stipulation that the error should have a correct sign for the current
16667 rounding direction.

16668 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the call to a
16669 successful execution of *fwprintf()* or *wprintf()* and the next successful completion of a call to
16670 *fflush()* or *fclose()* on the same stream, or a call to *exit()* or *abort()*.

16671 RETURN VALUE

16672 Upon successful completion, these functions shall return the number of wide characters
16673 transmitted, excluding the terminating null wide character in the case of *swprintf()*, or a negative
16674 value if an output error was encountered, and set *errno* to indicate the error.

16675 If n or more wide characters were requested to be written, *swprintf()* shall return a negative
16676 value, and set *errno* to indicate the error.

16677 ERRORS

16678 For the conditions under which *fwprintf()* and *wprintf()* fail and may fail, refer to *fputc()*.

16679 In addition, all forms of *fwprintf()* may fail if:

16680 CX [EILSEQ] A wide-character code that does not correspond to a valid character has been
16681 detected.

16682 CX [EINVAL] There are insufficient arguments.

16683 In addition, *fwprintf()* and *wprintf()* may fail if:

16684 CX [ENOMEM] Insufficient storage space is available.

16685 The *swprintf()* shall fail if:

16686 CX [EOVERFLOW] The value of n is greater than {INT_MAX} or the number of bytes needed to
16687 hold the output excluding the terminating null is greater than {INT_MAX}.

16688 EXAMPLES

16689 To print the language-independent date and time format, the following statement could be used:

16690 `wprintf(format, weekday, month, day, hour, min);`

16691 For American usage, *format* could be a pointer to the wide-character string:

16692 `L"%s, %s %d, %d:%.2d\n"`

16693 producing the message:

16694 Sunday, July 3, 10:02

16695 whereas for German usage, *format* could be a pointer to the wide-character string:

16696 `L"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"`

16697 producing the message:

16698 Sonntag, 3. Juli, 10:02

16699 APPLICATION USAGE

16700 None.

16701 RATIONALE

16702 None.

16703 FUTURE DIRECTIONS

16704 None.

16705 SEE ALSO

16706 *btowc()*, *fputwc()*, *fwscanf()*, *mbrtowc()*, *setlocale()*, the Base Definitions volume of
16707 IEEE Std 1003.1-200x, Chapter 7, Locale, `<stdio.h>`, `<wchar.h>`

16708 CHANGE HISTORY

16709 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
16710 (E).

16711 Issue 6

16712 The Open Group Corrigendum U040/1 is applied to the RETURN VALUE section, describing
16713 the case if *n* or more wide characters are requested to be written using *swprintf()*.

16714 The normative text is updated to avoid use of the term “must” for application requirements.

16715 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 16716 • The prototypes for *fwprintf()*, *swprintf()*, and *wprintf()* are updated.
- 16717 • The DESCRIPTION is updated.
- 16718 • The hh, ll, j, t, and z length modifiers are added.
- 16719 • The a, A, and F conversion characters are added.
- 16720 • XSI shading is removed from the description of character string representations of infinity
16721 and NaN floating-point values.

16722 The DESCRIPTION is updated to use the terms “conversion specifier” and “conversion
16723 specification” consistently.

16724 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

16725 Issue 7

16726 Functionality relating to the “%n\$” form of conversion specification and the ‘ ’ (apostrophe)
16727 flag is moved from the XSI option to the Base.

16728 The [Eoverflow] error is added for *swprintf()*.

16729 **NAME**

16730 fwrite — binary output

16731 **SYNOPSIS**

16732 #include <stdio.h>

16733 size_t fwrite(const void *restrict ptr, size_t size, size_t nitems,
16734 FILE *restrict stream);16735 **DESCRIPTION**16736 CX The functionality described on this reference page is aligned with the ISO C standard. Any
16737 conflict between the requirements described here and the ISO C standard is unintentional. This
16738 volume of IEEE Std 1003.1-200x defers to the ISO C standard.16739 The *fwrite()* function shall write, from the array pointed to by *ptr*, up to *nitems* elements whose
16740 size is specified by *size*, to the stream pointed to by *stream*. For each object, *size* calls shall be
16741 made to the *fputc()* function, taking the values (in order) from an array of **unsigned char** exactly
16742 overlaying the object. The file-position indicator for the stream (if defined) shall be advanced by
16743 the number of bytes successfully written. If an error occurs, the resulting value of the file-
16744 position indicator for the stream is unspecified.16745 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
16746 execution of *fwrite()* and the next successful completion of a call to *fflush()* or *fclose()* on the
16747 same stream, or a call to *exit()* or *abort()*.16748 **RETURN VALUE**16749 The *fwrite()* function shall return the number of elements successfully written, which may be
16750 less than *nitems* if a write error is encountered. If *size* or *nitems* is 0, *fwrite()* shall return 0 and the
16751 state of the stream remains unchanged. Otherwise, if a write error occurs, the error indicator for
16752 CX the stream shall be set, and *errno* shall be set to indicate the error.16753 **ERRORS**16754 Refer to *fputc()*.16755 **EXAMPLES**

16756 None.

16757 **APPLICATION USAGE**16758 Because of possible differences in element length and byte ordering, files written using *fwrite()*
16759 are application-dependent, and possibly cannot be read using *fread()* by a different application
16760 or by the same application on a different processor.16761 **RATIONALE**

16762 None.

16763 **FUTURE DIRECTIONS**

16764 None.

16765 **SEE ALSO**16766 *ferror()*, *fopen()*, *printf()*, *putc()*, *puts()*, *write()*, the Base Definitions volume of
16767 IEEE Std 1003.1-200x, <stdio.h>16768 **CHANGE HISTORY**

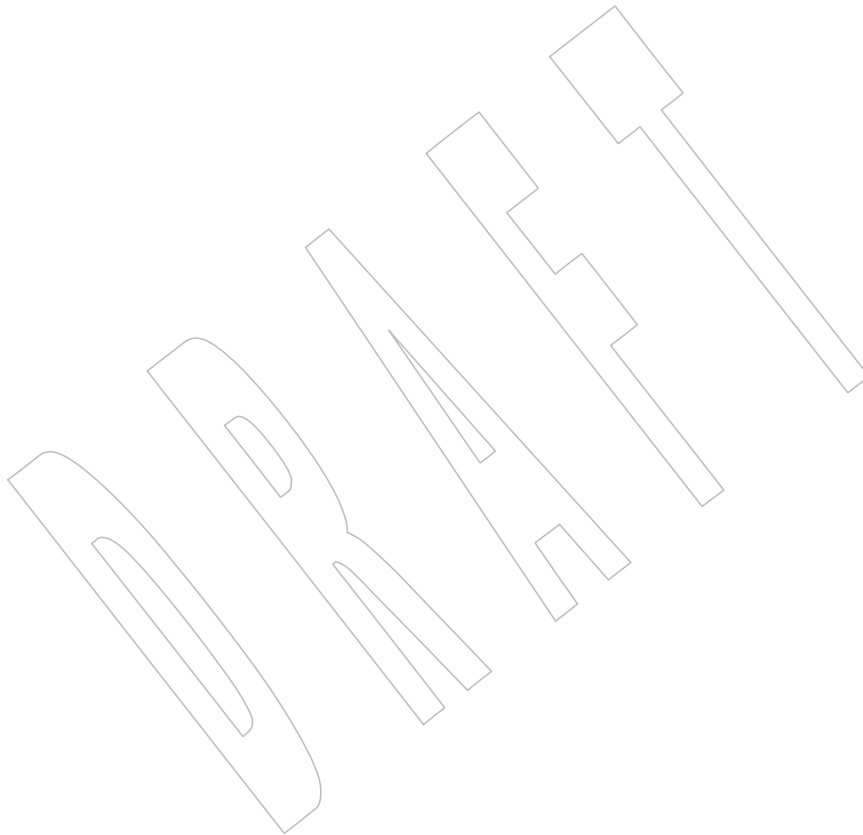
16769 First released in Issue 1. Derived from Issue 1 of the SVID.

16770
16771
16772
16773
16774**Issue 6**

Extensions beyond the ISO C standard are marked.

The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

- The *fwrite()* prototype is updated.
- The DESCRIPTION is updated to clarify how the data is written out using *fputc()*.



16775 **NAME**
 16776 fwscanf, swscanf, wscanf — convert formatted wide-character input

16777 **SYNOPSIS**

```
16778 #include <stdio.h>
16779 #include <wchar.h>

16780 int fwscanf(FILE *restrict stream, const wchar_t *restrict format, ... );
16781 int swscanf(const wchar_t *restrict ws,
16782             const wchar_t *restrict format, ... );
16783 int wscanf(const wchar_t *restrict format, ... );
```

16784 **DESCRIPTION**

16785 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 16786 conflict between the requirements described here and the ISO C standard is unintentional. This
 16787 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

16788 The *fwscanf()* function shall read from the named input *stream*. The *wscanf()* function shall read
 16789 from the standard input stream *stdin*. The *swscanf()* function shall read from the wide-character
 16790 string *ws*. Each function reads wide characters, interprets them according to a format, and stores
 16791 the results in its arguments. Each expects, as arguments, a control wide-character string *format*
 16792 described below, and a set of *pointer* arguments indicating where the converted input should be
 16793 stored. The result is undefined if there are insufficient arguments for the format. If the *format* is
 16794 exhausted while arguments remain, the excess arguments are evaluated but are otherwise
 16795 ignored.

16796 CX Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
 16797 to the next unused argument. In this case, the conversion specifier wide character % (see below)
 16798 is replaced by the sequence "%n\$", where *n* is a decimal integer in the range
 16799 [1,{NL_ARGMAX}]. This feature provides for the definition of *format* wide-character strings that
 16800 select arguments in an order appropriate to specific languages. In *format* wide-character strings
 16801 containing the "%n\$" form of conversion specifications, it is unspecified whether numbered
 16802 arguments in the argument list can be referenced from the *format* wide-character string more
 16803 than once.

16804 The *format* can contain either form of a conversion specification—that is, % or "%n\$"—but the
 16805 two forms cannot normally be mixed within a single *format* wide-character string. The only
 16806 exception to this is that %% or %* can be mixed with the "%n\$" form. When numbered argument
 16807 specifications are used, specifying the *N*th argument requires that all the leading arguments,
 16808 from the first to the (*N*−1)th, are pointers.

16809 The *fwscanf()* function in all its forms allows for detection of a language-dependent radix
 16810 character in the input string, encoded as a wide-character value. The radix character is defined
 16811 in the locale of the process (category *LC_NUMERIC*). In the POSIX locale, or in a locale where
 16812 the radix character is not defined, the radix character shall default to a period ('.').

16813 The *format* is a wide-character string composed of zero or more directives. Each directive is
 16814 composed of one of the following: one or more white-space wide characters (<space>*s*, <tab>*s*,
 16815 <newline>*s*, <vertical-tab>*s*, or <form-feed>*s*); an ordinary wide character (neither '%' nor a
 16816 white-space character); or a conversion specification.

16817 CX Each conversion specification is introduced by the '%' or by the character sequence "%n\$",
 16818 after which the following appear in sequence:

- 16819 • An optional assignment-suppressing character '*'.

- 16820 • An optional non-zero decimal integer that specifies the maximum field width.
- 16821 • An optional length modifier that specifies the size of the receiving object.
- 16822 • A conversion specifier wide character that specifies the type of conversion to be applied.
- 16823 The valid conversion specifiers are described below.

16824 The *fwscanf()* functions shall execute each directive of the format in turn. If a directive fails, as
 16825 detailed below, the function shall return. Failures are described as input failures (due to the
 16826 unavailability of input bytes) or matching failures (due to inappropriate input).

16827 A directive composed of one or more white-space wide characters is executed by reading input
 16828 until no more valid input can be read, or up to the first wide character which is not a white-
 16829 space wide character, which remains unread.

16830 A directive that is an ordinary wide character shall be executed as follows. The next wide
 16831 character is read from the input and compared with the wide character that comprises the
 16832 directive; if the comparison shows that they are not equivalent, the directive shall fail, and the
 16833 differing and subsequent wide characters remain unread. Similarly, if end-of-file, an encoding
 16834 error, or a read error prevents a wide character from being read, the directive shall fail.

16835 A directive that is a conversion specification defines a set of matching input sequences, as
 16836 described below for each conversion wide character. A conversion specification is executed in
 16837 the following steps.

16838 Input white-space wide characters (as specified by *iswspace()*) shall be skipped, unless the
 16839 conversion specification includes a *l*, *c*, or *n* conversion specifier.

16840 An item shall be read from the input, unless the conversion specification includes an *n*
 16841 conversion specifier wide character. An input item is defined as the longest sequence of input
 16842 wide characters, not exceeding any specified field width, which is an initial subsequence of a
 16843 matching sequence. The first wide character, if any, after the input item shall remain unread. If
 16844 the length of the input item is zero, the execution of the conversion specification shall fail; this
 16845 condition is a matching failure, unless end-of-file, an encoding error, or a read error prevented
 16846 input from the stream, in which case it is an input failure.

16847 Except in the case of a *%* conversion specifier, the input item (or, in the case of a *%n* conversion
 16848 specification, the count of input wide characters) shall be converted to a type appropriate to the
 16849 conversion wide character. If the input item is not a matching sequence, the execution of the
 16850 conversion specification shall fail; this condition is a matching failure. Unless assignment
 16851 suppression was indicated by a *'*'*, the result of the conversion shall be placed in the object
 16852 pointed to by the first argument following the *format* argument that has not already received a
 16853 conversion result if the conversion specification is introduced by *%*, or in the *n*th argument if
 16854 introduced by the wide-character sequence *"%n\$"*. If this object does not have an appropriate
 16855 type, or if the result of the conversion cannot be represented in the space provided, the behavior
 16856 is undefined.

CX

16857 The length modifiers and their meanings are:

- 16858 hh Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an
 16859 argument with type pointer to **signed char** or **unsigned char**.
- 16860 h Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an
 16861 argument with type pointer to **short** or **unsigned short**.
- 16862 l (ell) Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an
 16863 argument with type pointer to **long** or **unsigned long**; that a following *a*, *A*, *e*, *E*, *f*, *F*,
 16864 *g*, or *G* conversion specifier applies to an argument with type pointer to **double**; or that
 16865 a following *c*, *s*, or *l* conversion specifier applies to an argument with type pointer to
 16866 **wchar_t**.

16867	l1 (ell-ell)	
16868		Specifies that a following <i>d</i> , <i>i</i> , <i>o</i> , <i>u</i> , <i>x</i> , <i>X</i> , or <i>n</i> conversion specifier applies to an
16869		argument with type pointer to long long or unsigned long long .
16870	<i>j</i>	Specifies that a following <i>d</i> , <i>i</i> , <i>o</i> , <i>u</i> , <i>x</i> , <i>X</i> , or <i>n</i> conversion specifier applies to an
16871		argument with type pointer to intmax_t or uintmax_t .
16872	<i>z</i>	Specifies that a following <i>d</i> , <i>i</i> , <i>o</i> , <i>u</i> , <i>x</i> , <i>X</i> , or <i>n</i> conversion specifier applies to an
16873		argument with type pointer to size_t or the corresponding signed integer type.
16874	<i>t</i>	Specifies that a following <i>d</i> , <i>i</i> , <i>o</i> , <i>u</i> , <i>x</i> , <i>X</i> , or <i>n</i> conversion specifier applies to an
16875		argument with type pointer to ptrdiff_t or the corresponding unsigned type.
16876	<i>L</i>	Specifies that a following <i>a</i> , <i>A</i> , <i>e</i> , <i>E</i> , <i>f</i> , <i>F</i> , <i>g</i> , or <i>G</i> conversion specifier applies to an
16877		argument with type pointer to long double .
16878	If a length modifier appears with any conversion specifier other than as specified above, the	
16879	behavior is undefined.	
16880	The following conversion specifier wide characters are valid:	
16881	<i>d</i>	Matches an optionally signed decimal integer, whose format is the same as expected for
16882		the subject sequence of <i>wcstol()</i> with the value 10 for the <i>base</i> argument. In the absence
16883		of a size modifier, the application shall ensure that the corresponding argument is a
16884		pointer to int .
16885	<i>i</i>	Matches an optionally signed integer, whose format is the same as expected for the
16886		subject sequence of <i>wcstol()</i> with 0 for the <i>base</i> argument. In the absence of a size
16887		modifier, the application shall ensure that the corresponding argument is a pointer to
16888		int .
16889	<i>o</i>	Matches an optionally signed octal integer, whose format is the same as expected for
16890		the subject sequence of <i>wcstoul()</i> with the value 8 for the <i>base</i> argument. In the absence
16891		of a size modifier, the application shall ensure that the corresponding argument is a
16892		pointer to unsigned .
16893	<i>u</i>	Matches an optionally signed decimal integer, whose format is the same as expected for
16894		the subject sequence of <i>wcstoul()</i> with the value 10 for the <i>base</i> argument. In the absence
16895		of a size modifier, the application shall ensure that the corresponding argument is a
16896		pointer to unsigned .
16897	<i>x</i>	Matches an optionally signed hexadecimal integer, whose format is the same as
16898		expected for the subject sequence of <i>wcstoul()</i> with the value 16 for the <i>base</i> argument.
16899		In the absence of a size modifier, the application shall ensure that the corresponding
16900		argument is a pointer to unsigned .
16901	<i>a, e, f, g</i>	
16902		Matches an optionally signed floating-point number, infinity, or NaN whose format is
16903		the same as expected for the subject sequence of <i>wcstod()</i> . In the absence of a size
16904		modifier, the application shall ensure that the corresponding argument is a pointer to
16905		float .
16906		If the <i>fwprintf()</i> family of functions generates character string representations for
16907		infinity and NaN (a symbolic entity encoded in floating-point format) to support
16908		IEEE Std 754-1985, the <i>fwscanf()</i> family of functions shall recognize them as input.
16909	<i>s</i>	Matches a sequence of non white-space wide characters. If no <i>l</i> (ell) qualifier is present,
16910		characters from the input field shall be converted as if by repeated calls to the
16911		<i>wcrtomb()</i> function, with the conversion state described by an mbstate_t object
16912		initialized to zero before the first wide character is converted. The application shall
16913		ensure that the corresponding argument is a pointer to a character array large enough

- 16914 to accept the sequence and the terminating null character, which shall be added
16915 automatically.
- 16916 Otherwise, the application shall ensure that the corresponding argument is a pointer to
16917 an array of **wchar_t** large enough to accept the sequence and the terminating null wide
16918 character, which shall be added automatically.
- 16919 [Matches a non-empty sequence of wide characters from a set of expected wide
16920 characters (the *scanset*). If no **l** (ell) qualifier is present, wide characters from the input
16921 field shall be converted as if by repeated calls to the *wcrtomb()* function, with the
16922 conversion state described by an **mbstate_t** object initialized to zero before the first
16923 wide character is converted. The application shall ensure that the corresponding
16924 argument is a pointer to a character array large enough to accept the sequence and the
16925 terminating null character, which shall be added automatically.
- 16926 If an **l** (ell) qualifier is present, the application shall ensure that the corresponding
16927 argument is a pointer to an array of **wchar_t** large enough to accept the sequence and
16928 the terminating null wide character, which shall be added automatically.
- 16929 The conversion specification includes all subsequent wide characters in the *format*
16930 string up to and including the matching right square bracket (']'). The wide
16931 characters between the square brackets (the *scanlist*) comprise the scanset, unless the
16932 wide character after the left square bracket is a circumflex ('^'), in which case the
16933 scanset contains all wide characters that do not appear in the scanlist between the
16934 circumflex and the right square bracket. If the conversion specification begins with
16935 "[]" or "[^]", the right square bracket is included in the scanlist and the next right
16936 square bracket is the matching right square bracket that ends the conversion
16937 specification; otherwise, the first right square bracket is the one that ends the
16938 conversion specification. If a '-' is in the scanlist and is not the first wide character,
16939 nor the second where the first wide character is a '^', nor the last wide character, the
16940 behavior is implementation-defined.
- 16941 c Matches a sequence of wide characters of exactly the number specified by the field
16942 width (1 if no field width is present in the conversion specification).
- 16943 If no **l** (ell) length modifier is present, characters from the input field shall be converted
16944 as if by repeated calls to the *wcrtomb()* function, with the conversion state described by
16945 an **mbstate_t** object initialized to zero before the first wide character is converted. The
16946 corresponding argument shall be a pointer to the initial element of a character array
16947 large enough to accept the sequence. No null character is added.
- 16948 If an **l** (ell) length modifier is present, the corresponding argument shall be a pointer to
16949 the initial element of an array of **wchar_t** large enough to accept the sequence. No null
16950 wide character is added.
- 16951 Otherwise, the application shall ensure that the corresponding argument is a pointer to
16952 an array of **wchar_t** large enough to accept the sequence. No null wide character is
16953 added.
- 16954 p Matches an implementation-defined set of sequences, which shall be the same as the set
16955 of sequences that is produced by the %p conversion specification of the corresponding
16956 *fwprintf()* functions. The application shall ensure that the corresponding argument is a
16957 pointer to a pointer to **void**. The interpretation of the input item is implementation-
16958 defined. If the input item is a value converted earlier during the same program
16959 execution, the pointer that results shall compare equal to that value; otherwise, the
16960 behavior of the %p conversion is undefined.

16961 n No input is consumed. The application shall ensure that the corresponding argument is
 16962 a pointer to the integer into which is to be written the number of wide characters read
 16963 from the input so far by this call to the *fwscanf()* functions. Execution of a %n
 16964 conversion specification shall not increment the assignment count returned at the
 16965 completion of execution of the function. No argument shall be converted, but one shall
 16966 be consumed. If the conversion specification includes an assignment-suppressing wide
 16967 character or a field width, the behavior is undefined.

16968 XSI C Equivalent to `lc`.

16969 XSI S Equivalent to `ls`.

16970 % Matches a single '%' wide character; no conversion or assignment shall occur. The
 16971 complete conversion specification shall be %%.

16972 If a conversion specification is invalid, the behavior is undefined.

16973 The conversion specifiers A, E, F, G, and X are also valid and shall be equivalent to, respectively,
 16974 a, e, f, g, and x.

16975 If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before
 16976 any wide characters matching the current conversion specification (except for %n) have been
 16977 read (other than leading white-space, where permitted), execution of the current conversion
 16978 specification shall terminate with an input failure. Otherwise, unless execution of the current
 16979 conversion specification is terminated with a matching failure, execution of the following
 16980 conversion specification (if any) shall be terminated with an input failure.

16981 Reaching the end of the string in *swscanf()* shall be equivalent to encountering end-of-file for
 16982 *fwscanf()*.

16983 If conversion terminates on a conflicting input, the offending input shall be left unread in the
 16984 input. Any trailing white space (including <newline>) shall be left unread unless matched by a
 16985 conversion specification. The success of literal matches and suppressed assignments is only
 16986 directly determinable via the %n conversion specification.

16987 CX The *fwscanf()* and *wscanf()* functions may mark the *st_atime* field of the file associated with
 16988 *stream* for update. The *st_atime* field shall be marked for update by the first successful execution
 16989 of *fgetc()*, *fgetwc()*, *fgets()*, *fgetws()*, *fread()*, *getc()*, *getwc()*, *getchar()*, *getwchar()*, *gets()*, *fscanf()*,
 16990 or *fwscanf()* using *stream* that returns data not supplied by a prior call to *ungetc()*.

16991 RETURN VALUE

16992 Upon successful completion, these functions shall return the number of successfully matched
 16993 and assigned input items; this number can be zero in the event of an early matching failure. If
 16994 the input ends before the first matching failure or conversion, EOF shall be returned. If a read
 16995 error occurs, the error indicator for the stream is set, EOF shall be returned, and *errno* shall be
 16996 set to indicate the error.

16997 ERRORS

16998 For the conditions under which the *fwscanf()* functions shall fail and may fail, refer to *fgetwc()*.

16999 In addition, *fwscanf()* may fail if:

17000 CX [EILSEQ] Input byte sequence does not form a valid character.

17001 CX [EINVAL] There are insufficient arguments.

EXAMPLES17002
17003

The call:

17004
17005

```
int i, n; float x; char name[50];
n = wscanf(L"%d%f%s", &i, &x, name);
```

17006

with the input line:

17007

25 54.32E-1 Hamster

17008
17009

assigns to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* contains the string "Hamster".

17010

The call:

17011
17012

```
int i; float x; char name[50];
(void) wscanf(L"%2d%f*d %[0123456789]", &i, &x, name);
```

17013

with input:

17014

56789 0123 56a72

17015
17016

assigns 56 to *i*, 789.0 to *x*, skips 0123, and places the string "56\0" in *name*. The next call to *getchar()* shall return the character 'a'.

APPLICATION USAGE17018
17019

In format strings containing the '%' form of conversion specifications, each argument in the argument list is used exactly once.

RATIONALE

17020

None.

FUTURE DIRECTIONS

17022

None.

SEE ALSO17025
17026

[getwc\(\)](#), [fwprintf\(\)](#), [setlocale\(\)](#), [wcstod\(\)](#), [wcstol\(\)](#), [wcstoul\(\)](#), [wcrctomb\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale, [<langinfo.h>](#), [<stdio.h>](#), [<wchar.h>](#)

CHANGE HISTORY17028
17029

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

17030

The normative text is updated to avoid use of the term "must" for application requirements.

17032

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

17033

- The prototypes for *fwscanf()* and *swscanf()* are updated.

17034

- The DESCRIPTION is updated.

17035

- The hh, ll, j, t, and z length modifiers are added.

17036

- The a, A, and F conversion characters are added.

17037
17038

The DESCRIPTION is updated to use the terms "conversion specifier" and "conversion specification" consistently.

Issue 7

17040

Functionality relating to the "%n\$" form of conversion specification is moved from the XSI option to the Base.

17041

17042 **NAME**17043 `gai_strerror` — address and name information error description17044 **SYNOPSIS**17045 `#include <netdb.h>`17046 `const char *gai_strerror(int ecode);`17047 **DESCRIPTION**17048 The `gai_strerror()` function shall return a text string describing an error value for the `getaddrinfo()`
17049 and `getnameinfo()` functions listed in the `<netdb.h>` header.17050 When the `ecode` argument is one of the following values listed in the `<netdb.h>` header:

17051	<code>[EAI_AGAIN]</code>	<code>[EAI_NONAME]</code>
17052	<code>[EAI_BADFLAGS]</code>	<code>[EAI_OVERFLOW]</code>
17053	<code>[EAI_FAIL]</code>	<code>[EAI_SERVICE]</code>
17054	<code>[EAI_FAMILY]</code>	<code>[EAI_SOCKTYPE]</code>
17055	<code>[EAI_MEMORY]</code>	<code>[EAI_SYSTEM]</code>

17056 the function return value shall point to a string describing the error. If the argument is not one
17057 of those values, the function shall return a pointer to a string whose contents indicate an
17058 unknown error.17059 **RETURN VALUE**17060 Upon successful completion, `gai_strerror()` shall return a pointer to an implementation-defined
17061 string.17062 **ERRORS**

17063 No errors are defined.

17064 **EXAMPLES**

17065 None.

17066 **APPLICATION USAGE**

17067 None.

17068 **RATIONALE**

17069 None.

17070 **FUTURE DIRECTIONS**

17071 None.

17072 **SEE ALSO**17073 `getaddrinfo()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<netdb.h>`17074 **CHANGE HISTORY**

17075 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

17076 The Open Group Base Resolution bwg2001-009 is applied, which changes the return type from
17077 `char *` to `const char *`. This is for coordination with the IPnG Working Group.17078 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/22 is applied, adding the
17079 `[EAI_OVERFLOW]` error code.

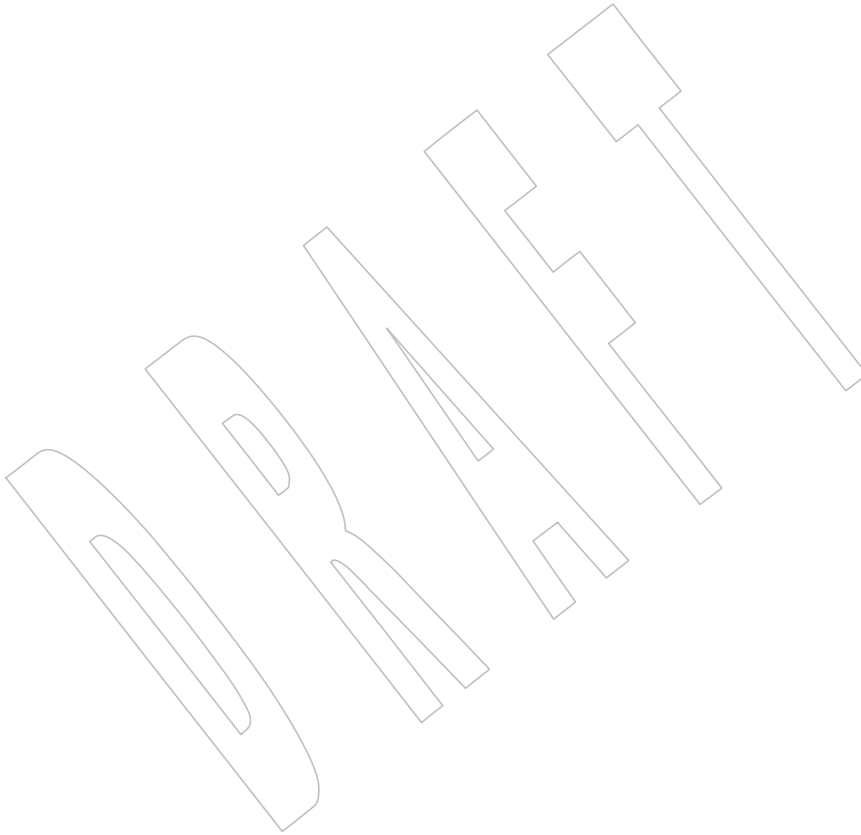
17080 **NAME**
17081 getaddrinfo — get address information

17082 **SYNOPSIS**

```
17083       #include <sys/socket.h>  
17084       #include <netdb.h>  
  
17085       int getaddrinfo(const char *restrict nodename,  
17086                      const char *restrict servname,  
17087                      const struct addrinfo *restrict hints,  
17088                      struct addrinfo **restrict res);
```

17089 **DESCRIPTION**

17090 Refer to *freeaddrinfo()*.



17091 **NAME**

17092 getc — get a byte from a stream

17093 **SYNOPSIS**

17094 #include <stdio.h>

17095 int getc(FILE *stream);

17096 **DESCRIPTION**

17097 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 17098 conflict between the requirements described here and the ISO C standard is unintentional. This
 17099 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

17100 The *getc()* function shall be equivalent to *fgetc()*, except that if it is implemented as a macro it
 17101 may evaluate *stream* more than once, so the argument should never be an expression with side
 17102 effects.

17103 **RETURN VALUE**17104 Refer to *fgetc()*.17105 **ERRORS**17106 Refer to *fgetc()*.17107 **EXAMPLES**

17108 None.

17109 **APPLICATION USAGE**

17110 If the integer value returned by *getc()* is stored into a variable of type **char** and then compared
 17111 against the integer constant EOF, the comparison may never succeed, because sign-extension of
 17112 a variable of type **char** on widening to integer is implementation-defined.

17113 Since it may be implemented as a macro, *getc()* may treat incorrectly a *stream* argument with
 17114 side effects. In particular, *getc(*f++)* does not necessarily work as expected. Therefore, use of this
 17115 function should be preceded by "#undef getc" in such situations; *fgetc()* could also be used.

17116 **RATIONALE**

17117 None.

17118 **FUTURE DIRECTIONS**

17119 None.

17120 **SEE ALSO**17121 *fgetc()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>17122 **CHANGE HISTORY**

17123 First released in Issue 1. Derived from Issue 1 of the SVID.

17124 **NAME**

17125 `getc_unlocked`, `getchar_unlocked`, `putc_unlocked`, `putchar_unlocked` — `stdio` with explicit client
 17126 locking

17127 **SYNOPSIS**

```
17128 CX      #include <stdio.h>
17129
17129      int getc_unlocked(FILE *stream);
17130      int getchar_unlocked(void);
17131      int putc_unlocked(int c, FILE *stream);
17132      int putchar_unlocked(int c);
```

17133 **DESCRIPTION**

17134 Versions of the functions `getc()`, `getchar()`, `putc()`, and `putchar()` respectively named
 17135 `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`, and `putchar_unlocked()` shall be provided
 17136 which are functionally equivalent to the original versions, with the exception that they are not
 17137 required to be implemented in a thread-safe manner. They may only safely be used within a
 17138 scope protected by `flockfile()` (or `ftrylockfile()`) and `funlockfile()`. These functions may safely be
 17139 used in a multi-threaded program if and only if they are called while the invoking thread owns
 17140 the (**FILE ***) object, as is the case after a successful call to the `flockfile()` or `ftrylockfile()` functions.

17141 **RETURN VALUE**

17142 See `getc()`, `getchar()`, `putc()`, and `putchar()`.

17143 **ERRORS**

17144 See `getc()`, `getchar()`, `putc()`, and `putchar()`.

17145 **EXAMPLES**

17146 None.

17147 **APPLICATION USAGE**

17148 Since they may be implemented as macros, `getc_unlocked()` and `putc_unlocked()` may treat
 17149 incorrectly a `stream` argument with side effects. In particular, `getc_unlocked(*f++)` and
 17150 `putc_unlocked(*f++)` do not necessarily work as expected. Therefore, use of these functions in
 17151 such situations should be preceded by the following statement as appropriate:

```
17152      #undef getc_unlocked
17153      #undef putc_unlocked
```

17154 **RATIONALE**

17155 Some I/O functions are typically implemented as macros for performance reasons (for example,
 17156 `putc()` and `getc()`). For safety, they need to be synchronized, but it is often too expensive to
 17157 synchronize on every character. Nevertheless, it was felt that the safety concerns were more
 17158 important; consequently, the `getc()`, `getchar()`, `putc()`, and `putchar()` functions are required to be
 17159 thread-safe. However, unlocked versions are also provided with names that clearly indicate the
 17160 unsafe nature of their operation but can be used to exploit their higher performance. These
 17161 unlocked versions can be safely used only within explicitly locked program regions, using
 17162 exported locking primitives. In particular, a sequence such as:

```
17163      flockfile(fileptr);
17164      putc_unlocked('1', fileptr);
17165      putc_unlocked('\n', fileptr);
17166      fprintf(fileptr, "Line 2\n");
17167      funlockfile(fileptr);
```

17168 is permissible, and results in the text sequence:

17169 1
17170 Line 2

17171 being printed without being interspersed with output from other threads.

17172 It would be wrong to have the standard names such as *getc()*, *putc()*, and so on, map to the
17173 “faster, but unsafe” rather than the “slower, but safe” versions. In either case, you would still
17174 want to inspect all uses of *getc()*, *putc()*, and so on, by hand when converting existing code.
17175 Choosing the safe bindings as the default, at least, results in correct code and maintains the
17176 “atomicity at the function” invariant. To do otherwise would introduce gratuitous
17177 synchronization errors into converted code. Other routines that modify the *stdio* (**FILE** *)
17178 structures or buffers are also safely synchronized.

17179 Note that there is no need for functions of the form *getc_locked()*, *putc_locked()*, and so on, since
17180 this is the functionality of *getc()*, *putc()*, *et al.* It would be inappropriate to use a feature test
17181 macro to switch a macro definition of *getc()* between *getc_locked()* and *getc_unlocked()*, since the
17182 ISO C standard requires an actual function to exist, a function whose behavior could not be
17183 changed by the feature test macro. Also, providing both the *xxx_locked()* and *xxx_unlocked()*
17184 forms leads to the confusion of whether the suffix describes the behavior of the function or the
17185 circumstances under which it should be used.

17186 Three additional routines, *flockfile()*, *ftrylockfile()*, and *funlockfile()* (which may be macros), are
17187 provided to allow the user to delineate a sequence of I/O statements that are executed
17188 synchronously.

17189 The *ungetc()* function is infrequently called relative to the other functions/macros so no
17190 unlocked variation is needed.

17191 FUTURE DIRECTIONS

17192 None.

17193 SEE ALSO

17194 *getc()*, *getchar()*, *putc()*, *putchar()*, the Base Definitions volume of IEEE Std 1003.1-200x,
17195 <stdio.h>

17196 CHANGE HISTORY

17197 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

17198 Issue 6

17199 These functions are marked as part of the Thread-Safe Functions option.

17200 The Open Group Corrigendum U030/2 is applied, adding APPLICATION USAGE describing
17201 how applications should be written to avoid the case when the functions are implemented as
17202 macros.

17203 Issue 7

17204 The *getc_unlocked()*, *getchar_unlocked()*, *putc_unlocked()*, and *putchar_unlocked()* functions are
17205 moved from the Thread-Safe Functions option to the Base.

17206 **NAME**
 17207 getchar — get a byte from a stdin stream

17208 **SYNOPSIS**
 17209 #include <stdio.h>
 17210 int getchar(void);

17211 **DESCRIPTION**
 17212 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 17213 conflict between the requirements described here and the ISO C standard is unintentional. This
 17214 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

17215 The *getchar()* function shall be equivalent to *getc(stdin)*.

17216 **RETURN VALUE**
 17217 Refer to *fgetc()*.

17218 **ERRORS**
 17219 Refer to *fgetc()*.

17220 **EXAMPLES**
 17221 None.

17222 **APPLICATION USAGE**
 17223 If the integer value returned by *getchar()* is stored into a variable of type **char** and then
 17224 compared against the integer constant EOF, the comparison may never succeed, because sign-
 17225 extension of a variable of type **char** on widening to integer is implementation-defined.

17226 **RATIONALE**
 17227 None.

17228 **FUTURE DIRECTIONS**
 17229 None.

17230 **SEE ALSO**
 17231 *getc()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>

17232 **CHANGE HISTORY**
 17233 First released in Issue 1. Derived from Issue 1 of the SVID.

17234 **NAME**
17235 getchar_unlocked — stdio with explicit client locking

17236 **SYNOPSIS**

```
17237 CX       #include <stdio.h>  
17238       int getchar_unlocked(void);
```

17239 **DESCRIPTION**

17240 Refer to *getc_unlocked()*.

17241 **NAME**17242 `getcwd` — get the pathname of the current working directory17243 **SYNOPSIS**17244 `#include <unistd.h>`17245 `char *getcwd(char *buf, size_t size);`17246 **DESCRIPTION**

17247 The `getcwd()` function shall place an absolute pathname of the current working directory in the
 17248 array pointed to by `buf`, and return `buf`. The pathname copied to the array shall contain no
 17249 components that are symbolic links. The `size` argument is the size in bytes of the character array
 17250 pointed to by the `buf` argument. If `buf` is a null pointer, the behavior of `getcwd()` is unspecified.

17251 **RETURN VALUE**

17252 Upon successful completion, `getcwd()` shall return the `buf` argument. Otherwise, `getcwd()` shall
 17253 return a null pointer and set `errno` to indicate the error. The contents of the array pointed to by
 17254 `buf` are then undefined.

17255 **ERRORS**17256 The `getcwd()` function shall fail if:17257 [EINVAL] The `size` argument is 0.17258 [ERANGE] The `size` argument is greater than 0, but is smaller than the length of the
17259 pathname +1.17260 The `getcwd()` function may fail if:

17261 [EACCES] Read or search permission was denied for a component of the pathname.

17262 [ENOMEM] Insufficient storage space is available.

17263 **EXAMPLES**17264 **Determining the Absolute Pathname of the Current Working Directory**

17265 The following example returns a pointer to an array that holds the absolute pathname of the
 17266 current working directory. The pointer is returned in the `ptr` variable, which points to the `buf`
 17267 array where the pathname is stored.

```
17268 #include <stdlib.h>
17269 #include <unistd.h>
17270 ...
17271 long size;
17272 char *buf;
17273 char *ptr;
17274
17275 size = pathconf(".", _PC_PATH_MAX);
17276 if ((buf = (char *)malloc((size_t)size)) != NULL)
17277     ptr = getcwd(buf, (size_t)size);
17278 ...
```

17278 **APPLICATION USAGE**

17279 None.

RATIONALE

Since the maximum pathname length is arbitrary unless `{PATH_MAX}` is defined, an application generally cannot supply a *buf* with *size* `{{PATH_MAX}+1}`.

Having `getcwd()` take no arguments and instead use the `malloc()` function to produce space for the returned argument was considered. The advantage is that `getcwd()` knows how big the working directory pathname is and can allocate an appropriate amount of space. But the programmer would have to use the `free()` function to free the resulting object, or each use of `getcwd()` would further reduce the available memory. Also, `malloc()` and `free()` are used nowhere else in this volume of IEEE Std 1003.1-200x. Finally, `getcwd()` is taken from the SVID where it has the two arguments used in this volume of IEEE Std 1003.1-200x.

The older function `getwd()` was rejected for use in this context because it had only a buffer argument and no *size* argument, and thus had no way to prevent overwriting the buffer, except to depend on the programmer to provide a large enough buffer.

On some implementations, if *buf* is a null pointer, `getcwd()` may obtain *size* bytes of memory using `malloc()`. In this case, the pointer returned by `getcwd()` may be used as the argument in a subsequent call to `free()`. Invoking `getcwd()` with *buf* as a null pointer is not recommended in conforming applications.

If a program is operating in a directory where some (grand)parent directory does not permit reading, `getcwd()` may fail, as in most implementations it must read the directory to determine the name of the file. This can occur if search, but not read, permission is granted in an intermediate directory, or if the program is placed in that directory by some more privileged process (for example, login). Including the `[EACCES]` error condition makes the reporting of the error consistent and warns the application writer that `getcwd()` can fail for reasons beyond the control of the application writer or user. Some implementations can avoid this occurrence (for example, by implementing `getcwd()` using `pwd`, where `pwd` is a set-user-root process), thus the error was made optional. Since this volume of IEEE Std 1003.1-200x permits the addition of other errors, this would be a common addition and yet one that applications could not be expected to deal with without this addition.

FUTURE DIRECTIONS

None.

SEE ALSO

`malloc()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<unistd.h>`

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The `[ENOMEM]` optional error condition is added.

17318 **NAME**17319 `getdate` — convert user format date and time17320 **SYNOPSIS**

```
17321 xSI #include <time.h>
17322 struct tm *getdate(const char *string);
```

17323 **DESCRIPTION**

17324 The `getdate()` function shall convert a string representation of a date or time into a broken-down
 17325 time.

17326 The external variable or macro `getdate_err` is used by `getdate()` to return error values.

17327 Templates are used to parse and interpret the input string. The templates are contained in a text
 17328 file identified by the environment variable `DATEMSK`. The `DATEMSK` variable should be set to
 17329 indicate the full pathname of the file that contains the templates. The first line in the template
 17330 that matches the input specification is used for interpretation and conversion into the internal
 17331 time format.

17332 The following conversion specifications shall be supported:

17333	<code>%%</code>	Equivalent to <code>%</code> .
17334	<code>%a</code>	Abbreviated weekday name.
17335	<code>%A</code>	Full weekday name.
17336	<code>%b</code>	Abbreviated month name.
17337	<code>%B</code>	Full month name.
17338	<code>%c</code>	Locale's appropriate date and time representation.
17339	<code>%C</code>	Century number [00,99]; leading zeros are permitted but not required.
17340	<code>%d</code>	Day of month [01,31]; the leading 0 is optional.
17341	<code>%D</code>	Date as <code>%m/%d/%y</code> .
17342	<code>%e</code>	Equivalent to <code>%d</code> .
17343	<code>%h</code>	Abbreviated month name.
17344	<code>%H</code>	Hour [00,23].
17345	<code>%I</code>	Hour [01,12].
17346	<code>%m</code>	Month number [01,12].
17347	<code>%M</code>	Minute [00,59].
17348	<code>%n</code>	Equivalent to <code><newline></code> .
17349	<code>%p</code>	Locale's equivalent of either AM or PM.
17350	<code>%r</code>	The locale's appropriate representation of time in AM and PM notation. In the POSIX 17351 locale, this shall be equivalent to <code>%I:%M:%S %p</code> .
17352	<code>%R</code>	Time as <code>%H:%M</code> .
17353	<code>%S</code>	Seconds [00,60]. The range goes to 60 (rather than stopping at 59) to allow positive leap 17354 seconds to be expressed. Since leap seconds cannot be predicted by any algorithm, leap 17355 second data must come from some external source.

17356	%t	Equivalent to <tab>.
17357	%T	Time as %H:%M:%S.
17358	%w	Weekday number (Sunday = [0,6]).
17359	%x	Locale's appropriate date representation.
17360	%X	Locale's appropriate time representation.
17361	%y	Year within century. When a century is not otherwise specified, values in the range [69,99] shall refer to years 1969 to 1999 inclusive, and values in the range [00,68] shall refer to years 2000 to 2068 inclusive.
17364		Note: It is expected that in a future version of IEEE Std 1003.1-200x the default century inferred from a 2-digit year will change. (This would apply to all commands accepting a 2-digit year as input.)
17365		
17366		
17367	%Y	Year as "ccyy" (for example, 2001).
17368	%Z	Timezone name or no characters if no timezone exists. If the timezone supplied by %Z is not the timezone that <i>getdate()</i> expects, an invalid input specification error shall result. The <i>getdate()</i> function calculates an expected timezone based on information supplied to the function (such as the hour, day, and month).
17369		
17370		
17371		
17372		The match between the template and input specification performed by <i>getdate()</i> shall be case-insensitive.
17373		
17374		The month and weekday names can consist of any combination of upper and lowercase letters. The process can request that the input date or time specification be in a specific language by setting the <i>LC_TIME</i> category (see <i>setlocale()</i>).
17375		
17376		
17377		Leading zeros are not necessary for the descriptors that allow leading zeros. However, at most two digits are allowed for those descriptors, including leading zeros. Extra whitespace in either the template file or in <i>string</i> shall be ignored.
17378		
17379		
17380		The results are undefined if the conversion specifications %c, %x, and %X include unsupported conversion specifications.
17381		
17382		The following rules apply for converting the input specification into the internal format:
17383		• If %Z is being scanned, then <i>getdate()</i> shall initialize the broken-down time to be the current time in the scanned timezone. Otherwise, it shall initialize the broken-down time based on the current local time as if <i>localtime()</i> had been called.
17384		
17385		
17386		• If only the weekday is given, the day chosen shall be the day, starting with today and moving into the future, which first matches the named day.
17387		
17388		• If only the month (and no year) is given, the month chosen shall be the month, starting with the current month and moving into the future, which first matches the named month. The first day of the month shall be assumed if no day is given.
17389		
17390		
17391		• If no hour, minute, and second are given, the current hour, minute, and second shall be assumed.
17392		
17393		• If no date is given, the hour chosen shall be the hour, starting with the current hour and moving into the future, which first matches the named hour.
17394		
17395		If a conversion specification in the <i>DATMSK</i> file does not correspond to one of the conversion specifications above, the behavior is unspecified.
17396		
17397		The <i>getdate()</i> function need not be thread-safe. A function that is not required to be thread-safe is not required to be reentrant.
17398		

RETURN VALUE

17399

17400

17401

Upon successful completion, *getdate()* shall return a pointer to a **struct tm**. Otherwise, it shall return a null pointer and set *getdate_err* to indicate the error.

ERRORS

17402

17403

17404

The *getdate()* function shall fail in the following cases, setting *getdate_err* to the value shown in the list below. Any changes to *errno* are unspecified.

17405

1. The *DATETIME* environment variable is null or undefined.

17406

2. The template file cannot be opened for reading.

17407

3. Failed to get file status information.

17408

4. The template file is not a regular file.

17409

5. An I/O error is encountered while reading the template file.

17410

6. Memory allocation failed (not enough memory available).

17411

7. There is no line in the template that matches the input.

17412

8. Invalid input specification. For example, February 31; or a time is specified that cannot be represented in a **time_t** (representing the time in seconds since the Epoch).

17413

17414

EXAMPLES

17415

1. The following example shows the possible contents of a template:

17416

```
%m
%A %B %d, %Y, %H:%M:%S
```

17417

```
%A
```

17418

```
%B
```

17419

```
%m/%d/%y %I %p
```

17420

```
%d,%m,%Y %H:%M
```

17421

```
at %A the %dst of %B in %Y
```

17422

```
run job at %I %p,%B %dnd
```

17423

```
%A den %d. %B %Y %H.%M Uhr
```

17424

17425

2. The following are examples of valid input specifications for the template in Example 1:

17426

```
getdate("10/1/87 4 PM");
```

17427

```
getdate("Friday");
```

17428

```
getdate("Friday September 18, 1987, 10:30:30");
```

17429

```
getdate("24,9,1986 10:30");
```

17430

```
getdate("at monday the 1st of december in 1986");
```

17431

```
getdate("run job at 3 PM, december 2nd");
```

17432

If the *LC_TIME* category is set to a German locale that includes *freitag* as a weekday name and *oktober* as a month name, the following would be valid:

17433

17434

```
getdate("freitag den 10. oktober 1986 10.30 Uhr");
```

17435

3. The following example shows how local date and time specification can be defined in the template:

17436

17437

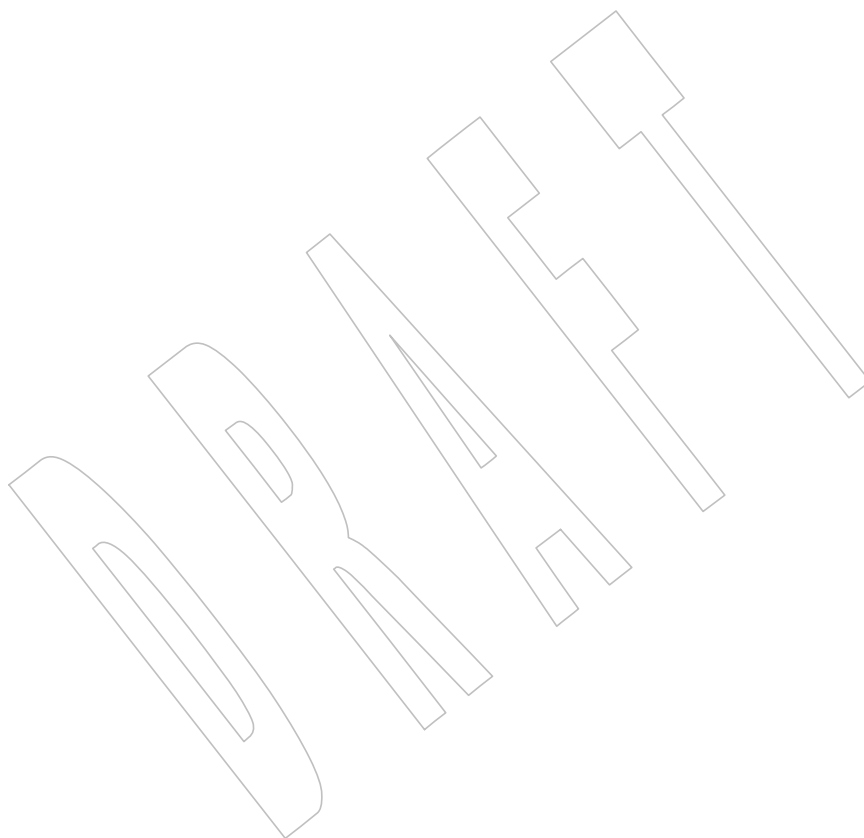
17438

17439

17440

17441

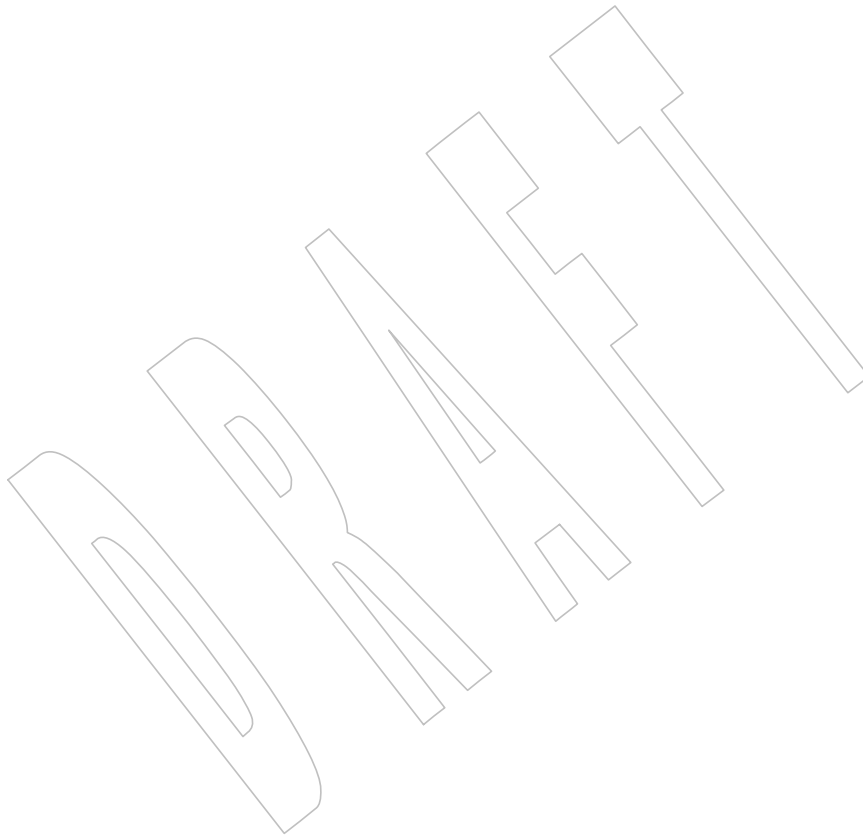
Invocation	Line in Template
<code>getdate("11/27/86")</code>	<code>%m/%d/%y</code>
<code>getdate("27.11.86")</code>	<code>%d.%m.%y</code>
<code>getdate("86-11-27")</code>	<code>%y-%m-%d</code>
<code>getdate("Friday 12:00:00")</code>	<code>%A %H:%M:%S</code>



17488

17489

The DESCRIPTION is updated to refer to conversion specifications instead of field descriptors for consistency with other functions.



17490 **NAME**17491 getdelim, getline — read a delimited record from *stream*17492 **SYNOPSIS**

```

17493 CX      #include <stdio.h>
17494         ssize_t getdelim(char **lineptr, size_t *n, int delimiter,
17495         FILE *stream);
17496         ssize_t getline(char **lineptr, size_t *n, FILE *stream);

```

17497 **DESCRIPTION**

17498 The *getdelim()* function shall read from *stream* until it encounters a character matching the
 17499 *delimiter* character. The *delimiter* argument is an **int**, the value of which the application shall
 17500 ensure is a character representable as an **unsigned char** of equal value that terminates the read
 17501 process. If the *delimiter* argument has any other value, the behavior is undefined.

17502 The application shall ensure that **lineptr* is a valid argument that could be passed to the *free()*
 17503 function. If **n* is non-zero, the application shall ensure that **lineptr* points to an object of size at
 17504 least **n* bytes.

17505 The size of the object pointed to by **lineptr* shall be increased to fit the incoming line, if it isn't
 17506 already large enough. The characters read shall be stored in the string pointed to by the *lineptr*
 17507 argument.

17508 The *getline()* function shall be equivalent to the *getdelim()* function with the *delimiter* character
 17509 equal to the <newline> character.

17510 **RETURN VALUE**

17511 Upon successful completion, the *getdelim()* function shall return the number of characters
 17512 written into the buffer, including the delimiter character if one was encountered before EOF.
 17513 Otherwise, it shall return -1 and set *errno* to indicate the error.

17514 **ERRORS**

17515 These functions shall fail if:

17516 [EINVAL] When *lineptr* or *n* are a null pointer.

17517 [ENOMEM] Insufficient memory is available.

17518 These functions may fail if:

17519 [EINVAL] *stream* is not a valid file descriptor.

17520 [EOVERFLOW] More than {SSIZE_MAX} characters were read without encountering the
 17521 *delimiter* character.

17522 **EXAMPLES**

```

17523     #include <stdio.h>
17524     #include <stdlib.h>
17525
17526     int main(void)
17527     {
17528         FILE *fp;
17529         char *line = NULL;
17530         size_t len = 0;
17531         ssize_t read;
17532         fp = fopen("/etc/motd", "r");
17533         if (fp == NULL)

```

```

17533         exit(1);
17534     while ((read = getline(&line, &len, fp)) != -1) {
17535         printf("Retrieved line of length %zu :\n", read);
17536         printf("%s", line);
17537     }
17538     if (line)
17539         free(line);
17540     fclose(fp);
17541     return 0;
17542 }

```

APPLICATION USAGE

Setting **lineptr* to a null pointer and **n* to zero are allowed and a recommended way to start parsing a file.

RATIONALE

These functions are widely used to solve the problem that the *fgets()* function has with long lines. The functions automatically enlarge the target buffers if needed. These are especially useful since they reduce code needed for applications.

FUTURE DIRECTIONS

None.

SEE ALSO

fgets(), *free()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 7.

17556 **NAME**

17557 getegid — get the effective group ID

17558 **SYNOPSIS**

17559 #include <unistd.h>

17560 gid_t getegid(void);

17561 **DESCRIPTION**17562 The *getegid()* function shall return the effective group ID of the calling process.17563 **RETURN VALUE**17564 The *getegid()* function shall always be successful and no return value is reserved to indicate an
17565 error.17566 **ERRORS**

17567 No errors are defined.

17568 **EXAMPLES**

17569 None.

17570 **APPLICATION USAGE**

17571 None.

17572 **RATIONALE**

17573 None.

17574 **FUTURE DIRECTIONS**

17575 None.

17576 **SEE ALSO**17577 *geteuid()*, *getgid()*, *getuid()*, *setegid()*, *seteuid()*, *setgid()*, *setregid()*, *setreuid()*, *setuid()*, the Base
17578 Definitions volume of IEEE Std 1003.1-200x, <sys/types.h>, <unistd.h>17579 **CHANGE HISTORY**

17580 First released in Issue 1. Derived from Issue 1 of the SVID.

17581 **Issue 6**

17582 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

17583 The following new requirements on POSIX implementations derive from alignment with the
17584 Single UNIX Specification:

- 17585
- The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
17586 required for conforming implementations of previous POSIX specifications, it was not
17587 required for UNIX applications.

17588 **NAME**
 17589 `getenv` — get value of an environment variable

17590 **SYNOPSIS**
 17591 `#include <stdlib.h>`

17592 `char *getenv(const char *name);`

17593 **DESCRIPTION**

17594 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 17595 conflict between the requirements described here and the ISO C standard is unintentional. This
 17596 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

17597 The `getenv()` function shall search the environment of the calling process (see the Base
 17598 Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables) for the
 17599 environment variable *name* if it exists and return a pointer to the value of the environment
 17600 variable. If the specified environment variable cannot be found, a null pointer shall be returned.
 17601 The application shall ensure that it does not modify the string pointed to by the `getenv()`
 17602 function.

17603 CX The string pointed to may be overwritten by a subsequent call to `getenv()`, `setenv()`, `unsetenv()`,
 17604 XSI or `putenv()` but shall not be overwritten by a call to any other function in this volume of
 17605 IEEE Std 1003.1-200x.

17606 CX If the application modifies *environ* or the pointers to which it points, the behavior of `getenv()` is
 17607 undefined.

17608 The `getenv()` function need not be thread-safe. A function that is not required to be thread-safe is
 17609 not required to be reentrant.

17610 **RETURN VALUE**

17611 Upon successful completion, `getenv()` shall return a pointer to a string containing the *value* for
 17612 the specified *name*. If the specified *name* cannot be found in the environment of the calling
 17613 process, a null pointer shall be returned.

17614 **ERRORS**

17615 No errors are defined.

17616 **EXAMPLES**

17617 **Getting the Value of an Environment Variable**

17618 The following example gets the value of the *HOME* environment variable.

17619 `#include <stdlib.h>`
 17620 `...`
 17621 `const char *name = "HOME";`
 17622 `char *value;`
 17623 `value = getenv(name);`

17624 **APPLICATION USAGE**

17625 None.

17626 **RATIONALE**

17627 The `clearenv()` function was considered but rejected. The `putenv()` function has now been
 17628 included for alignment with the Single UNIX Specification.

17629 The `getenv()` function is inherently not reentrant because it returns a value pointing to static
 17630 data.

17631 Conforming applications are required not to modify *environ* directly, but to use only the
 17632 functions described here to manipulate the process environment as an abstract object. Thus, the
 17633 implementation of the environment access functions has complete control over the data
 17634 structure used to represent the environment (subject to the requirement that *environ* be
 17635 maintained as a list of strings with embedded equal signs for applications that wish to scan the
 17636 environment). This constraint allows the implementation to properly manage the memory it
 17637 allocates, either by using allocated storage for all variables (copying them on the first invocation
 17638 of *setenv()* or *unsetenv()*), or keeping track of which strings are currently in allocated space and
 17639 which are not, via a separate table or some other means. This enables the implementation to free
 17640 any allocated space used by strings (and perhaps the pointers to them) stored in *environ* when
 17641 *unsetenv()* is called. A C runtime start-up procedure (that which invokes *main()* and perhaps
 17642 initializes *environ*) can also initialize a flag indicating that none of the environment has yet been
 17643 copied to allocated storage, or that the separate table has not yet been initialized.

17644 In fact, for higher performance of *getenv()*, the implementation could also maintain a separate
 17645 copy of the environment in a data structure that could be searched much more quickly (such as
 17646 an indexed hash table, or a binary tree), and update both it and the linear list at *environ* when
 17647 *setenv()* or *unsetenv()* is invoked.

17648 Performance of *getenv()* can be important for applications which have large numbers of
 17649 environment variables. Typically, applications like this use the environment as a resource
 17650 database of user-configurable parameters. The fact that these variables are in the user's shell
 17651 environment usually means that any other program that uses environment variables (such as *ls*,
 17652 which attempts to use *COLUMNS*), or really almost any utility (*LANG*, *LC_ALL*, and so on) is
 17653 similarly slowed down by the linear search through the variables.

17654 An implementation that maintains separate data structures, or even one that manages the
 17655 memory it consumes, is not currently required as it was thought it would reduce consensus
 17656 among implementors who do not want to change their historical implementations.

17657 The POSIX Threads Extension states that multi-threaded applications must not modify *environ*
 17658 directly, and that IEEE Std 1003.1-200x is providing functions which such applications can use in
 17659 the future to manipulate the environment in a thread-safe manner. Thus, moving away from
 17660 application use of *environ* is desirable from that standpoint as well.

17661 FUTURE DIRECTIONS

17662 None.

17663 SEE ALSO

17664 *exec*, *putenv()*, *setenv()*, *unsetenv()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter
 17665 8, Environment Variables, `<stdlib.h>`

17666 CHANGE HISTORY

17667 First released in Issue 1. Derived from Issue 1 of the SVID.

17668 Issue 5

17669 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
 17670 VALUE section.

17671 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

17672 Issue 6

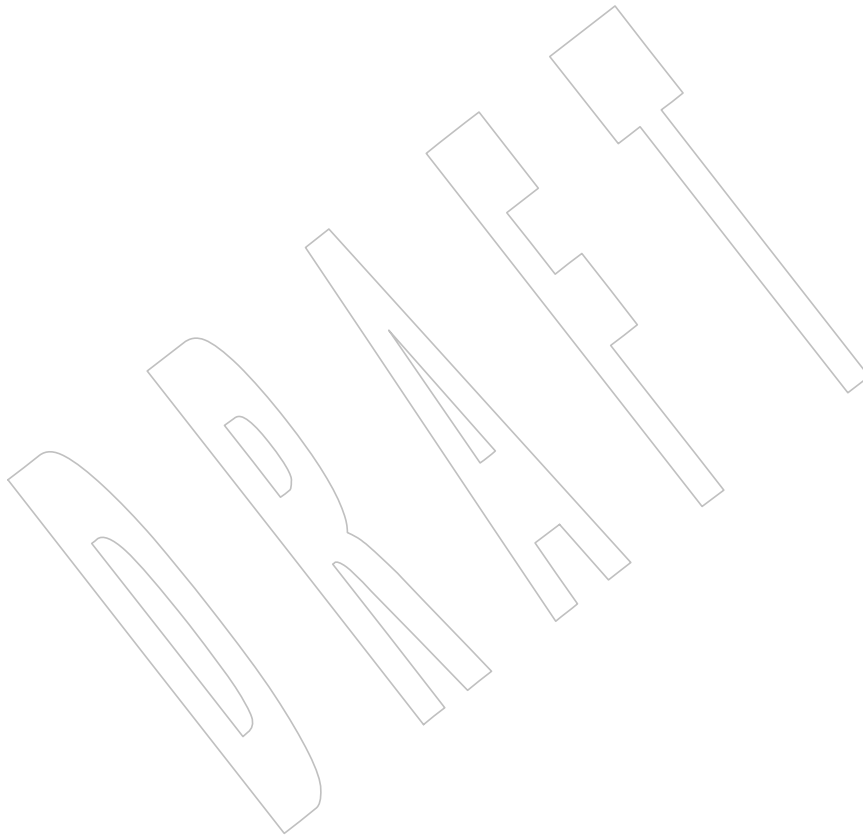
17673 The following changes were made to align with the IEEE P1003.1a draft standard:

- 17674 • References added to the new *setenv()* and *unsetenv()* functions.

17675 The normative text is updated to avoid use of the term “must” for application requirements.

17676
17677
17678**Issue 7**

Austin Group Interpretation 1003.1-2001 #062 is applied, clarifying that a call to *putenv()* may also cause the string to be overwritten.



17679 **NAME**

17680 geteuid — get the effective user ID

17681 **SYNOPSIS**

17682 #include <unistd.h>

17683 uid_t geteuid(void);

17684 **DESCRIPTION**17685 The *geteuid()* function shall return the effective user ID of the calling process.17686 **RETURN VALUE**17687 The *geteuid()* function shall always be successful and no return value is reserved to indicate an
17688 error.17689 **ERRORS**

17690 No errors are defined.

17691 **EXAMPLES**

17692 None.

17693 **APPLICATION USAGE**

17694 None.

17695 **RATIONALE**

17696 None.

17697 **FUTURE DIRECTIONS**

17698 None.

17699 **SEE ALSO**17700 *getegid()*, *getgid()*, *getuid()*, *setegid()*, *seteuid()*, *setgid()*, *setregid()*, *setreuid()*, *setuid()*, the Base
17701 Definitions volume of IEEE Std 1003.1-200x, <sys/types.h>, <unistd.h>17702 **CHANGE HISTORY**

17703 First released in Issue 1. Derived from Issue 1 of the SVID.

17704 **Issue 6**

17705 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

17706 The following new requirements on POSIX implementations derive from alignment with the
17707 Single UNIX Specification:

- 17708
- The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
17709 required for conforming implementations of previous POSIX specifications, it was not
17710 required for UNIX applications.

17711 **NAME**

17712 getgid — get the real group ID

17713 **SYNOPSIS**

17714 #include <unistd.h>

17715 gid_t getgid(void);

17716 **DESCRIPTION**17717 The *getgid()* function shall return the real group ID of the calling process.17718 **RETURN VALUE**17719 The *getgid()* function shall always be successful and no return value is reserved to indicate an
17720 error.17721 **ERRORS**

17722 No errors are defined.

17723 **EXAMPLES**

17724 None.

17725 **APPLICATION USAGE**

17726 None.

17727 **RATIONALE**

17728 None.

17729 **FUTURE DIRECTIONS**

17730 None.

17731 **SEE ALSO**17732 *getegid()*, *geteuid()*, *getuid()*, *setegid()*, *seteuid()*, *setgid()*, *setregid()*, *setreuid()*, *setuid()*, the Base
17733 Definitions volume of IEEE Std 1003.1-200x, <sys/types.h>, <unistd.h>17734 **CHANGE HISTORY**

17735 First released in Issue 1. Derived from Issue 1 of the SVID.

17736 **Issue 6**

17737 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

17738 The following new requirements on POSIX implementations derive from alignment with the
17739 Single UNIX Specification:

- 17740
- The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
17741 required for conforming implementations of previous POSIX specifications, it was not
17742 required for UNIX applications.

17743 **NAME**
17744 getgrent — get the group database entry

17745 **SYNOPSIS**

```
17746 xSI #include <grp.h>  
17747 struct group *getgrent(void);
```

17748 **DESCRIPTION**

17749 Refer to *endgrent()*.

17750 **NAME**17751 `getgrgid, getgrgid_r` — get group database entry for a group ID17752 **SYNOPSIS**

```
17753 #include <grp.h>
17754
17754 struct group *getgrgid(gid_t gid);
17755 int getgrgid_r(gid_t gid, struct group *grp, char *buffer,
17756 size_t bufsize, struct group **result);
```

17757 **DESCRIPTION**17758 The `getgrgid()` function shall search the group database for an entry with a matching *gid*.17759 The `getgrgid()` function need not be thread-safe. A function that is not required to be thread-safe
17760 is not required to be reentrant.

17761 The `getgrgid_r()` function shall update the **group** structure pointed to by *grp* and store a pointer
17762 to that structure at the location pointed to by *result*. The structure shall contain an entry from
17763 the group database with a matching *gid*. Storage referenced by the group structure is allocated
17764 from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The
17765 maximum size needed for this buffer can be determined with the `{_SC_GETGR_R_SIZE_MAX}`
17766 `sysconf()` parameter. A NULL pointer shall be returned at the location pointed to by *result* on
17767 error or if the requested entry is not found.

17768 **RETURN VALUE**

17769 Upon successful completion, `getgrgid()` shall return a pointer to a **struct group** with the structure
17770 defined in `<grp.h>` with a matching entry if one is found. The `getgrgid()` function shall return a
17771 null pointer if either the requested entry was not found, or an error occurred. On error, *errno*
17772 shall be set to indicate the error.

17773 The return value may point to a static area which is overwritten by a subsequent call to
17774 `getgrent()`, `getgrgid()`, or `getgrnam()`.

17775 If successful, the `getgrgid_r()` function shall return zero; otherwise, an error number shall be
17776 returned to indicate the error.

17777 **ERRORS**17778 The `getgrgid()` and `getgrgid_r()` functions may fail if:

- 17779 [EIO] An I/O error has occurred.
- 17780 [EINTR] A signal was caught during `getgrgid()`.
- 17781 [EMFILE] All file descriptors available to the process are currently open.
- 17782 [ENFILE] The maximum allowable number of files is currently open in the system.

17783 The `getgrgid_r()` function may fail if:

- 17784 [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be
17785 referenced by the resulting **group** structure.

EXAMPLES**Finding an Entry in the Group Database**

The following example uses `getgrgid()` to search the group database for a group ID that was previously stored in a `stat` structure, then prints out the group name if it is found. If the group is not found, the program prints the numeric value of the group for the entry.

```

17791 #include <sys/types.h>
17792 #include <grp.h>
17793 #include <stdio.h>
17794 ...
17795 struct stat statbuf;
17796 struct group *grp;
17797 ...
17798 if ((grp = getgrgid(statbuf.st_gid)) != NULL)
17799     printf(" %-8.8s", grp->gr_name);
17800 else
17801     printf(" %-8d", statbuf.st_gid);
17802 ...

```

APPLICATION USAGE

Applications wishing to check for error situations should set `errno` to 0 before calling `getgrgid()`. If `errno` is set on return, an error occurred.

The `getgrgid_r()` function is thread-safe and shall return values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`endgrent()`, `getgrnam()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<grp.h>`, `<limits.h>`, `<sys/types.h>`

CHANGE HISTORY

First released in Issue 1. Derived from System V Release 2.0.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

The `getgrgid_r()` function is included for alignment with the POSIX Threads Extension.

A note indicating that the `getgrgid()` function need not be reentrant is added to the DESCRIPTION.

Issue 6

The `getgrgid_r()` function is marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION describing matching the `gid`.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

17831

17832

17833

- The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

17834

- In the RETURN VALUE section, the requirement to set *errno* on error is added.

17835

- The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.

17836

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

17837

17838

IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the buffer from *bufsize* characters to bytes.

17839

17840

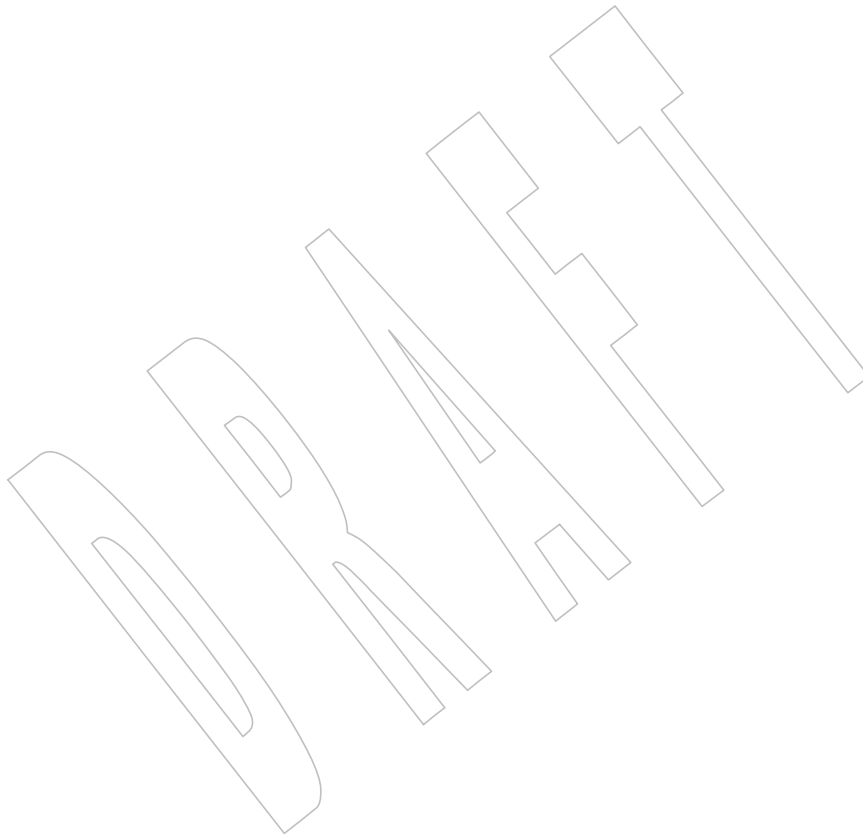
Issue 7

17841

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

17842

The *getgrgid_r()* function is moved from the Thread-Safe Functions option to the Base.



17843 **NAME**

17844 getgrnam, getgrnam_r — search group database for a name

17845 **SYNOPSIS**

17846 #include <grp.h>

17847 struct group *getgrnam(const char *name);

17848 int getgrnam_r(const char *name, struct group *grp, char *buffer,
17849 size_t bufsize, struct group **result);17850 **DESCRIPTION**17851 The *getgrnam()* function shall search the group database for an entry with a matching *name*.17852 The *getgrnam()* function need not be thread-safe. A function that is not required to be thread-
17853 safe is not required to be reentrant.17854 The *getgrnam_r()* function shall update the **group** structure pointed to by *grp* and store a pointer
17855 to that structure at the location pointed to by *result*. The structure shall contain an entry from
17856 the group database with a matching *gid* or *name*. Storage referenced by the **group** structure is
17857 allocated from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The
17858 maximum size needed for this buffer can be determined with the `{_SC_GETGR_R_SIZE_MAX}`
17859 *sysconf()* parameter. A NULL pointer is returned at the location pointed to by *result* on error or if
17860 the requested entry is not found.17861 **RETURN VALUE**17862 The *getgrnam()* function shall return a pointer to a **struct group** with the structure defined in
17863 <grp.h> with a matching entry if one is found. The *getgrnam_r()* function shall return a null
17864 pointer if either the requested entry was not found, or an error occurred. On error, *errno* shall be
17865 set to indicate the error.17866 The return value may point to a static area which is overwritten by a subsequent call to
17867 *getgrent()*, *getgrgid()*, or *getgrnam()*.17868 The *getgrnam_r()* function shall return zero on success or if the requested entry was not found
17869 and no error has occurred. If any error has occurred, an error number shall be returned to
17870 indicate the error.17871 **ERRORS**17872 The *getgrnam()* and *getgrnam_r()* functions may fail if:

17873 [EIO] An I/O error has occurred.

17874 [EINTR] A signal was caught during *getgrnam()*.

17875 [EMFILE] All file descriptors available to the process are currently open.

17876 [ENFILE] The maximum allowable number of files is currently open in the system.

17877 The *getgrnam_r()* function may fail if:17878 [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be
17879 referenced by the resulting **group** structure.

17880
17881
17882
17883
17884
17885
17886
17887
17888
17889
17890
17891
17892
17893
17894
17895
17896
17897
17898
17899
17900
17901
17902
17903
17904
17905
17906
17907
17908
17909
17910
17911
17912
17913
17914
17915
17916
17917
17918
17919
17920

EXAMPLES

None.

APPLICATION USAGE

Applications wishing to check for error situations should set *errno* to 0 before calling *getgrnam()*. If *errno* is set on return, an error occurred.

The *getgrnam_r()* function is thread-safe and shall return values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endgrent(), *getgrgid()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<grp.h>**, **<limits.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from System V Release 2.0.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

The *getgrnam_r()* function is included for alignment with the POSIX Threads Extension.

A note indicating that the *getgrnam()* function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *getgrnam_r()* function is marked as part of the Thread-Safe Functions option.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the RETURN VALUE section, the requirement to set *errno* on error is added.
- The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the buffer from *bufsize* characters to bytes.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

Austin Group Interpretation 1003.1-2001 #081 is applied, clarifying the RETURN VALUE section.

The *getgrnam_r()* function is moved from the Thread-Safe Functions option to the Base.

17921 **NAME**

17922 getgroups — get supplementary group IDs

17923 **SYNOPSIS**

17924 #include <unistd.h>

17925 int getgroups(int *gidsetsize*, gid_t *grouplist*[]);17926 **DESCRIPTION**

17927 The *getgroups()* function shall fill in the array *grouplist* with the current supplementary group
 17928 IDs of the calling process. It is implementation-defined whether *getgroups()* also returns the
 17929 effective group ID in the *grouplist* array.

17930 The *gidsetsize* argument specifies the number of elements in the array *grouplist*. The actual
 17931 number of group IDs stored in the array shall be returned. The values of array entries with
 17932 indices greater than or equal to the value returned are undefined.

17933 If *gidsetsize* is 0, *getgroups()* shall return the number of group IDs that it would otherwise return
 17934 without modifying the array pointed to by *grouplist*.

17935 If the effective group ID of the process is returned with the supplementary group IDs, the value
 17936 returned shall always be greater than or equal to one and less than or equal to the value of
 17937 {NGROUPS_MAX}+1.

17938 **RETURN VALUE**

17939 Upon successful completion, the number of supplementary group IDs shall be returned. A
 17940 return value of -1 indicates failure and *errno* shall be set to indicate the error.

17941 **ERRORS**17942 The *getgroups()* function shall fail if:

17943 [EINVAL] The *gidsetsize* argument is non-zero and less than the number of group IDs
 17944 that would have been returned.

17945 **EXAMPLES**17946 **Getting the Supplementary Group IDs of the Calling Process**

17947 The following example places the current supplementary group IDs of the calling process into
 17948 the *group* array.

```

17949 #include <sys/types.h>
17950 #include <unistd.h>
17951 ...
17952 gid_t *group;
17953 int nogroups;
17954 long ngroups_max;

17955 ngroups_max = sysconf(_SC_NGROUPS_MAX) + 1;
17956 group = (gid_t *)malloc(ngroups_max *sizeof(gid_t));
17957 ngroups = getgroups(ngroups_max, group);

```

17958 **APPLICATION USAGE**

17959 None.

RATIONALE

The related function *setgroups()* is a privileged operation and therefore is not covered by this volume of IEEE Std 1003.1-200x.

As implied by the definition of supplementary groups, the effective group ID may appear in the array returned by *getgroups()* or it may be returned only by *getegid()*. Duplication may exist, but the application needs to call *getegid()* to be sure of getting all of the information. Various implementation variations and administrative sequences cause the set of groups appearing in the result of *getgroups()* to vary in order and as to whether the effective group ID is included, even when the set of groups is the same (in the mathematical sense of “set”). (The history of a process and its parents could affect the details of the result.)

Application writers should note that {NGROUPS_MAX} is not necessarily a constant on all implementations.

FUTURE DIRECTIONS

None.

SEE ALSO

getegid(), *setgid()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<sys/types.h>`, `<unistd.h>`

CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

Issue 6

In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- A return value of 0 is not permitted, because {NGROUPS_MAX} cannot be 0. This is a FIPS requirement.

The following changes were made to align with the IEEE P1003.1a draft standard:

- An explanation is added that the effective group ID may be included in the supplementary group list.

17994 **NAME**
17995 gethostent — network host database functions

17996 **SYNOPSIS**
17997 #include <netdb.h>
17998 struct hostent *gethostent(void);

17999 **DESCRIPTION**
18000 Refer to *endhostent()*.



18001 **NAME**
 18002 gethostid — get an identifier for the current host

18003 **SYNOPSIS**

18004 XSI #include <unistd.h>
 18005 long gethostid(void);

18006 **DESCRIPTION**

18007 The *gethostid()* function shall retrieve a 32-bit identifier for the current host.

18008 **RETURN VALUE**

18009 Upon successful completion, *gethostid()* shall return an identifier for the current host.

18010 **ERRORS**

18011 No errors are defined.

18012 **EXAMPLES**

18013 None.

18014 **APPLICATION USAGE**

18015 This volume of IEEE Std 1003.1-200x does not define the domain in which the return value is
 18016 unique.

18017 **RATIONALE**

18018 None.

18019 **FUTURE DIRECTIONS**

18020 None.

18021 **SEE ALSO**

18022 *random()*, the Base Definitions volume of IEEE Std 1003.1-200x, <unistd.h>

18023 **CHANGE HISTORY**

18024 First released in Issue 4, Version 2.

18025 **Issue 5**

18026 Moved from X/OPEN UNIX extension to BASE.

18027 **NAME**

18028 gethostname — get name of current host

18029 **SYNOPSIS**

18030 #include <unistd.h>

18031 int gethostname(char *name, size_t namelen);

18032 **DESCRIPTION**

18033 The *gethostname()* function shall return the standard host name for the current machine. The
 18034 *namelen* argument shall specify the size of the array pointed to by the *name* argument. The
 18035 returned name shall be null-terminated, except that if *namelen* is an insufficient length to hold
 18036 the host name, then the returned name shall be truncated and it is unspecified whether the
 18037 returned name is null-terminated.

18038 Host names are limited to {HOST_NAME_MAX} bytes.

18039 **RETURN VALUE**

18040 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned.

18041 **ERRORS**

18042 No errors are defined.

18043 **EXAMPLES**

18044 None.

18045 **APPLICATION USAGE**

18046 None.

18047 **RATIONALE**

18048 None.

18049 **FUTURE DIRECTIONS**

18050 None.

18051 **SEE ALSO**18052 *gethostid()*, *uname()*, the Base Definitions volume of IEEE Std 1003.1-200x, <unistd.h>18053 **CHANGE HISTORY**

18054 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

18055 The Open Group Base Resolution bwg2001-008 is applied, changing the *namelen* parameter from
 18056 *socklen_t* to *size_t*.

18057 **NAME**

18058 getitimer, setitimer — get and set value of interval timer

18059 **SYNOPSIS**

```

18060 OB XSI #include <sys/time.h>
18061
18061 int getitimer(int which, struct itimerval *value);
18062 int setitimer(int which, const struct itimerval *restrict value,
18063              struct itimerval *restrict ovalue);

```

18064 **DESCRIPTION**

18065 The *getitimer()* function shall store the current value of the timer specified by *which* into the
 18066 structure pointed to by *value*. The *setitimer()* function shall set the timer specified by *which* to the
 18067 value specified in the structure pointed to by *value*, and if *ovalue* is not a null pointer, store the
 18068 previous value of the timer in the structure pointed to by *ovalue*.

18069 A timer value is defined by the **itimerval** structure, specified in **<sys/time.h>**. If *it_value* is non-
 18070 zero, it shall indicate the time to the next timer expiration. If *it_interval* is non-zero, it shall
 18071 specify a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to 0 shall
 18072 disable a timer, regardless of the value of *it_interval*. Setting *it_interval* to 0 shall disable a timer
 18073 after its next expiration (assuming *it_value* is non-zero).

18074 Implementations may place limitations on the granularity of timer values. For each interval
 18075 timer, if the requested timer value requires a finer granularity than the implementation supports,
 18076 the actual timer value shall be rounded up to the next supported value.

18077 An XSI-conforming implementation provides each process with at least three interval timers,
 18078 which are indicated by the *which* argument:

18079 **ITIMER_PROF** Decrements both in process virtual time and when the system is running
 18080 on behalf of the process. It is designed to be used by interpreters in
 18081 statistically profiling the execution of interpreted programs. Each time the
 18082 **ITIMER_PROF** timer expires, the **SIGPROF** signal is delivered.

18083 **ITIMER_REAL** Decrements in real time. A **SIGALRM** signal is delivered when this timer
 18084 expires.

18085 **ITIMER_VIRTUAL** Decrements in process virtual time. It runs only when the process is
 18086 executing. A **SIGVTALRM** signal is delivered when it expires.

18087 The interaction between *setitimer()* and *alarm()* or *sleep()* is unspecified.

18088 **RETURN VALUE**

18089 Upon successful completion, *getitimer()* or *setitimer()* shall return 0; otherwise, -1 shall be
 18090 returned and *errno* set to indicate the error.

18091 **ERRORS**

18092 The *setitimer()* function shall fail if:

18093 **[EINVAL]** The *value* argument is not in canonical form. (In canonical form, the number of
 18094 microseconds is a non-negative integer less than 1 000 000 and the number of
 18095 seconds is a non-negative integer.)

18096 The *getitimer()* and *setitimer()* functions may fail if:

18097 **[EINVAL]** The *which* argument is not recognized.

18098
18099
18100
18101
18102
18103
18104
18105
18106
18107
18108
18109
18110
18111
18112
18113
18114
18115
18116
18117
18118

EXAMPLES

None.

APPLICATION USAGE

Applications should use the *timer_gettime()* and *timer_settime()* functions instead of the obsolescent *getitimer()* and *setitimer()* functions, respectively.

RATIONALE

None.

FUTURE DIRECTIONS

The *getitimer()* and *setitimer()* functions may be removed in a future version.

SEE ALSO

alarm(), *exec*, *sleep()*, *timer_getoverrun()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<sys/time.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The **restrict** keyword is added to the *setitimer()* prototype for alignment with the ISO/IEC 9899:1999 standard.

Issue 7

The *getitimer()* and *setitimer()* functions are marked obsolescent.

getline()18119 **NAME**18120 `getline` — read a delimited record from *stream*18121 **SYNOPSIS**18122 CX `#include <stdio.h>`
18123 `ssize_t getline(char **lineptr, size_t *n, FILE *stream);`18124 **DESCRIPTION**18125 Refer to [*getdelim\(\)*](#).

18126 **NAME**

18127 getlogin, getlogin_r — get login name

18128 **SYNOPSIS**

18129 #include <unistd.h>

18130 char *getlogin(void);

18131 int getlogin_r(char *name, size_t namesize);

18132 **DESCRIPTION**

18133 The *getlogin()* function shall return a pointer to a string containing the user name associated by
 18134 the login activity with the controlling terminal of the current process. If *getlogin()* returns a non-
 18135 null pointer, then that pointer points to the name that the user logged in under, even if there are
 18136 several login names with the same user ID.

18137 The *getlogin()* function need not be thread-safe. A function that is not required to be thread-safe
 18138 is not required to be reentrant.

18139 The *getlogin_r()* function shall put the name associated by the login activity with the controlling
 18140 terminal of the current process in the character array pointed to by *name*. The array is *namesize*
 18141 characters long and should have space for the name and the terminating null character. The
 18142 maximum size of the login name is {LOGIN_NAME_MAX}.

18143 If *getlogin_r()* is successful, *name* points to the name the user used at login, even if there are
 18144 several login names with the same user ID.

18145 **RETURN VALUE**

18146 Upon successful completion, *getlogin()* shall return a pointer to the login name or a null pointer
 18147 if the user's login name cannot be found. Otherwise, it shall return a null pointer and set *errno* to
 18148 indicate the error.

18149 The return value from *getlogin()* may point to static data whose content is overwritten by each
 18150 call.

18151 If successful, the *getlogin_r()* function shall return zero; otherwise, an error number shall be
 18152 returned to indicate the error.

18153 **ERRORS**

18154 The *getlogin()* and *getlogin_r()* functions may fail if:

18155 [EMFILE] All file descriptors available to the process are currently open.

18156 [ENFILE] The maximum allowable number of files is currently open in the system.

18157 [ENXIO] The calling process has no controlling terminal.

18158 The *getlogin_r()* function may fail if:

18159 [ERANGE] The value of *namesize* is smaller than the length of the string to be returned
 18160 including the terminating null character.

EXAMPLES**Getting the User Login Name**

The following example calls the *getlogin()* function to obtain the name of the user associated with the calling process, and passes this information to the *getpwnam()* function to get the associated user database information.

```

18166 #include <unistd.h>
18167 #include <sys/types.h>
18168 #include <pwd.h>
18169 #include <stdio.h>
18170 ...
18171 char *lgn;
18172 struct passwd *pw;
18173 ...
18174 if ((lgn = getlogin()) == NULL || (pw = getpwnam(lgn)) == NULL) {
18175     fprintf(stderr, "Get of user information failed.\n"); exit(1);
18176 }

```

APPLICATION USAGE

Three names associated with the current process can be determined: *getpwuid(geteuid())* shall return the name associated with the effective user ID of the process; *getlogin()* shall return the name associated with the current login activity; and *getpwuid(getuid())* shall return the name associated with the real user ID of the process.

The *getlogin_r()* function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

The *getlogin()* function returns a pointer to the user's login name. The same user ID may be shared by several login names. If it is desired to get the user database entry that is used during login, the result of *getlogin()* should be used to provide the argument to the *getpwnam()* function. (This might be used to determine the user's login shell, particularly where a single user has multiple login shells with distinct login names, but the same user ID.)

The information provided by the *cuserid()* function, which was originally defined in the POSIX.1-1988 standard and subsequently removed, can be obtained by the following:

```
getpwuid(geteuid())
```

while the information provided by historical implementations of *cuserid()* can be obtained by:

```
getpwuid(getuid())
```

The thread-safe version of this function places the user name in a user-supplied buffer and returns a non-zero value if it fails. The non-thread-safe version may return the name in a static data area that may be overwritten by each call.

FUTURE DIRECTIONS

None.

SEE ALSO

getpwnam(), *getpwuid()*, *geteuid()*, *getuid()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from System V Release 2.0.

18205

Issue 5

18206

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

18207

18208

The *getlogin_r()* function is included for alignment with the POSIX Threads Extension.

18209

A note indicating that the *getlogin()* function need not be reentrant is added to the DESCRIPTION.

18210

18211

Issue 6

18212

The *getlogin_r()* function is marked as part of the Thread-Safe Functions option.

18213

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

18214

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

18215

18216

- In the RETURN VALUE section, the requirement to set *errno* on error is added.

18217

- The [EMFILE], [ENFILE], and [ENXIO] optional error conditions are added.

18218

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

18219

18220

Issue 7

18221

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

18222

The *getlogin_r()* function is moved from the Thread-Safe Functions option to the Base.

DRAFT

18223 **NAME**18224 getmsg, getpmsg — receive next message from a STREAMS file (**STREAMS**)18225 **SYNOPSIS**

```

18226 OB XSR #include <stropts.h>
18227
18227 int getmsg(int fildes, struct strbuf *restrict ctlptr,
18228           struct strbuf *restrict dataptr, int *restrict flagsp);
18229 int getpmsg(int fildes, struct strbuf *restrict ctlptr,
18230           struct strbuf *restrict dataptr, int *restrict bandp,
18231           int *restrict flagsp);

```

18232 **DESCRIPTION**

18233 The *getmsg()* function shall retrieve the contents of a message located at the head of the
 18234 STREAM head read queue associated with a STREAMS file and place the contents into one or
 18235 more buffers. The message contains either a data part, a control part, or both. The data and
 18236 control parts of the message shall be placed into separate buffers, as described below. The
 18237 semantics of each part are defined by the originator of the message.

18238 The *getpmsg()* function shall be equivalent to *getmsg()*, except that it provides finer control over
 18239 the priority of the messages received. Except where noted, all requirements on *getmsg()* also
 18240 pertain to *getpmsg()*.

18241 The *fildes* argument specifies a file descriptor referencing a STREAMS-based file.

18242 The *ctlptr* and *dataptr* arguments each point to a **strbuf** structure, in which the *buf* member points
 18243 to a buffer in which the data or control information is to be placed, and the *maxlen* member
 18244 indicates the maximum number of bytes this buffer can hold. On return, the *len* member shall
 18245 contain the number of bytes of data or control information actually received. The *len* member
 18246 shall be set to 0 if there is a zero-length control or data part and *len* shall be set to -1 if no data or
 18247 control information is present in the message.

18248 When *getmsg()* is called, *flagsp* should point to an integer that indicates the type of message the
 18249 process is able to receive. This is described further below.

18250 The *ctlptr* argument is used to hold the control part of the message, and *dataptr* is used to hold
 18251 the data part of the message. If *ctlptr* (or *dataptr*) is a null pointer or the *maxlen* member is -1, the
 18252 control (or data) part of the message shall not be processed and shall be left on the STREAM
 18253 head read queue, and if the *ctlptr* (or *dataptr*) is not a null pointer, *len* shall be set to -1. If the
 18254 *maxlen* member is set to 0 and there is a zero-length control (or data) part, that zero-length part
 18255 shall be removed from the read queue and *len* shall be set to 0. If the *maxlen* member is set to 0
 18256 and there are more than 0 bytes of control (or data) information, that information shall be left on
 18257 the read queue and *len* shall be set to 0. If the *maxlen* member in *ctlptr* (or *dataptr*) is less than the
 18258 control (or data) part of the message, *maxlen* bytes shall be retrieved. In this case, the remainder
 18259 of the message shall be left on the STREAM head read queue and a non-zero return value shall
 18260 be provided.

18261 By default, *getmsg()* shall process the first available message on the STREAM head read queue.
 18262 However, a process may choose to retrieve only high-priority messages by setting the integer
 18263 pointed to by *flagsp* to RS_HIPRI. In this case, *getmsg()* shall only process the next message if it is
 18264 a high-priority message. When the integer pointed to by *flagsp* is 0, any available message shall
 18265 be retrieved. In this case, on return, the integer pointed to by *flagsp* shall be set to RS_HIPRI if a
 18266 high-priority message was retrieved, or 0 otherwise.

18267 For *getpmsg()*, the flags are different. The *flagsp* argument points to a bitmask with the following
 18268 mutually-exclusive flags defined: MSG_HIPRI, MSG_BAND, and MSG_ANY. Like *getmsg()*,

18269 *getpmsg()* shall process the first available message on the STREAM head read queue. A process
 18270 may choose to retrieve only high-priority messages by setting the integer pointed to by *flagsp* to
 18271 MSG_HIPRI and the integer pointed to by *bandp* to 0. In this case, *getpmsg()* shall only process
 18272 the next message if it is a high-priority message. In a similar manner, a process may choose to
 18273 retrieve a message from a particular priority band by setting the integer pointed to by *flagsp* to
 18274 MSG_BAND and the integer pointed to by *bandp* to the priority band of interest. In this case,
 18275 *getpmsg()* shall only process the next message if it is in a priority band equal to, or greater than,
 18276 the integer pointed to by *bandp*, or if it is a high-priority message. If a process wants to get the
 18277 first message off the queue, the integer pointed to by *flagsp* should be set to MSG_ANY and the
 18278 integer pointed to by *bandp* should be set to 0. On return, if the message retrieved was a high-
 18279 priority message, the integer pointed to by *flagsp* shall be set to MSG_HIPRI and the integer
 18280 pointed to by *bandp* shall be set to 0. Otherwise, the integer pointed to by *flagsp* shall be set to
 18281 MSG_BAND and the integer pointed to by *bandp* shall be set to the priority band of the message.

18282 If O_NONBLOCK is not set, *getmsg()* and *getpmsg()* shall block until a message of the type
 18283 specified by *flagsp* is available at the front of the STREAM head read queue. If O_NONBLOCK is
 18284 set and a message of the specified type is not present at the front of the read queue, *getmsg()* and
 18285 *getpmsg()* shall fail and set *errno* to [EAGAIN].

18286 If a hangup occurs on the STREAM from which messages are retrieved, *getmsg()* and *getpmsg()*
 18287 shall continue to operate normally, as described above, until the STREAM head read queue is
 18288 empty. Thereafter, they shall return 0 in the *len* members of *ctlptr* and *dataptr*.

18289 RETURN VALUE

18290 Upon successful completion, *getmsg()* and *getpmsg()* shall return a non-negative value. A value
 18291 of 0 indicates that a full message was read successfully. A return value of MORECTL indicates
 18292 that more control information is waiting for retrieval. A return value of MOREDATA indicates
 18293 that more data is waiting for retrieval. A return value of the bitwise-logical OR of MORECTL
 18294 and MOREDATA indicates that both types of information remain. Subsequent *getmsg()* and
 18295 *getpmsg()* calls shall retrieve the remainder of the message. However, if a message of higher
 18296 priority has come in on the STREAM head read queue, the next call to *getmsg()* or *getpmsg()*
 18297 shall retrieve that higher-priority message before retrieving the remainder of the previous
 18298 message.

18299 If the high priority control part of the message is consumed, the message shall be placed back on
 18300 the queue as a normal message of band 0. Subsequent *getmsg()* and *getpmsg()* calls shall retrieve
 18301 the remainder of the message. If, however, a priority message arrives or already exists on the
 18302 STREAM head, the subsequent call to *getmsg()* or *getpmsg()* shall retrieve the higher-priority
 18303 message before retrieving the remainder of the message that was put back.

18304 Upon failure, *getmsg()* and *getpmsg()* shall return -1 and set *errno* to indicate the error.

18305 ERRORS

18306 The *getmsg()* and *getpmsg()* functions shall fail if:

- | | | |
|-------|-----------|--|
| 18307 | [EAGAIN] | The O_NONBLOCK flag is set and no messages are available. |
| 18308 | [EBADF] | The <i>fildev</i> argument is not a valid file descriptor open for reading. |
| 18309 | [EBADMSG] | The queued message to be read is not valid for <i>getmsg()</i> or <i>getpmsg()</i> or a pending file descriptor is at the STREAM head. |
| 18310 | | |
| 18311 | [EINTR] | A signal was caught during <i>getmsg()</i> or <i>getpmsg()</i> . |
| 18312 | [EINVAL] | An illegal value was specified by <i>flagsp</i> , or the STREAM or multiplexer referenced by <i>fildev</i> is linked (directly or indirectly) downstream from a multiplexer. |
| 18313 | | |
| 18314 | | |

18315 [ENOSTR] A STREAM is not associated with *files*.

18316 In addition, *getmsg()* and *getpmsg()* shall fail if the STREAM head had processed an
 18317 asynchronous error before the call. In this case, the value of *errno* does not reflect the result of
 18318 *getmsg()* or *getpmsg()* but reflects the prior error.

18319 EXAMPLES

18320 Getting Any Message

18321 In the following example, the value of *fd* is assumed to refer to an open STREAMS file. The call
 18322 to *getmsg()* retrieves any available message on the associated STREAM-head read queue,
 18323 returning control and data information to the buffers pointed to by *ctrlbuf* and *databuf*,
 18324 respectively.

```
18325 #include <stropts.h>
18326 ...
18327 int fd;
18328 char ctrlbuf[128];
18329 char databuf[512];
18330 struct strbuf ctrl;
18331 struct strbuf data;
18332 int flags = 0;
18333 int ret;

18334 ctrl.buf = ctrlbuf;
18335 ctrl.maxlen = sizeof(ctrlbuf);

18336 data.buf = databuf;
18337 data.maxlen = sizeof(databuf);
18338 ret = getmsg (fd, &ctrl, &data, &flags);
```

18339 Getting the First Message off the Queue

18340 In the following example, the call to *getpmsg()* retrieves the first available message on the
 18341 associated STREAM-head read queue.

```
18342 #include <stropts.h>
18343 ...
18344 int fd;
18345 char ctrlbuf[128];
18346 char databuf[512];
18347 struct strbuf ctrl;
18348 struct strbuf data;
18349 int band = 0;
18350 int flags = MSG_ANY;
18351 int ret;

18352 ctrl.buf = ctrlbuf;
18353 ctrl.maxlen = sizeof(ctrlbuf);

18354 data.buf = databuf;
18355 data.maxlen = sizeof(databuf);
18356 ret = getpmsg (fd, &ctrl, &data, &band, &flags);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

The *getmsg()* and *getpmsg()* functions may be removed in a future version.

SEE ALSO

Section 2.6 (on page 38), *poll()*, *putmsg()*, *read()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

A paragraph regarding “high-priority control parts of messages” is added to the RETURN

DRAFT

18378 **NAME**

18379 getnameinfo — get name information

18380 **SYNOPSIS**

18381 #include <sys/socket.h>

18382 #include <netdb.h>

```
18383 int getnameinfo(const struct sockaddr *restrict sa, socklen_t salen,
18384 char *restrict node, socklen_t nodelen, char *restrict service,
18385 socklen_t servicelen, int flags);
```

18386 **DESCRIPTION**

18387 The *getnameinfo()* function shall translate a socket address to a node name and service location, all of which are defined as in *getaddrinfo()*.

18389 The *sa* argument points to a socket address structure to be translated.

18390 IP6 If the socket address structure contains an IPv4-mapped IPv6 address or an IPv4-compatible IPv6 address, the implementation shall extract the embedded IPv4 address and lookup the node name for that IPv4 address.

18393 If the address is the IPv6 unspecified address ("::"), a lookup shall not be performed and the behavior shall be the same as when the node's name cannot be located.

18395 If the *node* argument is non-NULL and the *nodelen* argument is non-zero, then the *node* argument points to a buffer able to contain up to *nodelen* characters that receives the node name as a null-terminated string. If the *node* argument is NULL or the *nodelen* argument is zero, the node name shall not be returned. If the node's name cannot be located, the numeric form of the address contained in the socket address structure pointed to by the *sa* argument is returned instead of its name.

18401 If the *service* argument is non-NULL and the *servicelen* argument is non-zero, then the *service* argument points to a buffer able to contain up to *servicelen* bytes that receives the service name as a null-terminated string. If the *service* argument is NULL or the *servicelen* argument is zero, the service name shall not be returned. If the service's name cannot be located, the numeric form of the service address (for example, its port number) shall be returned instead of its name.

18406 The *flags* argument is a flag that changes the default actions of the function. By default the fully-qualified domain name (FQDN) for the host shall be returned, but:

- 18408 • If the flag bit NI_NOFQDN is set, only the node name portion of the FQDN shall be returned for local hosts.
- 18409
- 18410 • If the flag bit NI_NUMERICHOST is set, the numeric form of the address contained in the socket address structure pointed to by the *sa* argument shall be returned instead of its name.
- 18411
- 18412
- 18413 • If the flag bit NI_NAMEREQD is set, an error shall be returned if the host's name cannot be located.
- 18414
- 18415 • If the flag bit NI_NUMERICSERV is set, the numeric form of the service address shall be returned (for example, its port number) instead of its name.
- 18416
- 18417 • If the flag bit NI_NUMERICSERVICE is set, the numeric form of the service address shall be returned (for example, its port number) instead of its name.
- 18418
- 18419
- 18419 • If the flag bit NI_NUMERICSERVICE is set, the numeric form of the scope identifier shall be returned (for example, interface index) instead of its name. This flag shall be ignored if the *sa* argument is not an IPv6 address.

- 18420 • If the flag bit NI_DGRAM is set, this indicates that the service is a datagram service
 18421 (SOCK_DGRAM). The default behavior shall assume that the service is a stream service
 18422 (SOCK_STREAM).

18423 **Notes:**

- 18424 1. The two NI_NUMERICxxx flags are required to support the `-n` flag that many
 18425 commands provide.
- 18426 2. The NI_DGRAM flag is required for the few AF_INET and AF_INET6 port numbers (for
 18427 example, [512,514]) that represent different services for UDP and TCP.

18428 The `getnameinfo()` function shall be thread-safe.

18429 **RETURN VALUE**

18430 A zero return value for `getnameinfo()` indicates successful completion; a non-zero return value
 18431 indicates failure. The possible values for the failures are listed in the ERRORS section.

18432 Upon successful completion, `getnameinfo()` shall return the *node* and *service* names, if requested,
 18433 in the buffers provided. The returned names are always null-terminated strings.

18434 **ERRORS**

18435 The `getnameinfo()` function shall fail and return the corresponding value if:

18436 [EAI_AGAIN] The name could not be resolved at this time. Future attempts may succeed.

18437 [EAI_BADFLAGS]

18438 The *flags* had an invalid value.

18439 [EAI_FAIL] A non-recoverable error occurred.

18440 [EAI_FAMILY] The address family was not recognized or the address length was invalid for
 18441 the specified family.

18442 [EAI_MEMORY] There was a memory allocation failure.

18443 [EAI_NONAME] The name does not resolve for the supplied parameters.

18444 NI_NAMEREQD is set and the host's name cannot be located, or both
 18445 *nodename* and *servname* were null.

18446 [EAI_OVERFLOW]

18447 An argument buffer overflowed. The buffer pointed to by the *node* argument
 18448 or the *service* argument was too small.

18449 [EAI_SYSTEM] A system error occurred. The error code can be found in *errno*.

18450 **EXAMPLES**

18451 None.

18452 **APPLICATION USAGE**

18453 If the returned values are to be used as part of any further name resolution (for example, passed
 18454 to `getaddrinfo()`), applications should provide buffers large enough to store any result possible on
 18455 the system.

18456 Given the IPv4-mapped IPv6 address "`::ffff:1.2.3.4`", the implementation performs a
 18457 lookup as if the socket address structure contains the IPv4 address "`1.2.3.4`".

18458 The IPv6 unspecified address ("`:::`") and the IPv6 loopback address ("`:::1`") are not
 18459 IPv4-compatible addresses.

getnameinfo()18460
18461
18462
18463
18464
18465
18466
18467
18468
18469
18470
18471
18472
18473
18474
18475
18476
18477**RATIONALE**

None.

FUTURE DIRECTIONS

None.

SEE ALSO

gai_strerror(), *getaddrinfo()*, *getservbyname()*, *inet_ntop()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The **restrict** keyword is added to the *getnameinfo()* prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/23 is applied, making various changes in the SYNOPSIS and DESCRIPTION for alignment with IPv6.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/24 is applied, adding the [EAL_OVERFLOW] error to the ERRORS section.

Issue 7

SD5-XSH-ERN-127 is applied, clarifying the behavior if the address is the IPv6 unspecified address.

18478 **NAME**
18479 `getnetbyaddr, getnetbyname, getnetent` — network database functions

18480 **SYNOPSIS**
18481 `#include <netdb.h>`
18482 `struct netent *getnetbyaddr(uint32_t net, int type);`
18483 `struct netent *getnetbyname(const char *name);`
18484 `struct netent *getnetent(void);`

18485 **DESCRIPTION**
18486 Refer to *endnetent()*.



18487 **NAME**

18488 getopt, optarg, opterr, optind, optopt — command option parsing

18489 **SYNOPSIS**

18490 #include <unistd.h>

18491 int getopt(int argc, char * const argv[], const char *optstring);

18492 extern char *optarg;

18493 extern int optind, opterr, optopt;

18494 **DESCRIPTION**18495 The *getopt()* function is a command-line parser that shall follow Utility Syntax Guidelines 3, 4, 5,
18496 6, 7, 9, and 10 in the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax
18497 Guidelines.18498 The parameters *argc* and *argv* are the argument count and argument array as passed to *main()*
18499 (see *exec*). The argument *optstring* is a string of recognized option characters; if a character is
18500 followed by a colon, the option takes an argument. All option characters allowed by Utility
18501 Syntax Guideline 3 are allowed in *optstring*. The implementation may accept other characters as
18502 an extension.18503 The variable *optind* is the index of the next element of the *argv[]* vector to be processed. It shall
18504 be initialized to 1 by the system, and *getopt()* shall update it when it finishes with each element
18505 of *argv[]*. When an element of *argv[]* contains multiple option characters, it is unspecified how
18506 *getopt()* determines which options have already been processed.18507 The *getopt()* function shall return the next option character (if one is found) from *argv* that
18508 matches a character in *optstring*, if there is one that matches. If the option takes an argument,
18509 *getopt()* shall set the variable *optarg* to point to the option-argument as follows:

- 18510 1. If the option was the last character in the string pointed to by an element of
- argv*
- , then
-
- 18511
- optarg*
- shall contain the next element of
- argv*
- , and
- optind*
- shall be incremented by 2. If the
-
- 18512 resulting value of
- optind*
- is greater than
- argc*
- , this indicates a missing option-argument,
-
- 18513 and
- getopt()*
- shall return an error indication.
-
- 18514 2. Otherwise,
- optarg*
- shall point to the string following the option character in that element
-
- 18515 of
- argv*
- , and
- optind*
- shall be incremented by 1.

18516 If, when *getopt()* is called:18517 *argv[optind]* is a null pointer
18518 **argv[optind]* is not the character '-'
18519 *argv[optind]* points to the string "--"18520 *getopt()* shall return -1 without changing *optind*. If:18521 *argv[optind]* points to the string "--"18522 *getopt()* shall return -1 after incrementing *optind*.18523 If *getopt()* encounters an option character that is not contained in *optstring*, it shall return the
18524 question-mark ('?') character. If it detects a missing option-argument, it shall return the colon
18525 character (':') if the first character of *optstring* was a colon, or a question-mark character ('?')
18526 otherwise. In either case, *getopt()* shall set the variable *optopt* to the option character that caused
18527 the error. If the application has not set the variable *opterr* to 0 and the first character of *optstring* is
18528 not a colon, *getopt()* shall also print a diagnostic message to *stderr* in the format specified for the
18529 *getopts* utility.18530 The *getopt()* function need not be thread-safe. A function that is not required to be thread-safe is

18531 not required to be reentrant.

18532 RETURN VALUE

18533 The *getopt()* function shall return the next option character specified on the command line.

18534 A colon (':') shall be returned if *getopt()* detects a missing argument and the first character of
18535 *optstring* was a colon (':').

18536 A question mark ('?') shall be returned if *getopt()* encounters an option character not in
18537 *optstring* or detects a missing argument and the first character of *optstring* was not a colon (':').

18538 Otherwise, *getopt()* shall return -1 when all command line options are parsed.

18539 ERRORS

18540 No errors are defined.

18541 EXAMPLES

18542 Parsing Command Line Options

18543 The following code fragment shows how you might process the arguments for a utility that can
18544 take the mutually-exclusive options *a* and *b* and the options *f* and *o*, both of which require
18545 arguments:

```
18546 #include <unistd.h>
18547
18548 int
18549 main(int argc, char *argv[ ])
18550 {
18551     int c;
18552     int bflg, aflag, errflg;
18553     char *ifile;
18554     char *ofile;
18555     extern char *optarg;
18556     extern int optind, optopt;
18557     . . .
18558     while ((c = getopt(argc, argv, ":abf:o:")) != -1) {
18559         switch(c) {
18560             case 'a':
18561                 if (bflg)
18562                     errflg++;
18563                 else
18564                     aflag++;
18565                 break;
18566             case 'b':
18567                 if (aflag)
18568                     errflg++;
18569                 else {
18570                     bflg++;
18571                     bproc();
18572                 }
18573                 break;
18574             case 'f':
18575                 ifile = optarg;
18576                 break;
18577             case 'o':
18578                 ofile = optarg;
18579                 break;
18580             case '::':
18581                 /* -f or -o without operand */
```

```

18580             fprintf(stderr,
18581                 "Option -%c requires an operand\n", optopt);
18582             errflg++;
18583             break;
18584         case '?':
18585             fprintf(stderr,
18586                 "Unrecognized option: -%c\n", optopt);
18587             errflg++;
18588         }
18589     }
18590     if (errflg) {
18591         fprintf(stderr, "usage: . . . ");
18592         exit(2);
18593     }
18594     for ( ; optind < argc; optind++) {
18595         if (access(argv[optind], R_OK)) {
18596             . . .
18597         }

```

This code accepts any of the following as equivalent:

```

18599 cmd -ao arg path path
18600 cmd -a -o arg path path
18601 cmd -o arg -a path path
18602 cmd -a -o arg -- path path
18603 cmd -a -oarg path path
18604 cmd -aoarg path path

```

Checking Options and Arguments

The following example parses a set of command line options and prints messages to standard output for each option and argument that it encounters.

```

18608 #include <unistd.h>
18609 #include <stdio.h>
18610 ...
18611 int c;
18612 char *filename;
18613 extern char *optarg;
18614 extern int optind, optopt, opterr;
18615 ...
18616 while ((c = getopt(argc, argv, ":abf:")) != -1) {
18617     switch(c) {
18618     case 'a':
18619         printf("a is set\n");
18620         break;
18621     case 'b':
18622         printf("b is set\n");
18623         break;
18624     case 'f':
18625         filename = optarg;
18626         printf("filename is %s\n", filename);
18627         break;
18628     case ':':
18629         printf("-%c without filename\n", optopt);
18630         break;

```

```

18631         case '?':
18632             printf("unknown arg %c\n", optopt);
18633             break;
18634         }
18635     }

```

18636 Selecting Options from the Command Line

18637 The following example selects the type of database routines the user wants to use based on the
 18638 *Options* argument.

```

18639 #include <unistd.h>
18640 #include <string.h>
18641 ...
18642 char *Options = "hdbt1";
18643 ...
18644 int dbtype, i;
18645 char c;
18646 char *st;
18647 ...
18648 dbtype = 0;
18649 while ((c = getopt(argc, argv, Options)) != -1) {
18650     if ((st = strchr(Options, c)) != NULL) {
18651         dbtype = st - Options;
18652         break;
18653     }
18654 }

```

18655 APPLICATION USAGE

18656 The *getopt()* function is only required to support option characters included in Utility Syntax
 18657 Guideline 3. Many historical implementations of *getopt()* support other characters as options.
 18658 This is an allowed extension, but applications that use extensions are not maximally portable.
 18659 Note that support for multi-byte option characters is only possible when such characters can be
 18660 represented as type **int**.

18661 RATIONALE

18662 The *optopt* variable represents historical practice and allows the application to obtain the identity
 18663 of the invalid option.

18664 The description has been written to make it clear that *getopt()*, like the *getopts* utility, deals with
 18665 option-arguments whether separated from the option by <blank>s or not. Note that the
 18666 requirements on *getopt()* and *getopts* are more stringent than the Utility Syntax Guidelines.

18667 The *getopt()* function shall return -1 , rather than EOF, so that **<stdio.h>** is not required.

18668 The special significance of a colon as the first character of *optstring* makes *getopt()* consistent
 18669 with the *getopts* utility. It allows an application to make a distinction between a missing
 18670 argument and an incorrect option letter without having to examine the option letter. It is true
 18671 that a missing argument can only be detected in one case, but that is a case that has to be
 18672 considered.

18673 FUTURE DIRECTIONS

18674 None.

getopt()

18675

SEE ALSO

18676

exec, the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x

18677

18678

CHANGE HISTORY

18679

First released in Issue 1. Derived from Issue 1 of the SVID.

18680

Issue 5

18681

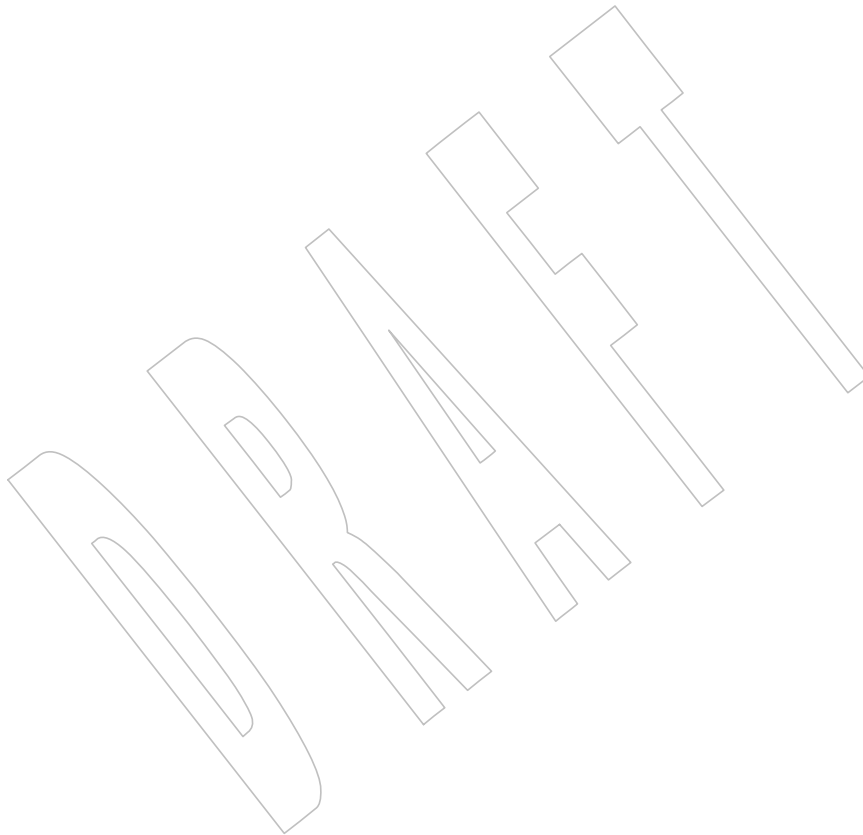
A note indicating that the *getopt()* function need not be reentrant is added to the DESCRIPTION.

18682

Issue 6

18683

IEEE PASC Interpretation 1003.2 #150 is applied.



18684 **NAME**18685 `getpeername` — get the name of the peer socket18686 **SYNOPSIS**

```
18687 #include <sys/socket.h>
18688
18688 int getpeername(int socket, struct sockaddr *restrict address,
18689               socklen_t *restrict address_len);
```

18690 **DESCRIPTION**

18691 The `getpeername()` function shall retrieve the peer address of the specified socket, store this
 18692 address in the **sockaddr** structure pointed to by the `address` argument, and store the length of this
 18693 address in the object pointed to by the `address_len` argument.

18694 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
 18695 the stored address shall be truncated.

18696 If the protocol permits connections by unbound clients, and the peer is not bound, then the
 18697 value stored in the object pointed to by `address` is unspecified.

18698 **RETURN VALUE**

18699 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and `errno` set to
 18700 indicate the error.

18701 **ERRORS**

18702 The `getpeername()` function shall fail if:

- 18703 [EBADF] The `socket` argument is not a valid file descriptor.
- 18704 [EINVAL] The socket has been shut down.
- 18705 [ENOTCONN] The socket is not connected or otherwise has not had the peer pre-specified.
- 18706 [ENOTSOCK] The `socket` argument does not refer to a socket.
- 18707 [EOPNOTSUPP] The operation is not supported for the socket protocol.

18708 The `getpeername()` function may fail if:

- 18709 [ENOBUFS] Insufficient resources were available in the system to complete the call.

18710 **EXAMPLES**

18711 None.

18712 **APPLICATION USAGE**

18713 None.

18714 **RATIONALE**

18715 None.

18716 **FUTURE DIRECTIONS**

18717 None.

18718 **SEE ALSO**

18719 [*accept\(\)*](#), [*bind\(\)*](#), [*getsockname\(\)*](#), [*socket\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x,
 18720 [*<sys/socket.h>*](#)

18721

18722

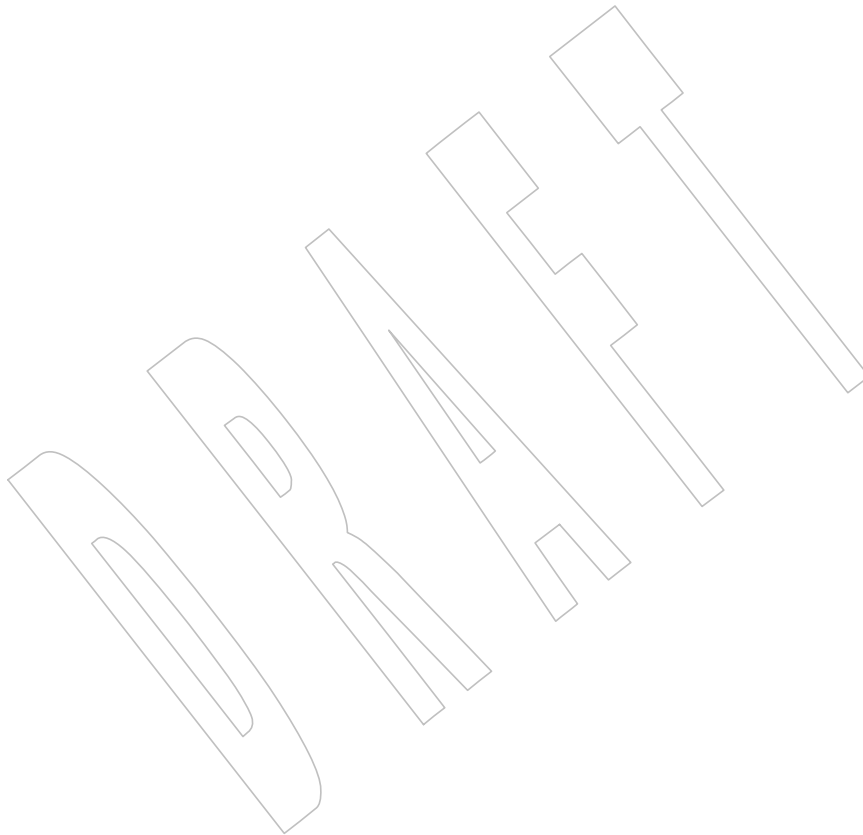
18723

18724

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The **restrict** keyword is added to the *getpeername()* prototype for alignment with the ISO/IEC 9899:1999 standard.



18725 **NAME**

18726 getpgid — get the process group ID for a process

18727 **SYNOPSIS**

18728 #include <unistd.h>

18729 pid_t getpgid(pid_t pid);

18730 **DESCRIPTION**18731 The *getpgid()* function shall return the process group ID of the process whose process ID is equal
18732 to *pid*. If *pid* is equal to 0, *getpgid()* shall return the process group ID of the calling process.18733 **RETURN VALUE**18734 Upon successful completion, *getpgid()* shall return a process group ID. Otherwise, it shall return
18735 (**pid_t**)-1 and set *errno* to indicate the error.18736 **ERRORS**18737 The *getpgid()* function shall fail if:18738 [EPERM] The process whose process ID is equal to *pid* is not in the same session as the
18739 calling process, and the implementation does not allow access to the process
18740 group ID of that process from the calling process.18741 [ESRCH] There is no process with a process ID equal to *pid*.18742 The *getpgid()* function may fail if:18743 [EINVAL] The value of the *pid* argument is invalid.18744 **EXAMPLES**

18745 None.

18746 **APPLICATION USAGE**

18747 None.

18748 **RATIONALE**

18749 None.

18750 **FUTURE DIRECTIONS**

18751 None.

18752 **SEE ALSO**18753 *exec*, *fork()*, *getpgrp()*, *getpid()*, *getsid()*, *setpgid()*, *setsid()*, the Base Definitions volume of
18754 IEEE Std 1003.1-200x, <unistd.h>18755 **CHANGE HISTORY**

18756 First released in Issue 4, Version 2.

18757 **Issue 5**

18758 Moved from X/OPEN UNIX extension to BASE.

18759 **Issue 7**18760 The *getpgid()* function is moved from the XSI option to the Base.

18761 **NAME**
 18762 `getpgrp` — get the process group ID of the calling process

18763 **SYNOPSIS**
 18764 `#include <unistd.h>`
 18765 `pid_t getpgrp(void);`

18766 **DESCRIPTION**
 18767 The `getpgrp()` function shall return the process group ID of the calling process.

18768 **RETURN VALUE**
 18769 The `getpgrp()` function shall always be successful and no return value is reserved to indicate an
 18770 error.

18771 **ERRORS**
 18772 No errors are defined.

18773 **EXAMPLES**
 18774 None.

18775 **APPLICATION USAGE**
 18776 None.

18777 **RATIONALE**
 18778 4.3 BSD provides a `getpgrp()` function that returns the process group ID for a specified process.
 18779 Although this function supports job control, all known job control shells always specify the
 18780 calling process with this function. Thus, the simpler System V `getpgrp()` suffices, and the added
 18781 complexity of the 4.3 BSD `getpgrp()` is provided by the XSI extension `getpgid()`.

18782 **FUTURE DIRECTIONS**
 18783 None.

18784 **SEE ALSO**
 18785 `exec`, `fork()`, `getpgid()`, `getpid()`, `getppid()`, `kill()`, `setpgid()`, `setsid()`, the Base Definitions volume of
 18786 IEEE Std 1003.1-200x, `<sys/types.h>`, `<unistd.h>`

18787 **CHANGE HISTORY**
 18788 First released in Issue 1. Derived from Issue 1 of the SVID.

18789 **Issue 6**
 18790 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

18791 The following new requirements on POSIX implementations derive from alignment with the
 18792 Single UNIX Specification:

- 18793 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
 18794 required for conforming implementations of previous POSIX specifications, it was not
 18795 required for UNIX applications.

18796 **NAME**

18797 getpid — get the process ID

18798 **SYNOPSIS**

18799 #include <unistd.h>

18800 pid_t getpid(void);

18801 **DESCRIPTION**18802 The *getpid()* function shall return the process ID of the calling process.18803 **RETURN VALUE**18804 The *getpid()* function shall always be successful and no return value is reserved to indicate an
18805 error.18806 **ERRORS**

18807 No errors are defined.

18808 **EXAMPLES**

18809 None.

18810 **APPLICATION USAGE**

18811 None.

18812 **RATIONALE**

18813 None.

18814 **FUTURE DIRECTIONS**

18815 None.

18816 **SEE ALSO**18817 *exec*, *fork()*, *getpgrp()*, *getppid()*, *kill()*, *mkdtemp()*, *setpgid()*, *setsid()*, the Base Definitions volume
18818 of IEEE Std 1003.1-200x, <sys/types.h>, <unistd.h>18819 **CHANGE HISTORY**

18820 First released in Issue 1. Derived from Issue 1 of the SVID.

18821 **Issue 6**

18822 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

18823 The following new requirements on POSIX implementations derive from alignment with the
18824 Single UNIX Specification:

- 18825
- The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
18826 required for conforming implementations of previous POSIX specifications, it was not
18827 required for UNIX applications.

18828 **NAME**
18829 getpmsg — receive next message from a STREAMS file

18830 **SYNOPSIS**

```
18831 OB XSI #include <stropts.h>  
18832 int getpmsg(int fildev, struct strbuf *restrict ctlptr,  
18833 struct strbuf *restrict dataptr, int *restrict bandp,  
18834 int *restrict flagsp);
```

18835 **DESCRIPTION**

18836 Refer to [getmsg\(\)](#).

18837 **NAME**

18838 getppid — get the parent process ID

18839 **SYNOPSIS**18840 #include <unistd.h>
18841 pid_t getppid(void);18842 **DESCRIPTION**18843 The *getppid()* function shall return the parent process ID of the calling process.18844 **RETURN VALUE**18845 The *getppid()* function shall always be successful and no return value is reserved to indicate an
18846 error.18847 **ERRORS**

18848 No errors are defined.

18849 **EXAMPLES**

18850 None.

18851 **APPLICATION USAGE**

18852 None.

18853 **RATIONALE**

18854 None.

18855 **FUTURE DIRECTIONS**

18856 None.

18857 **SEE ALSO**18858 *exec*, *fork()*, *getpgid()*, *getpgrp()*, *getpid()*, *kill()*, *setpgid()*, *setsid()*, the Base Definitions volume of
18859 IEEE Std 1003.1-200x, <sys/types.h>, <unistd.h>18860 **CHANGE HISTORY**

18861 First released in Issue 1. Derived from Issue 1 of the SVID.

18862 **Issue 6**

18863 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

18864 The following new requirements on POSIX implementations derive from alignment with the
18865 Single UNIX Specification:

- 18866
- The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
18867 required for conforming implementations of previous POSIX specifications, it was not
18868 required for UNIX applications.

18869 **NAME**18870 `getpriority, setpriority` — get and set the nice value18871 **SYNOPSIS**

```
18872 XSI      #include <sys/resource.h>
18873
18873      int getpriority(int which, id_t who);
18874      int setpriority(int which, id_t who, int value);
```

18875 **DESCRIPTION**

18876 The `getpriority()` function shall obtain the nice value of a process, process group, or user. The
 18877 `setpriority()` function shall set the nice value of a process, process group, or user to
 18878 `value+{NZERO}`.

18879 Target processes are specified by the values of the `which` and `who` arguments. The `which`
 18880 argument may be one of the following values: `PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`,
 18881 indicating that the `who` argument is to be interpreted as a process ID, a process group ID, or an
 18882 effective user ID, respectively. A 0 value for the `who` argument specifies the current process,
 18883 process group, or user.

18884 The nice value set with `setpriority()` shall be applied to the process. If the process is multi-
 18885 threaded, the nice value shall affect all system scope threads in the process.

18886 If more than one process is specified, `getpriority()` shall return value `{NZERO}` less than the
 18887 lowest nice value pertaining to any of the specified processes, and `setpriority()` shall set the nice
 18888 values of all of the specified processes to `value+{NZERO}`.

18889 The default nice value is `{NZERO}`; lower nice values shall cause more favorable scheduling.
 18890 While the range of valid nice values is `[0,{NZERO}*2-1]`, implementations may enforce more
 18891 restrictive limits. If `value+{NZERO}` is less than the system's lowest supported nice value,
 18892 `setpriority()` shall set the nice value to the lowest supported value; if `value+{NZERO}` is greater
 18893 than the system's highest supported nice value, `setpriority()` shall set the nice value to the
 18894 highest supported value.

18895 Only a process with appropriate privileges can lower its nice value.

18896 PS|TPS Any processes or threads using `SCHED_FIFO` or `SCHED_RR` shall be unaffected by a call to
 18897 `setpriority()`. This is not considered an error. A process which subsequently reverts to
 18898 `SCHED_OTHER` need not have its priority affected by such a `setpriority()` call.

18899 The effect of changing the nice value may vary depending on the process-scheduling algorithm
 18900 in effect.

18901 Since `getpriority()` can return the value `-1` on successful completion, it is necessary to set `errno` to
 18902 0 prior to a call to `getpriority()`. If `getpriority()` returns the value `-1`, then `errno` can be checked to
 18903 see if an error occurred or if the value is a legitimate nice value.

18904 **RETURN VALUE**

18905 Upon successful completion, `getpriority()` shall return an integer in the range `-{NZERO}` to
 18906 `{NZERO}-1`. Otherwise, `-1` shall be returned and `errno` set to indicate the error.

18907 Upon successful completion, `setpriority()` shall return 0; otherwise, `-1` shall be returned and `errno`
 18908 set to indicate the error.

18909 ERRORS

18910 The *getpriority()* and *setpriority()* functions shall fail if:

18911 [ESRCH] No process could be located using the *which* and *who* argument values
18912 specified.

18913 [EINVAL] The value of the *which* argument was not recognized, or the value of the *who*
18914 argument is not a valid process ID, process group ID, or user ID.

18915 In addition, *setpriority()* may fail if:

18916 [EPERM] A process was located, but neither the real nor effective user ID of the
18917 executing process match the effective user ID of the process whose nice value
18918 is being changed.

18919 [EACCES] A request was made to change the nice value to a lower numeric value and the
18920 current process does not have appropriate privileges.

18921 EXAMPLES**18922 Using *getpriority()***

18923 The following example returns the current scheduling priority for the process ID returned by the
18924 call to *getpid()*.

```
18925 #include <sys/resource.h>
18926 ...
18927 int which = PRIO_PROCESS;
18928 id_t pid;
18929 int ret;

18930 pid = getpid();
18931 ret = getpriority(which, pid);
```

18932 Using *setpriority()*

18933 The following example sets the priority for the current process ID to -20.

```
18934 #include <sys/resource.h>
18935 ...
18936 int which = PRIO_PROCESS;
18937 id_t pid;
18938 int priority = -20;
18939 int ret;

18940 pid = getpid();
18941 ret = setpriority(which, pid, priority);
```

18942 APPLICATION USAGE

18943 The *getpriority()* and *setpriority()* functions work with an offset nice value (nice value
18944 $-\{\text{NZERO}\}$). The nice value is in the range $[0, 2*\{\text{NZERO}\} - 1]$, while the return value for
18945 *getpriority()* and the third parameter for *setpriority()* are in the range $[-\{\text{NZERO}\}, \{\text{NZERO}\} - 1]$.

18946 RATIONALE

18947 None.

18948 FUTURE DIRECTIONS

18949 None.

getpriority()

18950

SEE ALSO

18951

nice(), *sched_get_priority_max()*, *sched_setscheduler()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**sys/resource.h**>

18952

18953

CHANGE HISTORY

18954

First released in Issue 4, Version 2.

18955

Issue 5

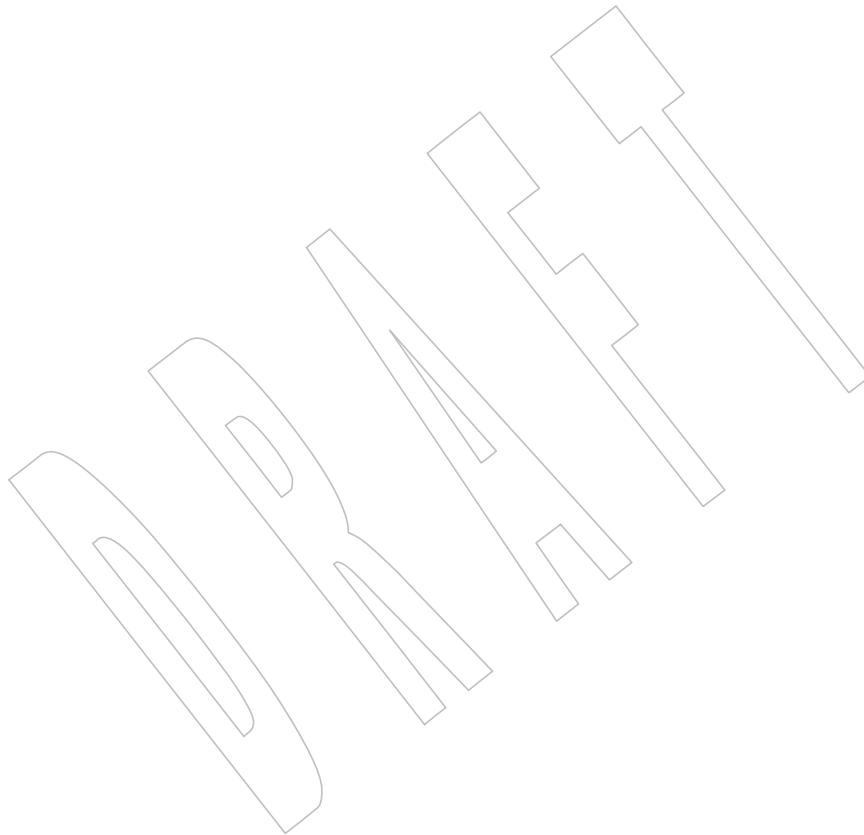
18956

Moved from X/OPEN UNIX extension to BASE.

18957

The DESCRIPTION is reworded in terms of the nice value rather than *priority* to avoid confusion with functionality in the POSIX Realtime Extension.

18958



18959 **NAME**
18960 `getprotobyname, getprotobynumber, getprotent` — network protocol database functions

18961 **SYNOPSIS**
18962 `#include <netdb.h>`
18963 `struct protoent *getprotobyname(const char *name);`
18964 `struct protoent *getprotobynumber(int proto);`
18965 `struct protoent *getprotoent(void);`

18966 **DESCRIPTION**
18967 Refer to *endprotoent()*.



getpwent()

18968 **NAME**
18969 getpwent — get user database entry

18970 **SYNOPSIS**

18971 XSI #include <pwd.h>
18972 struct passwd *getpwent(void);

18973 **DESCRIPTION**

18974 Refer to *endpwent()*.

18975 **NAME**
 18976 `getpwnam, getpwnam_r` — search user database for a name

18977 **SYNOPSIS**
 18978 `#include <pwd.h>`
 18979 `struct passwd *getpwnam(const char *name);`
 18980 `int getpwnam_r(const char *name, struct passwd *pwd, char *buffer,`
 18981 `size_t bufsize, struct passwd **result);`

18982 **DESCRIPTION**
 18983 The `getpwnam()` function shall search the user database for an entry with a matching *name*.
 18984 The `getpwnam()` function need not be thread-safe. A function that is not required to be thread-
 18985 safe is not required to be reentrant.

18986 Applications wishing to check for error situations should set *errno* to 0 before calling
 18987 `getpwnam()`. If `getpwnam()` returns a null pointer and *errno* is non-zero, an error occurred.

18988 The `getpwnam_r()` function shall update the **passwd** structure pointed to by *pwd* and store a
 18989 pointer to that structure at the location pointed to by *result*. The structure shall contain an entry
 18990 from the user database with a matching *name*. Storage referenced by the structure is allocated
 18991 from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The
 18992 maximum size needed for this buffer can be determined with the `{_SC_GETPW_R_SIZE_MAX}`
 18993 `sysconf()` parameter. A NULL pointer shall be returned at the location pointed to by *result* on
 18994 error or if the requested entry is not found.

18995 **RETURN VALUE**
 18996 The `getpwnam()` function shall return a pointer to a **struct passwd** with the structure as defined
 18997 in `<pwd.h>` with a matching entry if found. A null pointer shall be returned if the requested
 18998 entry is not found, or an error occurs. On error, *errno* shall be set to indicate the error.

18999 The return value may point to a static area which is overwritten by a subsequent call to
 19000 `getpwent()`, `getpwnam()`, or `getpwuid()`.

19001 The `getpwnam_r()` function shall return zero on success or if the requested entry was not found
 19002 and no error has occurred. If an error has occurred, an error number shall be returned to indicate
 19003 the error.

19004 **ERRORS**
 19005 The `getpwnam()` and `getpwnam_r()` functions may fail if:
 19006 [EIO] An I/O error has occurred.
 19007 [EINTR] A signal was caught during `getpwnam()`.
 19008 [EMFILE] All file descriptors available to the process are currently open.
 19009 [ENFILE] The maximum allowable number of files is currently open in the system.
 19010 The `getpwnam_r()` function may fail if:
 19011 [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be
 19012 referenced by the resulting **passwd** structure.

EXAMPLES**Getting an Entry for the Login Name**

The following example uses the `getlogin()` function to return the name of the user who logged in; this information is passed to the `getpwnam()` function to get the user database entry for that user.

```

19017 #include <sys/types.h>
19018 #include <pwd.h>
19019 #include <unistd.h>
19020 #include <stdio.h>
19021 #include <stdlib.h>
19022 ...
19023 char *lgn;
19024 struct passwd *pw;
19025 ...
19026 if ((lgn = getlogin()) == NULL || (pw = getpwnam(lgn)) == NULL) {
19027     fprintf(stderr, "Get of user information failed.\n"); exit(1);
19028 }
19029 ...

```

APPLICATION USAGE

Three names associated with the current process can be determined: `getpwuid(geteuid())` returns the name associated with the effective user ID of the process; `getlogin()` returns the name associated with the current login activity; and `getpwuid(getuid())` returns the name associated with the real user ID of the process.

The `getpwnam_r()` function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`getpwuid()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<limits.h>`, `<pwd.h>`, `<sys/types.h>`

CHANGE HISTORY

First released in Issue 1. Derived from System V Release 2.0.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

The `getpwnam_r()` function is included for alignment with the POSIX Threads Extension.

A note indicating that the `getpwnam()` function need not be reentrant is added to the DESCRIPTION.

Issue 6

The `getpwnam_r()` function is marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION describing matching the *name*.

In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

19058
19059
19060
19061
19062
19063
19064
19065
19066
19067
19068
19069
19070
19071

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the RETURN VALUE section, the requirement to set *errno* on error is added.
- The [EMFILE], [ENFILE], and [ENXIO] optional error conditions are added.

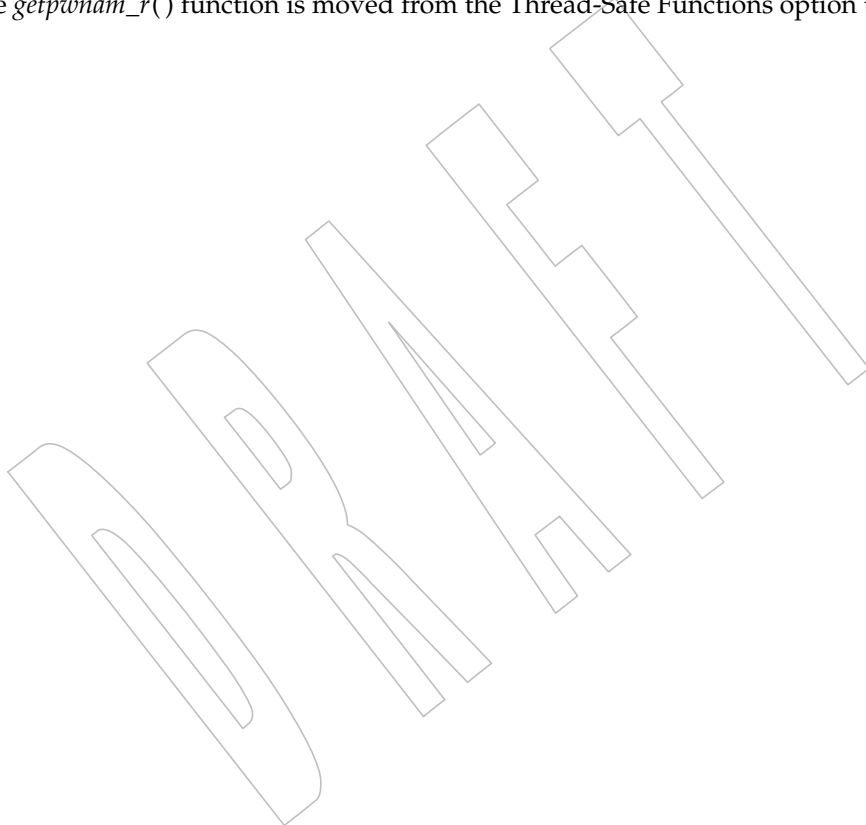
The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the buffer from *bufsize* characters to bytes.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

The `getpwnam_r()` function is moved from the Thread-Safe Functions option to the Base.



19072 **NAME**19073 `getpwuid, getpwuid_r` — search user database for a user ID19074 **SYNOPSIS**19075 `#include <pwd.h>`19076 `struct passwd *getpwuid(uid_t uid);`19077 `int getpwuid_r(uid_t uid, struct passwd *pwd, char *buffer,`19078 `size_t bufsize, struct passwd **result);`19079 **DESCRIPTION**19080 The `getpwuid()` function shall search the user database for an entry with a matching `uid`.19081 The `getpwuid()` function need not be thread-safe. A function that is not required to be thread-safe
19082 is not required to be reentrant.19083 Applications wishing to check for error situations should set `errno` to 0 before calling `getpwuid()`.
19084 If `getpwuid()` returns a null pointer and `errno` is set to non-zero, an error occurred.19085 The `getpwuid_r()` function shall update the **passwd** structure pointed to by `pwd` and store a
19086 pointer to that structure at the location pointed to by `result`. The structure shall contain an entry
19087 from the user database with a matching `uid`. Storage referenced by the structure is allocated
19088 from the memory provided with the `buffer` parameter, which is `bufsize` bytes in size. The
19089 maximum size needed for this buffer can be determined with the `{_SC_GETPW_R_SIZE_MAX}`
19090 `sysconf()` parameter. A NULL pointer shall be returned at the location pointed to by `result` on
19091 error or if the requested entry is not found.19092 **RETURN VALUE**19093 The `getpwuid()` function shall return a pointer to a **struct passwd** with the structure as defined in
19094 `<pwd.h>` with a matching entry if found. A null pointer shall be returned if the requested entry
19095 is not found, or an error occurs. On error, `errno` shall be set to indicate the error.19096 The return value may point to a static area which is overwritten by a subsequent call to
19097 `getpwent()`, `getpwnam()`, or `getpwuid()`.19098 If successful, the `getpwuid_r()` function shall return zero; otherwise, an error number shall be
19099 returned to indicate the error.19100 **ERRORS**19101 The `getpwuid()` and `getpwuid_r()` functions may fail if:

19102 [EIO] An I/O error has occurred.

19103 [EINTR] A signal was caught during `getpwuid()`.

19104 [EMFILE] All file descriptors available to the process are currently open.

19105 [ENFILE] The maximum allowable number of files is currently open in the system.

19106 The `getpwuid_r()` function may fail if:19107 [ERANGE] Insufficient storage was supplied via `buffer` and `bufsize` to contain the data to be
19108 referenced by the resulting **passwd** structure.

EXAMPLES

Getting an Entry for the Root User

The following example gets the user database entry for the user with user ID 0 (root).

```

19112 #include <sys/types.h>
19113 #include <pwd.h>
19114 ...
19115 uid_t id = 0;
19116 struct passwd *pwd;
19117
19118 pwd = getpwuid(id);

```

Finding the Name for the Effective User ID

The following example defines *pws* as a pointer to a structure of type **passwd**, which is used to store the structure pointer returned by the call to the *getpwuid()* function. The *geteuid()* function shall return the effective user ID of the calling process; this is used as the search criteria for the *getpwuid()* function. The call to *getpwuid()* shall return a pointer to the structure containing that user ID value.

```

19124 #include <unistd.h>
19125 #include <sys/types.h>
19126 #include <pwd.h>
19127 ...
19128 struct passwd *pws;
19129 pws = getpwuid(geteuid());

```

Finding an Entry in the User Database

The following example uses *getpwuid()* to search the user database for a user ID that was previously stored in a **stat** structure, then prints out the user name if it is found. If the user is not found, the program prints the numeric value of the user ID for the entry.

```

19134 #include <sys/types.h>
19135 #include <pwd.h>
19136 #include <stdio.h>
19137 ...
19138 struct stat statbuf;
19139 struct passwd *pwd;
19140 ...
19141 if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
19142     printf(" %-8.8s", pwd->pw_name);
19143 else
19144     printf(" %-8d", statbuf.st_uid);

```

APPLICATION USAGE

Three names associated with the current process can be determined: *getpwuid(geteuid())* returns the name associated with the effective user ID of the process; *getlogin()* returns the name associated with the current login activity; and *getpwuid(getuid())* returns the name associated with the real user ID of the process.

The *getpwuid_r()* function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

getpwuid()

19152
19153
19154
19155
19156
19157
19158
19159
19160
19161
19162
19163
19164
19165
19166
19167
19168
19169
19170
19171
19172
19173
19174
19175
19176
19177
19178
19179
19180
19181
19182
19183
19184
19185
19186

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getpwnam(), *geteuid()*, *getuid()*, *getlogin()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<limits.h>`, `<pwd.h>`, `<sys/types.h>`

CHANGE HISTORY

First released in Issue 1. Derived from System V Release 2.0.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

The *getpwuid_r()* function is included for alignment with the POSIX Threads Extension.

A note indicating that the *getpwuid()* function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *getpwuid_r()* function is marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION describing matching the *uid*.

In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the RETURN VALUE section, the requirement to set *errno* on error is added.
- The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the buffer from *bufsize* characters to bytes.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

The *getpwuid_r()* function is moved from the Thread-Safe Functions option to the Base.

19187 **NAME**

19188 getrlimit, setrlimit — control maximum resource consumption

19189 **SYNOPSIS**

```
19190 XSI #include <sys/resource.h>
19191 int getrlimit(int resource, struct rlimit *rlp);
19192 int setrlimit(int resource, const struct rlimit *rlp);
```

19193 **DESCRIPTION**

19194 The *getrlimit()* function shall get, and the *setrlimit()* function shall set, limits on the consumption
19195 of a variety of resources.

19196 Each call to either *getrlimit()* or *setrlimit()* identifies a specific resource to be operated upon as
19197 well as a resource limit. A resource limit is represented by an **rlimit** structure. The *rlim_cur*
19198 member specifies the current or soft limit and the *rlim_max* member specifies the maximum or
19199 hard limit. Soft limits may be changed by a process to any value that is less than or equal to the
19200 hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or
19201 equal to the soft limit. Only a process with appropriate privileges can raise a hard limit. Both
19202 hard and soft limits can be changed in a single call to *setrlimit()* subject to the constraints
19203 described above.

19204 The value RLIM_INFINITY, defined in **<sys/resource.h>**, shall be considered to be larger than
19205 any other limit value. If a call to *getrlimit()* returns RLIM_INFINITY for a resource, it means the
19206 implementation shall not enforce limits on that resource. Specifying RLIM_INFINITY as any
19207 resource limit value on a successful call to *setrlimit()* shall inhibit enforcement of that resource
19208 limit.

19209 The following resources are defined:

19210	RLIMIT_CORE	This is the maximum size of a core file, in bytes, that may be created by a process. A limit of 0 shall prevent the creation of a core file. If this limit is exceeded, the writing of a core file shall terminate at this size.
19211		
19212		
19213	RLIMIT_CPU	This is the maximum amount of CPU time, in seconds, used by a process. If this limit is exceeded, SIGXCPU shall be generated for the process. If the process is catching or ignoring SIGXCPU, or all threads belonging to that process are blocking SIGXCPU, the behavior is unspecified.
19214		
19215		
19216		
19217	RLIMIT_DATA	This is the maximum size of a data segment of the process, in bytes. If this limit is exceeded, the <i>malloc()</i> function shall fail with <i>errno</i> set to [ENOMEM].
19218		
19219		
19220	RLIMIT_FSIZE	This is the maximum size of a file, in bytes, that may be created by a process. If a write or truncate operation would cause this limit to be exceeded, SIGXFSZ shall be generated for the thread. If the thread is blocking, or the process is catching or ignoring SIGXFSZ, continued attempts to increase the size of a file from end-of-file to beyond the limit shall fail with <i>errno</i> set to [EFBIG].
19221		
19222		
19223		
19224		
19225		
19226	RLIMIT_NOFILE	This is a number one greater than the maximum value that the system may assign to a newly-created descriptor. If this limit is exceeded, functions that allocate a file descriptor shall fail with <i>errno</i> set to [EMFILE]. This limit constrains the number of file descriptors that a process may allocate.
19227		
19228		
19229		
19230		

19231 RLIMIT_STACK This is the maximum size of the initial thread's stack, in bytes. The
19232 implementation does not automatically grow the stack beyond this limit.
19233 If this limit is exceeded, SIGSEGV shall be generated for the thread. If the
19234 thread is blocking SIGSEGV, or the process is ignoring or catching
19235 SIGSEGV and has not made arrangements to use an alternate stack, the
19236 disposition of SIGSEGV shall be set to SIG_DFL before it is generated.

19237 RLIMIT_AS This is the maximum size of total available memory of the process, in
19238 bytes. If this limit is exceeded, the *malloc()* and *mmap()* functions shall fail
19239 with *errno* set to [ENOMEM]. In addition, the automatic stack growth
19240 fails with the effects outlined above.

19241 When using the *getrlimit()* function, if a resource limit can be represented correctly in an object
19242 of type **rlim_t**, then its representation is returned; otherwise, if the value of the resource limit is
19243 equal to that of the corresponding saved hard limit, the value returned shall be
19244 RLIM_SAVED_MAX; otherwise, the value returned shall be RLIM_SAVED_CUR.

19245 When using the *setrlimit()* function, if the requested new limit is RLIM_INFINITY, the new limit
19246 shall be "no limit"; otherwise, if the requested new limit is RLIM_SAVED_MAX, the new limit
19247 shall be the corresponding saved hard limit; otherwise, if the requested new limit is
19248 RLIM_SAVED_CUR, the new limit shall be the corresponding saved soft limit; otherwise, the
19249 new limit shall be the requested value. In addition, if the corresponding saved limit can be
19250 represented correctly in an object of type **rlim_t** then it shall be overwritten with the new limit.

19251 The result of setting a limit to RLIM_SAVED_MAX or RLIM_SAVED_CUR is unspecified unless
19252 a previous call to *getrlimit()* returned that value as the soft or hard limit for the corresponding
19253 resource limit.

19254 The determination of whether a limit can be correctly represented in an object of type **rlim_t** is
19255 implementation-defined. For example, some implementations permit a limit whose value is
19256 greater than RLIM_INFINITY and others do not.

19257 The *exec* family of functions shall cause resource limits to be saved.

19258 **RETURN VALUE**

19259 Upon successful completion, *getrlimit()* and *setrlimit()* shall return 0. Otherwise, these functions
19260 shall return -1 and set *errno* to indicate the error.

19261 **ERRORS**

19262 The *getrlimit()* and *setrlimit()* functions shall fail if:

19263 [EINVAL] An invalid *resource* was specified; or in a *setrlimit()* call, the new *rlim_cur*
19264 exceeds the new *rlim_max*.

19265 [EPERM] The limit specified to *setrlimit()* would have raised the maximum limit value,
19266 and the calling process does not have appropriate privileges.

19267 The *setrlimit()* function may fail if:

19268 [EINVAL] The limit specified cannot be lowered because current usage is already higher
19269 than the limit.

EXAMPLES

None.

APPLICATION USAGE

If a process attempts to set the hard limit or soft limit for RLIMIT_NOFILE to less than the value of `{_POSIX_OPEN_MAX}` from `<limits.h>`, unexpected behavior may occur.

If a process attempts to set the hard limit or soft limit for RLIMIT_NOFILE to less than the highest currently open file descriptor +1, unexpected behavior may occur.

RATIONALE

It should be noted that RLIMIT_STACK applies “at least” to the stack of the initial thread in the process, and not to the sum of all the stacks in the process, as that would be very limiting unless the value is so big as to provide no value at all with a single thread.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fork()*, *malloc()*, *open()*, *sigaltstack()*, *sysconf()*, *ulimit()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<stropts.h>`, `<sys/resource.h>`

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

An APPLICATION USAGE section is added.

Large File Summit extensions are added.

Issue 6

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/25 is applied, changing wording for RLIMIT_NOFILE in the DESCRIPTION related to functions that allocate a file descriptor failing with [EMFILE]. Text is added to the APPLICATION USAGE section noting the consequences of a process attempting to set the hard or soft limit for RLIMIT_NOFILE less than the highest currently open file descriptor +1.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/46 is applied, updating the definition of RLIMIT_STACK in the DESCRIPTION section from “the maximum size of a process stack” to “the maximum size of the initial thread’s stack”. Text is added to the RATIONALE section.

19301 **NAME**
 19302 getrusage — get information about resource utilization

19303 **SYNOPSIS**

19304 XSI `#include <sys/resource.h>`
 19305 `int getrusage(int who, struct rusage *r_usage);`

19306 **DESCRIPTION**

19307 The *getrusage()* function shall provide measures of the resources used by the current process or
 19308 its terminated and waited-for child processes. If the value of the *who* argument is
 19309 RUSAGE_SELF, information shall be returned about resources used by the current process. If the
 19310 value of the *who* argument is RUSAGE_CHILDREN, information shall be returned about
 19311 resources used by the terminated and waited-for children of the current process. If the child is
 19312 never waited for (for example, if the parent has SA_NOCLDWAIT set or sets SIGCHLD to
 19313 SIG_IGN), the resource information for the child process is discarded and not included in the
 19314 resource information provided by *getrusage()*.

19315 The *r_usage* argument is a pointer to an object of type **struct rusage** in which the returned
 19316 information is stored.

19317 **RETURN VALUE**

19318 Upon successful completion, *getrusage()* shall return 0; otherwise, -1 shall be returned and *errno*
 19319 set to indicate the error.

19320 **ERRORS**

19321 The *getrusage()* function shall fail if:
 19322 [EINVAL] The value of the *who* argument is not valid.

19323 **EXAMPLES**

19324 **Using getrusage()**

19325 The following example returns information about the resources used by the current process.

19326 `#include <sys/resource.h>`
 19327 `...`
 19328 `int who = RUSAGE_SELF;`
 19329 `struct rusage usage;`
 19330 `int ret;`
 19331 `ret = getrusage(who, &usage);`

19332 **APPLICATION USAGE**

19333 None.

19334 **RATIONALE**

19335 None.

19336 **FUTURE DIRECTIONS**

19337 None.

19338 **SEE ALSO**

19339 *exit()*, *sigaction()*, *time()*, *times()*, *wait()*, the Base Definitions volume of IEEE Std 1003.1-200x,
 19340 `<sys/resource.h>`

19341

CHANGE HISTORY

19342

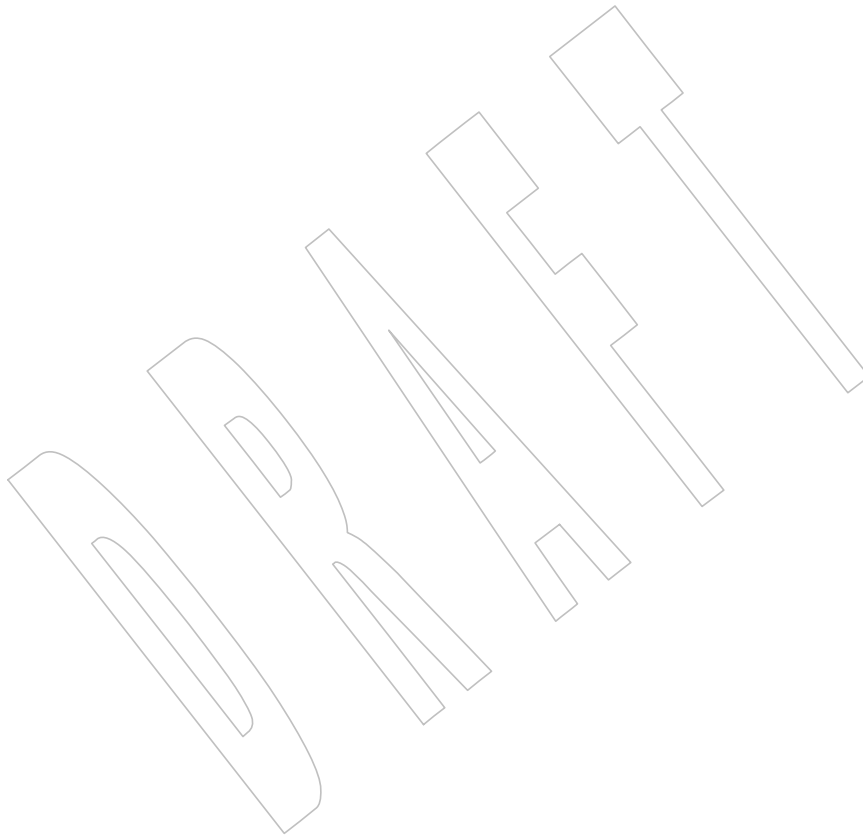
First released in Issue 4, Version 2.

19343

Issue 5

19344

Moved from X/OPEN UNIX extension to BASE.



19345 **NAME**

19346 gets — get a string from a stdin stream

19347 **SYNOPSIS**

19348 #include <stdio.h>

19349 char *gets(char *s);

19350 **DESCRIPTION**

19351 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 19352 conflict between the requirements described here and the ISO C standard is unintentional. This
 19353 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

19354 The *gets()* function shall read bytes from the standard input stream, *stdin*, into the array pointed
 19355 to by *s*, until a <newline> is read or an end-of-file condition is encountered. Any <newline> shall
 19356 be discarded and a null byte shall be placed immediately after the last byte read into the array.

19357 CX The *gets()* function may mark the *st_atime* field of the file associated with *stream* for update. The
 19358 *st_atime* field shall be marked for update by the first successful execution of *fgetc()*, *fgets()*,
 19359 *fread()*, *getc()*, *getchar()*, *gets()*, *fscanf()*, or *scanf()* using *stream* that returns data not supplied by
 19360 a prior call to *ungetc()*.

19361 **RETURN VALUE**

19362 Upon successful completion, *gets()* shall return *s*. If the end-of-file indicator for the stream is set,
 19363 or if the stream is at end-of-file, the end-of-file indicator for the stream shall be set and *gets()*
 19364 shall return a null pointer. If a read error occurs, the error indicator for the stream shall be set,
 19365 CX *gets()* shall return a null pointer, and set *errno* to indicate the error.

19366 **ERRORS**19367 Refer to *fgetc()*.19368 **EXAMPLES**

19369 None.

19370 **APPLICATION USAGE**

19371 Reading a line that overflows the array pointed to by *s* results in undefined behavior. The use of
 19372 *fgets()* is recommended.

19373 Since the user cannot specify the length of the buffer passed to *gets()*, use of this function is
 19374 discouraged. The length of the string read is unlimited. It is possible to overflow this buffer in
 19375 such a way as to cause applications to fail, or possible system security violations.

19376 It is recommended that the *fgets()* function should be used to read input lines.

19377 **RATIONALE**

19378 None.

19379 **FUTURE DIRECTIONS**

19380 None.

19381 **SEE ALSO**19382 *feof()*, *ferror()*, *fgets()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>19383 **CHANGE HISTORY**

19384 First released in Issue 1. Derived from Issue 1 of the SVID.

19385

Issue 6

19386

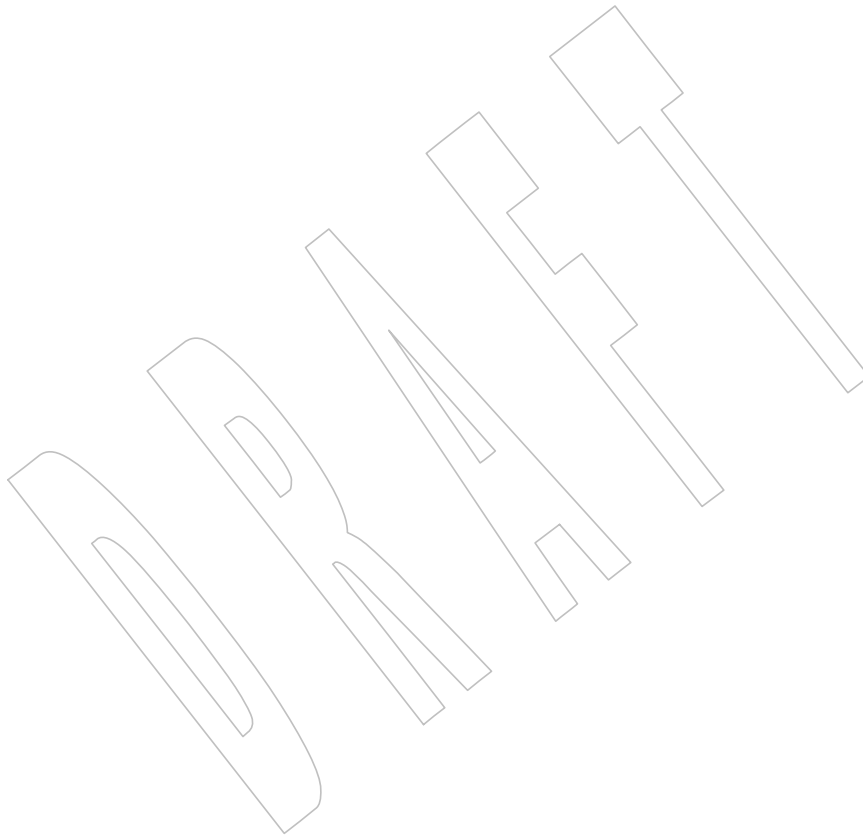
Extensions beyond the ISO C standard are marked.

19387

Issue 7

19388

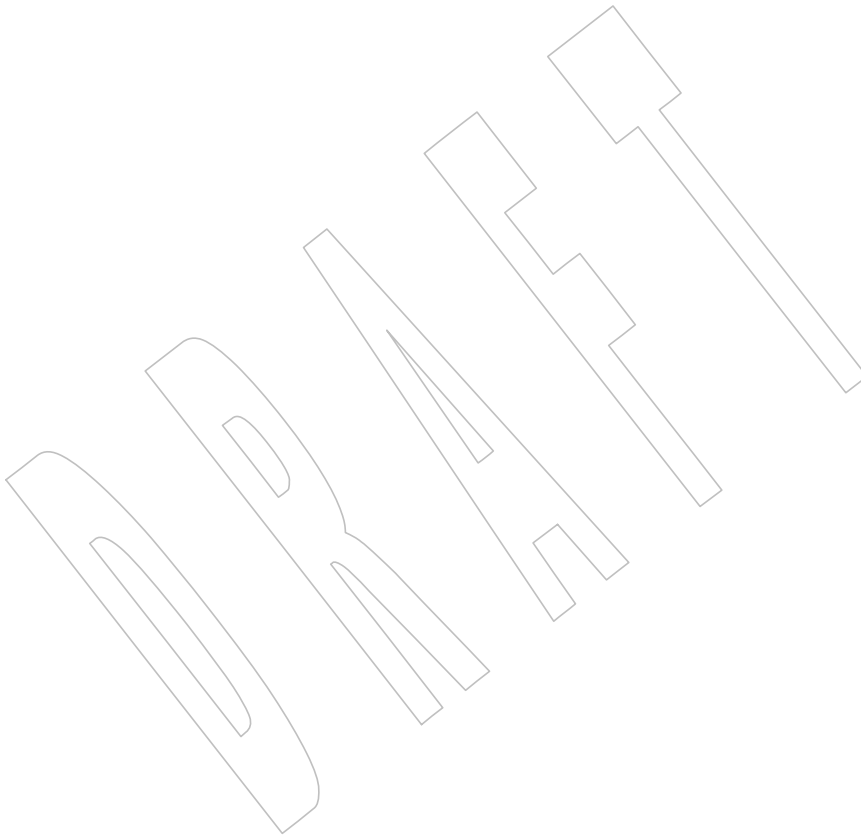
Austin Group Interpretation 1003.1-2001 #051 is applied, clarifying the RETURN VALUE section.



19389 **NAME**
19390 getservbyname, getservbyport, getservent — network services database functions

19391 **SYNOPSIS**
19392 #include <netdb.h>
19393 struct servent *getservbyname(const char *name, const char *proto);
19394 struct servent *getservbyport(int port, const char *proto);
19395 struct servent *getservent(void);

19396 **DESCRIPTION**
19397 Refer to *endservent()*.



19398 **NAME**

19399 *getsid* — get the process group ID of a session leader

19400 **SYNOPSIS**

19401 #include <unistd.h>

19402 pid_t getsid(pid_t *pid*);

19403 **DESCRIPTION**

19404 The *getsid()* function shall obtain the process group ID of the process that is the session leader of

19405 the process specified by *pid*. If *pid* is (**pid_t**)0, it specifies the calling process.

19406 **RETURN VALUE**

19407 Upon successful completion, *getsid()* shall return the process group ID of the session leader of

19408 the specified process. Otherwise, it shall return (**pid_t**)-1 and set *errno* to indicate the error.

19409 **ERRORS**

19410 The *getsid()* function shall fail if:

19411 [EPERM] The process specified by *pid* is not in the same session as the calling process,

19412 and the implementation does not allow access to the process group ID of the

19413 session leader of that process from the calling process.

19414 [ESRCH] There is no process with a process ID equal to *pid*.

19415 **EXAMPLES**

19416 None.

19417 **APPLICATION USAGE**

19418 None.

19419 **RATIONALE**

19420 None.

19421 **FUTURE DIRECTIONS**

19422 None.

19423 **SEE ALSO**

19424 *exec*, *fork()*, *getpid()*, *getpgid()*, *setpgid()*, *setsid()*, the Base Definitions volume of

19425 IEEE Std 1003.1-200x, <unistd.h>

19426 **CHANGE HISTORY**

19427 First released in Issue 4, Version 2.

19428 **Issue 5**

19429 Moved from X/OPEN UNIX extension to BASE.

19430 **Issue 7**

19431 The *getsid()* function is moved from the XSI option to the Base.

19432 **NAME**19433 `getsockname` — get the socket name19434 **SYNOPSIS**

```
19435 #include <sys/socket.h>
19436
19437 int getsockname(int socket, struct sockaddr *restrict address,
19438                socklen_t *restrict address_len);
```

19438 **DESCRIPTION**

19439 The `getsockname()` function shall retrieve the locally-bound name of the specified socket, store
 19440 this address in the **sockaddr** structure pointed to by the `address` argument, and store the length of
 19441 this address in the object pointed to by the `address_len` argument.

19442 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
 19443 the stored address shall be truncated.

19444 If the socket has not been bound to a local name, the value stored in the object pointed to by
 19445 `address` is unspecified.

19446 **RETURN VALUE**

19447 Upon successful completion, 0 shall be returned, the `address` argument shall point to the address
 19448 of the socket, and the `address_len` argument shall point to the length of the address. Otherwise, -1
 19449 shall be returned and `errno` set to indicate the error.

19450 **ERRORS**

19451 The `getsockname()` function shall fail if:

19452 [EBADF] The `socket` argument is not a valid file descriptor.

19453 [ENOTSOCK] The `socket` argument does not refer to a socket.

19454 [EOPNOTSUPP] The operation is not supported for this socket's protocol.

19455 The `getsockname()` function may fail if:

19456 [EINVAL] The socket has been shut down.

19457 [ENOBUFS] Insufficient resources were available in the system to complete the function.

19458 **EXAMPLES**

19459 None.

19460 **APPLICATION USAGE**

19461 None.

19462 **RATIONALE**

19463 None.

19464 **FUTURE DIRECTIONS**

19465 None.

19466 **SEE ALSO**

19467 [accept\(\)](#), [bind\(\)](#), [getpeername\(\)](#), [socket\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x,
 19468 [<sys/socket.h>](#)

19469 **CHANGE HISTORY**

19470 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19471 The **restrict** keyword is added to the `getsockname()` prototype for alignment with the
 19472 ISO/IEC 9899:1999 standard.

19473 **NAME**
 19474 `getsockopt` — get the socket options

19475 **SYNOPSIS**
 19476

```
#include <sys/socket.h>
```


 19477

```
int getsockopt(int socket, int level, int option_name,
```


 19478

```
void *restrict option_value, socklen_t *restrict option_len);
```

19479 **DESCRIPTION**
 19480 The `getsockopt()` function manipulates options associated with a socket.

19481 The `getsockopt()` function shall retrieve the value for the option specified by the `option_name`
 19482 argument for the socket specified by the `socket` argument. If the size of the option value is greater
 19483 than `option_len`, the value stored in the object pointed to by the `option_value` argument shall be
 19484 silently truncated. Otherwise, the object pointed to by the `option_len` argument shall be modified
 19485 to indicate the actual length of the value.

19486 The `level` argument specifies the protocol level at which the option resides. To retrieve options at
 19487 the socket level, specify the `level` argument as `SOL_SOCKET`. To retrieve options at other levels,
 19488 supply the appropriate level identifier for the protocol controlling the option. For example, to
 19489 indicate that an option is interpreted by the TCP (Transmission Control Protocol), set `level` to
 19490 `IPPROTO_TCP` as defined in the `<netinet/in.h>` header.

19491 The socket in use may require the process to have appropriate privileges to use the `getsockopt()`
 19492 function.

19493 The `option_name` argument specifies a single option to be retrieved. It can be one of the following
 19494 values defined in `<sys/socket.h>`:

19495 `SO_DEBUG` Reports whether debugging information is being recorded. This option
 19496 shall store an `int` value. This is a Boolean option.

19497 `SO_ACCEPTCONN` Reports whether socket listening is enabled. This option shall store an `int`
 19498 value. This is a Boolean option.

19499 `SO_BROADCAST` Reports whether transmission of broadcast messages is supported, if this
 19500 is supported by the protocol. This option shall store an `int` value. This is a
 19501 Boolean option.

19502 `SO_REUSEADDR` Reports whether the rules used in validating addresses supplied to `bind()`
 19503 should allow reuse of local addresses, if this is supported by the protocol.
 19504 This option shall store an `int` value. This is a Boolean option.

19505 `SO_KEEPALIVE` Reports whether connections are kept active with periodic transmission
 19506 of messages, if this is supported by the protocol.

19507 If the connected socket fails to respond to these messages, the connection
 19508 shall be broken and threads writing to that socket shall be notified with a
 19509 `SIGPIPE` signal. This option shall store an `int` value. This is a Boolean
 19510 option.

19511 `SO_LINGER` Reports whether the socket lingers on `close()` if data is present. If
 19512 `SO_LINGER` is set, the system shall block the calling thread during `close()`
 19513 until it can transmit the data or until the end of the interval indicated by
 19514 the `l_linger` member, whichever comes first. If `SO_LINGER` is not
 19515 specified, and `close()` is issued, the system handles the call in a way that
 19516 allows the calling thread to continue as quickly as possible. This option
 19517 shall store a `linger` structure.

getsockopt()

19518	SO_OOBLIN	Reports whether the socket leaves received out-of-band data (data marked urgent) inline. This option shall store an int value. This is a Boolean option.
19519		
19520		
19521	SO_SNDBUF	Reports send buffer size information. This option shall store an int value.
19522	SO_RCVBUF	Reports receive buffer size information. This option shall store an int value.
19523		
19524	SO_ERROR	Reports information about error status and clears it. This option shall store an int value.
19525		
19526	SO_TYPE	Reports the socket type. This option shall store an int value. Socket types are described in Section 2.10.6 (on page 61).
19527		
19528	SO_DONTROUTE	Reports whether outgoing messages bypass the standard routing facilities. The destination shall be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. The effect, if any, of this option depends on what protocol is in use. This option shall store an int value. This is a Boolean option.
19529		
19530		
19531		
19532		
19533		
19534	SO_RCVLOWAT	Reports the minimum number of bytes to process for socket input operations. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. (They may return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that returned; for example, out-of-band data.) This option shall store an int value. Note that not all implementations allow this option to be retrieved.
19535		
19536		
19537		
19538		
19539		
19540		
19541		
19542		
19543	SO_RCVTIMEO	Reports the timeout value for input operations. This option shall store a timeval structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. If a receive operation has blocked for this much time without receiving additional data, it shall return with a partial count or <i>errno</i> set to [EAGAIN] or [EWOULDBLOCK] if no data was received. The default for this option is zero, which indicates that a receive operation shall not time out. Note that not all implementations allow this option to be retrieved.
19544		
19545		
19546		
19547		
19548		
19549		
19550		
19551	SO_SNDLOWAT	Reports the minimum number of bytes to process for socket output operations. Non-blocking output operations shall process no data if flow control does not allow the smaller of the send low water mark value or the entire request to be processed. This option shall store an int value. Note that not all implementations allow this option to be retrieved.
19552		
19553		
19554		
19555		
19556	SO_SNDTIMEO	Reports the timeout value specifying the amount of time that an output function blocks because flow control prevents data from being sent. If a send operation has blocked for this time, it shall return with a partial count or with <i>errno</i> set to [EAGAIN] or [EWOULDBLOCK] if no data was sent. The default for this option is zero, which indicates that a send operation shall not time out. The option shall store a timeval structure. Note that not all implementations allow this option to be retrieved.
19557		
19558		
19559		
19560		
19561		
19562		
19563		
19564		
		For Boolean options, a zero value indicates that the option is disabled and a non-zero value indicates that the option is enabled.

19565

RETURN VALUE

19566

Upon successful completion, *getsockopt()* shall return 0; otherwise, -1 shall be returned and *errno* set to indicate the error.

19567

19568

ERRORS

19569

The *getsockopt()* function shall fail if:

19570

[EBADF] The *socket* argument is not a valid file descriptor.

19571

[EINVAL] The specified option is invalid at the specified socket level.

19572

[ENOPROTOOPT]

19573

The option is not supported by the protocol.

19574

[ENOTSOCK] The *socket* argument does not refer to a socket.

19575

The *getsockopt()* function may fail if:

19576

[EACCES] The calling process does not have the appropriate privileges.

19577

[EINVAL] The socket has been shut down.

19578

[ENOBUFS] Insufficient resources are available in the system to complete the function.

19579

EXAMPLES

19580

None.

19581

APPLICATION USAGE

19582

None.

19583

RATIONALE

19584

None.

19585

FUTURE DIRECTIONS

19586

None.

19587

SEE ALSO

19588

bind(), *close()*, *endprotoent()*, *setsockopt()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**, **<netinet/in.h>**

19589

19590

CHANGE HISTORY

19591

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19592

The **restrict** keyword is added to the *getsockopt()* prototype for alignment with the ISO/IEC 9899:1999 standard.

19593

19594

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/47 is applied, updating the description of SO_LINGER in the DESCRIPTION so that it blocks the calling thread rather than the process.

19595

19596 **NAME**
 19597 `getsubopt` — parse suboption arguments from a string

19598 **SYNOPSIS**
 19599 `#include <stdlib.h>`

19600 `int getsubopt(char **optionp, char * const *keylistp, char **valuep);`

19601 **DESCRIPTION**

19602 The `getsubopt()` function shall parse suboption arguments in a flag argument. Such options often
 19603 result from the use of `getopt()`.

19604 The `getsubopt()` argument *optionp* is a pointer to a pointer to the option argument string. The
 19605 suboption arguments shall be separated by commas and each may consist of either a single
 19606 token, or a token-value pair separated by an equal sign.

19607 The *keylistp* argument shall be a pointer to a vector of strings. The end of the vector is identified
 19608 by a null pointer. Each entry in the vector is one of the possible tokens that might be found in
 19609 **optionp*. Since commas delimit suboption arguments in *optionp*, they should not appear in any of
 19610 the strings pointed to by *keylistp*. Similarly, because an equal sign separates a token from its
 19611 value, the application should not include an equal sign in any of the strings pointed to by
 19612 *keylistp*.

19613 The *valuep* argument is the address of a value string pointer.

19614 If a comma appears in *optionp*, it shall be interpreted as a suboption separator. After commas
 19615 have been processed, if there are one or more equal signs in a suboption string, the first equal
 19616 sign in any suboption string shall be interpreted as a separator between a token and a value.
 19617 Subsequent equal signs in a suboption string shall be interpreted as part of the value.

19618 If the string at **optionp* contains only one suboption argument (equivalently, no commas),
 19619 `getsubopt()` shall update **optionp* to point to the null character at the end of the string. Otherwise,
 19620 it shall isolate the suboption argument by replacing the comma separator with a null character,
 19621 and shall update **optionp* to point to the start of the next suboption argument. If the suboption
 19622 argument has an associated value (equivalently, contains an equal sign), `getsubopt()` shall update
 19623 **valuep* to point to the value's first character. Otherwise, it shall set **valuep* to a null pointer. The
 19624 calling application may use this information to determine whether the presence or absence of a
 19625 value for the suboption is an error.

19626 Additionally, when `getsubopt()` fails to match the suboption argument with a token in the *keylistp*
 19627 array, the calling application should decide if this is an error, or if the unrecognized option
 19628 should be processed in another way.

19629 **RETURN VALUE**

19630 The `getsubopt()` function shall return the index of the matched token string, or -1 if no token
 19631 strings were matched.

19632 **ERRORS**

19633 No errors are defined.

EXAMPLES

```

19634
19635     #include <stdio.h>
19636     #include <stdlib.h>
19637
19638     int do_all;
19639     const char *type;
19640     int read_size;
19641     int write_size;
19642     int read_only;
19643
19644     enum
19645     {
19646         RO_OPTION = 0,
19647         RW_OPTION,
19648         READ_SIZE_OPTION,
19649         WRITE_SIZE_OPTION
19650     };
19651
19652     const char *mount_opts[] =
19653     {
19654         [RO_OPTION] = "ro",
19655         [RW_OPTION] = "rw",
19656         [READ_SIZE_OPTION] = "rsize",
19657         [WRITE_SIZE_OPTION] = "wsize",
19658         NULL
19659     };
19660
19661     int
19662     main(int argc, char *argv[])
19663     {
19664         char *subopts, *value;
19665         int opt;
19666
19667         while ((opt = getopt(argc, argv, "at:o:")) != -1)
19668             switch(opt)
19669             {
19670                 case 'a':
19671                     do_all = 1;
19672                     break;
19673                 case 't':
19674                     type = optarg;
19675                     break;
19676                 case 'o':
19677                     subopts = optarg;
19678                     while (*subopts != '\0')
19679                         switch(getsubopt(&subopts, mount_opts, &value))
19680                         {
19681                             case RO_OPTION:
19682                                 read_only = 1;
19683                                 break;
19684                             case RW_OPTION:
19685                                 read_only = 0;
19686                                 break;
19687                             case READ_SIZE_OPTION:
19688                                 if (value == NULL)
19689                                     abort();

```

```

19685         read_size = atoi(value);
19686         break;
19687     case WRITE_SIZE_OPTION:
19688         if (value == NULL)
19689             abort();
19690         write_size = atoi(value);
19691         break;
19692     default:
19693         /* Unknown suboption. */
19694         printf("Unknown suboption '%s'\n", value);
19695         break;
19696     }
19697     break;
19698     default:
19699         abort();
19700     }
19701     /* Do the real work. */
19702     return 0;
19703 }

```

19704 Parsing Suboptions

19705 The following example uses the `getsubopt()` function to parse a *value* argument in the *optarg*
 19706 external variable returned by a call to `getopt()`.

```

19707 #include <stdlib.h>
19708 ...
19709 char *tokens[] = {"HOME", "PATH", "LOGNAME", (char *) NULL };
19710 char *value;
19711 int opt, index;
19712 while ((opt = getopt(argc, argv, "e:")) != -1) {
19713     switch(opt) {
19714         case 'e':
19715             while ((index = getsubopt(&optarg, tokens, &value)) != -1) {
19716                 switch(index) {
19717                     ...
19718                 }
19719                 break;
19720             }
19721         }
19722     }
19723     ...

```

19724 APPLICATION USAGE

19725 None.

19726 RATIONALE

19727 None.

19728 FUTURE DIRECTIONS

19729 None.

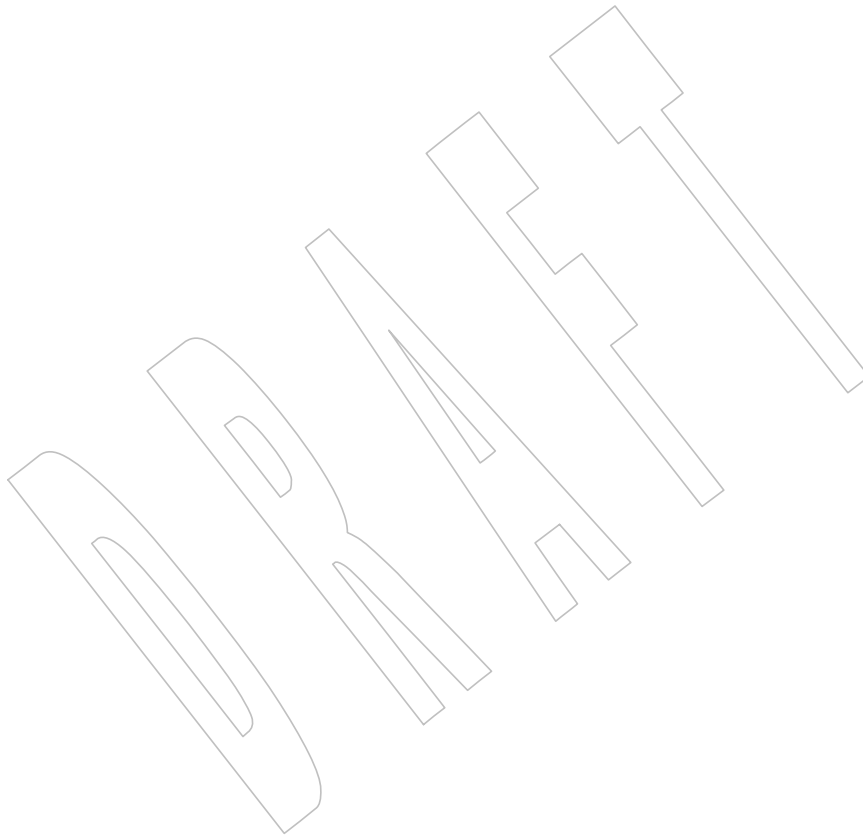
19730 **SEE ALSO**
19731 *getopt()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdlib.h>

19732 **CHANGE HISTORY**
19733 First released in Issue 4, Version 2.

19734 **Issue 5**
19735 Moved from X/OPEN UNIX extension to BASE.

19736 **Issue 6**
19737 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/26 is applied, correcting an editorial error
19738 in the SYNOPSIS.

19739 **Issue 7**
19740 The *getsubopt()* function is moved from the XSI option to the Base.



19741 **NAME**

19742 gettimeofday — get the date and time

19743 **SYNOPSIS**

```
19744 OB XSI #include <sys/time.h>
19745      int gettimeofday(struct timeval *restrict tp, void *restrict tzp);
```

19746 **DESCRIPTION**

19747 The *gettimeofday()* function shall obtain the current time, expressed as seconds and microseconds
 19748 since the Epoch, and store it in the **timeval** structure pointed to by *tp*. The resolution of the
 19749 system clock is unspecified.

19750 If *tzp* is not a null pointer, the behavior is unspecified.

19751 **RETURN VALUE**

19752 The *gettimeofday()* function shall return 0 and no value shall be reserved to indicate an error.

19753 **ERRORS**

19754 No errors are defined.

19755 **EXAMPLES**

19756 None.

19757 **APPLICATION USAGE**

19758 Applications should use the *clock_gettime()* function instead of the obsolescent *gettimeofday()*
 19759 function.

19760 **RATIONALE**

19761 None.

19762 **FUTURE DIRECTIONS**

19763 The *gettimeofday()* function may be removed in a future version.

19764 **SEE ALSO**

19765 *clock_getres()*, *ctime()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/time.h>**

19766 **CHANGE HISTORY**

19767 First released in Issue 4, Version 2.

19768 **Issue 5**

19769 Moved from X/OPEN UNIX extension to BASE.

19770 **Issue 6**

19771 The DESCRIPTION is updated to refer to “seconds since the Epoch” rather than “seconds since
 19772 00:00:00 UTC (Coordinated Universal Time), January 1 1970” for consistency with other *time*
 19773 functions.

19774 The **restrict** keyword is added to the *gettimeofday()* prototype for alignment with the
 19775 ISO/IEC 9899:1999 standard.

19776 **Issue 7**

19777 The *gettimeofday()* function is marked obsolescent.

19778 **NAME**
 19779 getuid — get a real user ID

19780 **SYNOPSIS**
 19781 #include <unistd.h>
 19782 uid_t getuid(void);

19783 **DESCRIPTION**
 19784 The *getuid()* function shall return the real user ID of the calling process.

19785 **RETURN VALUE**
 19786 The *getuid()* function shall always be successful and no return value is reserved to indicate the
 19787 error.

19788 **ERRORS**
 19789 No errors are defined.

19790 **EXAMPLES**

19791 **Setting the Effective User ID to the Real User ID**

19792 The following example sets the effective user ID and the real user ID of the current process to the
 19793 real user ID of the caller.

```
19794       #include <unistd.h>
19795       #include <sys/types.h>
19796       ...
19797       setreuid(getuid(), getuid());
19798       ...
```

19799 **APPLICATION USAGE**
 19800 None.

19801 **RATIONALE**
 19802 None.

19803 **FUTURE DIRECTIONS**
 19804 None.

19805 **SEE ALSO**
 19806 [getegid\(\)](#), [geteuid\(\)](#), [getgid\(\)](#), [setegid\(\)](#), [seteuid\(\)](#), [setgid\(\)](#), [setregid\(\)](#), [setreuid\(\)](#), [setuid\(\)](#), the Base
 19807 Definitions volume of IEEE Std 1003.1-200x, [<sys/types.h>](#), [<unistd.h>](#)

19808 **CHANGE HISTORY**
 19809 First released in Issue 1. Derived from Issue 1 of the SVID.

19810 **Issue 6**
 19811 In the SYNOPSIS, the optional include of the [<sys/types.h>](#) header is removed.

19812 The following new requirements on POSIX implementations derive from alignment with the
 19813 Single UNIX Specification:

- 19814 • The requirement to include [<sys/types.h>](#) has been removed. Although [<sys/types.h>](#) was
 19815 required for conforming implementations of previous POSIX specifications, it was not
 19816 required for UNIX applications.

19817 **NAME**
19818 getutxent, getutxid, getutxline — get user accounting database entries

19819 **SYNOPSIS**

```
19820 XSI       #include <utmpx.h>  
19821       struct utmpx *getutxent(void);  
19822       struct utmpx *getutxid(const struct utmpx *id);  
19823       struct utmpx *getutxline(const struct utmpx *line);
```

19824 **DESCRIPTION**

19825 Refer to *endutxent()*.

19826 **NAME**

19827 getwc — get a wide character from a stream

19828 **SYNOPSIS**

19829 #include <stdio.h>

19830 #include <wchar.h>

19831 wint_t getwc(FILE *stream);

19832 **DESCRIPTION**

19833 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 19834 conflict between the requirements described here and the ISO C standard is unintentional. This
 19835 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

19836 The *getwc()* function shall be equivalent to *fgetwc()*, except that if it is implemented as a macro it
 19837 may evaluate *stream* more than once, so the argument should never be an expression with side
 19838 effects.

19839 **RETURN VALUE**19840 Refer to *fgetwc()*.19841 **ERRORS**19842 Refer to *fgetwc()*.19843 **EXAMPLES**

19844 None.

19845 **APPLICATION USAGE**

19846 Since it may be implemented as a macro, *getwc()* may treat incorrectly a *stream* argument with
 19847 side effects. In particular, *getwc(*f++)* does not necessarily work as expected. Therefore, use of
 19848 this function is not recommended; *fgetwc()* should be used instead.

19849 **RATIONALE**

19850 None.

19851 **FUTURE DIRECTIONS**

19852 None.

19853 **SEE ALSO**19854 *fgetwc()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>, <wchar.h>19855 **CHANGE HISTORY**

19856 First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working
 19857 draft.

19858 **Issue 5**

19859 The Optional Header (OH) marking is removed from <stdio.h>.

19860 **NAME**
 19861 getwchar — get a wide character from a stdin stream

19862 **SYNOPSIS**
 19863 #include <wchar.h>
 19864 wint_t getwchar(void);

19865 **DESCRIPTION**
 19866 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 19867 conflict between the requirements described here and the ISO C standard is unintentional. This
 19868 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

19869 The *getwchar()* function shall be equivalent to *getwc(stdin)*.

19870 **RETURN VALUE**
 19871 Refer to *fgetwc()*.

19872 **ERRORS**
 19873 Refer to *fgetwc()*.

19874 **EXAMPLES**
 19875 None.

19876 **APPLICATION USAGE**
 19877 If the **wint_t** value returned by *getwchar()* is stored into a variable of type **wchar_t** and then
 19878 compared against the **wint_t** macro WEOF, the result may be incorrect. Only the **wint_t** type is
 19879 guaranteed to be able to represent any wide character and WEOF.

19880 **RATIONALE**
 19881 None.

19882 **FUTURE DIRECTIONS**
 19883 None.

19884 **SEE ALSO**
 19885 *fgetwc()*, *getwc()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**wchar.h**>

19886 **CHANGE HISTORY**
 19887 First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working
 19888 draft.

NAME

glob, globfree — generate pathnames matching a pattern

SYNOPSIS

```
#include <glob.h>

int glob(const char *restrict pattern, int flags,
         int (*errfunc)(const char *epath, int errno),
         glob_t *restrict pglob);
void globfree(glob_t *pglob);
```

DESCRIPTION

The *glob()* function is a pathname generator that shall implement the rules defined in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.13, Pattern Matching Notation, with optional support for rule 3 in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.13.3, Patterns Used for Filename Expansion.

The structure type **glob_t** is defined in **<glob.h>** and includes at least the following members:

Member Type	Member Name	Description
size_t	<i>gl_pathc</i>	Count of paths matched by <i>pattern</i> .
char **	<i>gl_pathv</i>	Pointer to a list of matched pathnames.
size_t	<i>gl_offs</i>	Slots to reserve at the beginning of <i>gl_pathv</i> .

The argument *pattern* is a pointer to a pathname pattern to be expanded. The *glob()* function shall match all accessible pathnames against this pattern and develop a list of all pathnames that match. In order to have access to a pathname, *glob()* requires search permission on every component of a path except the last, and read permission on each directory of any filename component of *pattern* that contains any of the following special characters: '*', '?', and '['.

The *glob()* function shall store the number of matched pathnames into *pglob->gl_pathc* and a pointer to a list of pointers to pathnames into *pglob->gl_pathv*. The pathnames shall be in sort order as defined by the current setting of the *LC_COLLATE* category; see the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3.2, *LC_COLLATE*. The first pointer after the last pathname shall be a null pointer. If the pattern does not match any pathnames, the returned number of matched paths is set to 0, and the contents of *pglob->gl_pathv* are implementation-defined.

It is the caller's responsibility to create the structure pointed to by *pglob*. The *glob()* function shall allocate other space as needed, including the memory pointed to by *gl_pathv*. The *globfree()* function shall free any space associated with *pglob* from a previous call to *glob()*.

The *flags* argument is used to control the behavior of *glob()*. The value of *flags* is a bitwise-inclusive OR of zero or more of the following constants, which ar

19932	GLOB_MARK	Each pathname that is a directory that matches <i>pattern</i> shall have a slash appended.
19933		
19934	GLOB_NOCHECK	Supports rule 3 in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.13.3, Patterns Used for Filename Expansion. If <i>pattern</i> does not match any pathname, then <i>glob()</i> shall return a list consisting of only <i>pattern</i> , and the number of matched pathnames is 1.
19935		
19936		
19937		
19938	GLOB_NOESCAPE	Disable backslash escaping.
19939	GLOB_NOSORT	Ordinarily, <i>glob()</i> sorts the matching pathnames according to the current setting of the <i>LC_COLLATE</i> category; see the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3.2, <i>LC_COLLATE</i> . When this flag is used, the order of pathnames returned is unspecified.
19940		
19941		
19942		

The GLOB_APPEND flag can be used to append a new set of pathnames to those found in a previous call to *glob()*. The following rules apply to applications when two or more calls to *glob()* are made with the same value of *pglob* and without intervening calls to *globfree()*:

1. The first such call shall not set GLOB_APPEND. All subsequent calls shall set it.
2. All the calls shall set GLOB_DOOFFS, or all shall not set it.
3. After the second call, *pglob->gl_pathv* points to a list containing the following:
 - a. Zero or more null pointers, as specified by GLOB_DOOFFS and *pglob->gl_offs*.
 - b. Pointers to the pathnames that were in the *pglob->gl_pathv* list before the call, in the same order as before.
 - c. Pointers to the new pathnames generated by the second call, in the specified order.
4. The count returned in *pglob->gl_pathc* shall be the total number of pathnames from the two calls.
5. The application can change any of the fields after a call to *glob()*. If it does, the application shall reset them to the original value before a subsequent call, using the same *pglob* value, to *globfree()* or *glob()* with the GLOB_APPEND flag.

If, during the search, a directory is encountered that cannot be opened or read and *errfunc* is not a null pointer, *glob()* calls (**errfunc()*) with two arguments:

1. The *epath* argument is a pointer to the path that failed.
2. The *eerrno* argument is the value of *errno* from the failure, as set by *opendir()*, *readdir()*, or *stat()*. (Other values may be used to report other errors not explicitly documented for those functions.)

If (**errfunc()*) is called and returns non-zero, or if the GLOB_ERR flag is set in *flags*, *glob()* shall stop the scan and return GLOB_ABORTED after setting *gl_pathc* and *gl_pathv* in *pglob* to reflect the paths already scanned. If GLOB_ERR is not set and either *errfunc* is a null pointer or (**errfunc()*) returns 0, the error shall be ignored.

The *glob()* function shall not fail because of large files.

RETURN VALUE

Upon successful completion, *glob()* shall return 0. The argument *pglob->gl_pathc* shall return the number of matched pathnames and the argument *pglob->gl_pathv* shall contain a pointer to a null-terminated list of matched and sorted pathnames. However, if *pglob->gl_pathc* is 0, the content of *pglob->gl_pathv* is undefined.

The *globfree()* function shall not return a value.

If *glob()* terminates due to an error, it shall return one of the non-zero constants defined in

19976 <glob.h>. The arguments *pglob->gl_pathc* and *pglob->gl_pathv* are still set as defined above.

19977 ERRORS

19978 The *glob()* function shall fail and return the corresponding value if:

19979 GLOB_ABORTED The scan was stopped because GLOB_ERR was set or (**errfunc()*)
19980 returned non-zero.

19981 GLOB_NOMATCH The pattern does not match any existing pathname, and
19982 GLOB_NOCHECK was not set in flags.

19983 GLOB_NOSPACE An attempt to allocate memory failed.

19984 EXAMPLES

19985 One use of the GLOB_DOOFFS flag is by applications that build an argument list for use with
19986 *execv()*, *execve()*, or *execvp()*. Suppose, for example, that an application wants to do the
19987 equivalent of:

```
19988            ls -l *.c
```

19989 but for some reason:

```
19990            system("ls -l *.c")
```

19991 is not acceptable. The application could obtain approximately the same result using the
19992 sequence:

```
19993            globbuf.gl_offs = 2;  
19994            glob("*.c", GLOB_DOOFFS, NULL, &globbuf);  
19995            globbuf.gl_pathv[0] = "ls";  
19996            globbuf.gl_pathv[1] = "-l";  
19997            execvp("ls", &globbuf.gl_pathv[0]);
```

19998 Using the same example:

```
19999            ls -l *.c *.h
```

20000 could be approximately simulated using GLOB_APPEND as follows:

```
20001            globbuf.gl_offs = 2;  
20002            glob("*.c", GLOB_DOOFFS, NULL, &globbuf);  
20003            glob("*.h", GLOB_DOOFFS|GLOB_APPEND, NULL, &globbuf);  
20004            ...
```

20005 APPLICATION USAGE

20006 This function is not provided for the purpose of enabling utilities to perform pathname
20007 expansion on their arguments, as this operation is performed by the shell, and utilities are
20008 explicitly not expected to redo this. Instead, it is provided for applications that need to do
20009 pathname expansion on strings obtained from other sources, such as a pattern typed by a user or
20010 read from a file.

20011 If a utility needs to see if a pathname matches a given pattern, it can use *fnmatch()*.

20012 Note that *gl_pathc* and *gl_pathv* have meaning even if *glob()* fails. This allows *glob()* to report
20013 partial results in the event of an error. However, if *gl_pathc* is 0, *gl_pathv* is unspecified even if
20014 *glob()* did not return an error.

20015 The GLOB_NOCHECK option could be used when an application wants to expand a pathname
20016 if wildcards are specified, but wants to treat the pattern as just a string otherwise. The *sh* utility
20017 might use this for option-arguments, for example.

20018 The new pathnames generated by a subsequent call with GLOB_APPEND are not sorted
20019 together with the previous pathnames. This mirrors the way that the shell handles pathname
20020 expansion when multiple expansions are done on a command line.

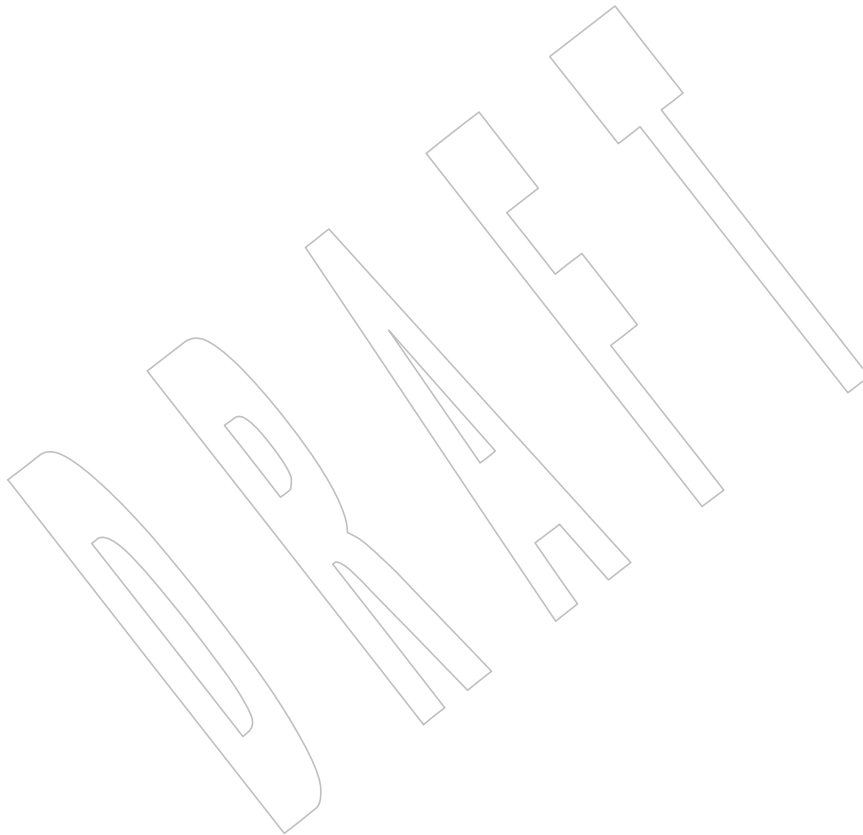
Applications that need tilde and parameter expansion should use *wordexp()*.

RATIONALE

It was claimed that the GLOB_DOOFFS flag is unnecessary because it could be simulated using:

```
new = (char **)malloc((n + pglob->gl_pathc + 1)
    * sizeof(char *));
(void) memcpy(new+n, pglob->gl_pathv,
    pglob->gl_pathc * sizeof(char *));
(void) memset(new, 0, n * sizeof(char *));
free(pglob->gl_pathv);
pglob->gl_pathv = new;
```

However, this assumes that the memory pointed to by *gl_pathv* is a block that was separately created using *malloc()*. This is not necessarily the case. An application should make no assumptions about how the memory referenced by fields in *pglob* was allocated. It might have



20052 **NAME**

20053 gmtime, gmtime_r — convert a time value to a broken-down UTC time

20054 **SYNOPSIS**

20055 #include <time.h>

20056 struct tm *gmtime(const time_t *timer);

20057 CX struct tm *gmtime_r(const time_t *restrict timer,
20058 struct tm *restrict result);20059 **DESCRIPTION**20060 CX For *gmtime()*: The functionality described on this reference page is aligned with the ISO C
20061 standard. Any conflict between the requirements described here and the ISO C standard is
20062 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.20063 The *gmtime()* function shall convert the time in seconds since the Epoch pointed to by *timer* into
20064 a broken-down time, expressed as Coordinated Universal Time (UTC).20065 CX The relationship between a time in seconds since the Epoch used as an argument to *gmtime()*
20066 and the **tm** structure (defined in the <**time.h**> header) is that the result shall be as specified in
20067 the expression given in the definition of seconds since the Epoch (see the Base Definitions
20068 volume of IEEE Std 1003.1-200x, Section 4.14, Seconds Since the Epoch), where the names in the
20069 structure and in the expression correspond.20070 The same relationship shall apply for *gmtime_r()*.20071 The *gmtime()* function need not be thread-safe. A function that is not required to be thread-safe
20072 is not required to be reentrant.20073 The *asctime()*, *ctime()*, *gmtime()*, and *localtime()* functions shall return values in one of two static
20074 objects: a broken-down time structure and an array of type **char**. Execution of any of the
20075 functions may overwrite the information returned in either of these objects by any of the other
20076 functions.20077 The *gmtime_r()* function shall convert the time in seconds since the Epoch pointed to by *timer*
20078 into a broken-down time expressed as Coordinated Universal Time (UTC). The broken-down
20079 time is stored in the structure referred to by *result*. The *gmtime_r()* function shall also return the
20080 address of the same structure.20081 **RETURN VALUE**20082 Upon successful completion, the *gmtime()* function shall return a pointer to a **struct tm**. If an
20083 error is detected, *gmtime()* shall return a null pointer and set *errno* to indicate the error.20084 Upon successful completion, *gmtime_r()* shall return the address of the structure pointed to by
20085 the argument *result*. If an error is detected, *gmtime_r()* shall return a null pointer and set *errno* to
20086 indicate the error.20087 **ERRORS**20088 CX The *gmtime()* and *gmtime_r()* functions shall fail if:20089 CX [E`OVERFLOW`] The result cannot be represented.

gmtime()**EXAMPLES**

None.

APPLICATION USAGE

The *gmtime_r()* function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asctime(), *clock()*, *ctime()*, *difftime()*, *localtime()*, *mktime()*, *strftime()*, *strptime()*, *time()*, *utime()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

A note indicating that the *gmtime()* function need not be reentrant is added to the DESCRIPTION.

The *gmtime_r()* function is included for alignment with the POSIX Threads Extension.

Issue 6

The *gmtime_r()* function is included for alignment with the POSIX Threads Extension. /R12 100 -(i3 B)

20121 **NAME**

20122 grantpt — grant access to the slave pseudo-terminal device

20123 **SYNOPSIS**

```
20124 XSI #include <stdlib.h>
20125 int grantpt(int fildev);
```

20126 **DESCRIPTION**

20127 The *grantpt()* function shall change the mode and ownership of the slave pseudo-terminal
 20128 device associated with its master pseudo-terminal counterpart. The *fildev* argument is a file
 20129 descriptor that refers to a master pseudo-terminal device. The user ID of the slave shall be set to
 20130 the real UID of the calling process and the group ID shall be set to an unspecified group ID. The
 20131 permission mode of the slave pseudo-terminal shall be set to readable and writable by the
 20132 owner, and writable by the group.

20133 The behavior of the *grantpt()* function is unspecified if the application has installed a signal
 20134 handler to catch SIGCHLD signals.

20135 **RETURN VALUE**

20136 Upon successful completion, *grantpt()* shall return 0; otherwise, it shall return -1 and set *errno* to
 20137 indicate the error.

20138 **ERRORS**

20139 The *grantpt()* function may fail if:

- | | | |
|-------|----------|--|
| 20140 | [EBADF] | The <i>fildev</i> argument is not a valid open file descriptor. |
| 20141 | [EINVAL] | The <i>fildev</i> argument is not associated with a master pseudo-terminal device. |
| 20142 | [EACCES] | The corresponding slave pseudo-terminal device could not be accessed. |

20143 **EXAMPLES**

20144 None.

20145 **APPLICATION USAGE**

20146 None.

20147 **RATIONALE**

20148 None.

20149 **FUTURE DIRECTIONS**

20150 None.

20151 **SEE ALSO**

20152 *open()*, *ptsname()*, *unlockpt()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

20153 **CHANGE HISTORY**

20154 First released in Issue 4, Version 2.

20155 **Issue 5**

20156 Moved from X/OPEN UNIX extension to BASE.

20157 The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section.

20158 **NAME**20159 `hcreate, hdestroy, hsearch` — manage hash search table20160 **SYNOPSIS**

```

20161 XSI #include <search.h>
20162
20162 int hcreate(size_t nel);
20163 void hdestroy(void);
20164 ENTRY *hsearch(ENTRY item, ACTION action);

```

20165 **DESCRIPTION**20166 The `hcreate()`, `hdestroy()`, and `hsearch()` functions shall manage hash search tables.

20167 The `hcreate()` function shall allocate sufficient space for the table, and the application shall
 20168 ensure it is called before `hsearch()` is used. The `nel` argument is an estimate of the maximum
 20169 number of entries that the table shall contain. This number may be adjusted upward by the
 20170 algorithm in order to obtain certain mathematically favorable circumstances.

20171 The `hdestroy()` function shall dispose of the search table, and may be followed by another call to
 20172 `hcreate()`. After the call to `hdestroy()`, the data can no longer be considered accessible.

20173 The `hsearch()` function is a hash-table search routine. It shall return a pointer into a hash table
 20174 indicating the location at which an entry can be found. The `item` argument is a structure of type
 20175 **ENTRY** (defined in the `<search.h>` header) containing two pointers: `item.key` points to the
 20176 comparison key (a `char *`), and `item.data` (a `void *`) points to any other data to be associated with
 20177 that key. The comparison function used by `hsearch()` is `strcmp()`. The `action` argument is a
 20178 member of an enumeration type **ACTION** indicating the disposition of the entry if it cannot be
 20179 found in the table. **ENTER** indicates that the item should be inserted in the table at an
 20180 appropriate point. **FIND** indicates that no entry should be made. Unsuccessful resolution is
 20181 indicated by the return of a null pointer.

20182 These functions need not be thread-safe. A function that is not required to be thread-safe is not
 20183 required to be reentrant.

20184 **RETURN VALUE**

20185 The `hcreate()` function shall return 0 if it cannot allocate sufficient space for the table; otherwise,
 20186 it shall return non-zero.

20187 The `hdestroy()` function shall not return a value.

20188 The `hsearch()` function shall return a null pointer if either the action is **FIND** and the item could
 20189 not be found or the action is **ENTER** and the table is full.

20190 **ERRORS**

20191 The `hcreate()` and `hsearch()` functions may fail if:

20192 [ENOMEM] Insufficient storage space is available.

EXAMPLES

The following example reads in strings followed by two numbers and stores them in a hash table, discarding duplicates. It then reads in strings and finds the matching entry in the hash table and prints it out.

```

20193
20194
20195
20196
20197
20198
20199
20200
20201
20202
20203
20204
20205
20206
20207
20208
20209
20210
20211
20212
20213
20214
20215
20216
20217
20218
20219
20220
20221
20222
20223
20224
20225
20226
20227
20228
20229
20230
20231
20232
20233
20234
20235
20236
20237
20238
20239
20240
#include <stdio.h>
#include <search.h>
#include <string.h>

struct info {          /* This is the info stored in the table */
    int age, room;     /* other than the key. */
};

#define NUM_EMPL      5000    /* # of elements in search table. */

int main(void)
{
    char string_space[NUM_EMPL*20]; /* Space to store strings. */
    struct info info_space[NUM_EMPL]; /* Space to store employee info. */
    char *str_ptr = string_space; /* Next space in string_space. */
    struct info *info_ptr = info_space; /* Next space in info_space. */

    ENTRY item;
    ENTRY *found_item; /* Name to look for in table. */
    char name_to_find[30];

    int i = 0;

    /* Create table; no error checking is performed. */
    (void) hcreate(NUM_EMPL);
    while (scanf("%s%d%d", str_ptr, &info_ptr->age,
                &info_ptr->room) != EOF && i++ < NUM_EMPL) {

        /* Put information in structure, and structure in item. */
        item.key = str_ptr;
        item.data = info_ptr;
        str_ptr += strlen(str_ptr) + 1;
        info_ptr++;

        /* Put item into table. */
        (void) hsearch(item, ENTER);
    }

    /* Access table. */
    item.key = name_to_find;
    while (scanf("%s", item.key) != EOF) {
        if ((found_item = hsearch(item, FIND)) != NULL) {

            /* If item is in the table. */
            (void)printf("found %s, age = %d, room = %d\n",
                found_item->key,
                ((struct info *)found_item->data)->age,
                ((struct info *)found_item->data)->room);
        } else
            (void)printf("no such employee %s\n", name_to_find);
    }
    return 0;
}

```

hcreate()

20241 **APPLICATION USAGE**
20242 The *hcreate()* and *hsearch()* functions may use *malloc()* to allocate space.

20243 **RATIONALE**
20244 None.

20245 **FUTURE DIRECTIONS**
20246 None.

20247 **SEE ALSO**
20248 *bsearch()*, *lsearch()*, *malloc()*, *strcmp()*, *tsearch()*, the Base Definitions volume of
20249 IEEE Std 1003.1-200x, <**search.h**>

20250 **CHANGE HISTORY**
20251 First released in Issue 1. Derived from Issue 1 of the SVID.

20252 **Issue 6**
20253 The normative text is updated to avoid use of the term “must” for application requirements.
20254 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

DRAFT

20255 **NAME**

20256 htonl, htons, ntohl, ntohs — convert values between host and network byte order

20257 **SYNOPSIS**

```
20258 #include <arpa/inet.h>
20259
20259 uint32_t htonl(uint32_t hostlong);
20260 uint16_t htons(uint16_t hostshort);
20261 uint32_t ntohl(uint32_t netlong);
20262 uint16_t ntohs(uint16_t netshort);
```

20263 **DESCRIPTION**20264 These functions shall convert 16-bit and 32-bit quantities between network byte order and host
20265 byte order.

20266 On some implementations, these functions are defined as macros.

20267 The **uint32_t** and **uint16_t** types are defined in **<inttypes.h>**.20268 **RETURN VALUE**20269 The *htonl()* and *htons()* functions shall return the argument value converted from host to
20270 network byte order.20271 The *ntohl()* and *ntohs()* functions shall return the argument value converted from network to
20272 host byte order.20273 **ERRORS**

20274 No errors are defined.

20275 **EXAMPLES**

20276 None.

20277 **APPLICATION USAGE**20278 These functions are most often used in conjunction with IPv4 addresses and ports as returned by
20279 *gethostent()* and *getservent()*.20280 **RATIONALE**

20281 None.

20282 **FUTURE DIRECTIONS**

20283 None.

20284 **SEE ALSO**20285 *endhostent()*, *endservent()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<inttypes.h>**,
20286 **<arpa/inet.h>**20287 **CHANGE HISTORY**

20288 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

20289 **NAME**
 20290 hypot, hypotf, hypotl — Euclidean distance function

20291 **SYNOPSIS**
 20292 #include <math.h>
 20293 double hypot(double x, double y);
 20294 float hypotf(float x, float y);
 20295 long double hypotl(long double x, long double y);

20296 **DESCRIPTION**
 20297 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 20298 conflict between the requirements described here and the ISO C standard is unintentional. This
 20299 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

20300 These functions shall compute the value of the square root of x^2+y^2 without undue overflow or
 20301 underflow.

20302 An application wishing to check for error situations should set *errno* to zero and call
 20303 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 20304 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 20305 zero, an error has occurred.

20306 **RETURN VALUE**
 20307 Upon successful completion, these functions shall return the length of the hypotenuse of a right-
 20308 angled triangle with sides of length *x* and *y*.

20309 If the correct value would cause overflow, a range error shall occur and *hypot()*, *hypotf()*, and
 20310 *hypotl()* shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL,
 20311 respectively.

20312 MX If *x* or *y* is $\pm\text{Inf}$, $+\text{Inf}$ shall be returned (even if one of *x* or *y* is NaN).
 20313 If *x* or *y* is NaN, and the other is not $\pm\text{Inf}$, a NaN shall be returned.
 20314 If both arguments are subnormal and the correct result is subnormal, a range error may occur
 20315 and the correct result is returned.

20316 **ERRORS**
 20317 These functions shall fail if:

20318 Range Error The result overflows.
 20319 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 20320 then *errno* shall be set to [ERANGE]. If the integer expression
 20321 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
 20322 floating-point exception shall be raised.

20323 These functions may fail if:

20324 MX Range Error The result underflows.
 20325 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 20326 then *errno* shall be set to [ERANGE]. If the integer expression
 20327 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 20328 floating-point exception shall be raised.

20329
20330
20331
20332
20333
20334
20335
20336
20337
20338
20339
20340
20341
20342
20343
20344
20345
20346
20347
20348
20349
20350
20351
20352
20353
20354
20355
20356
20357
20358
20359
20360
20361

EXAMPLES

See the EXAMPLES section in *atan2()*.

APPLICATION USAGE

hypot(x,y), *hypot(y,x)*, and *hypot(x, -y)* are equivalent.

hypot(x, ±0) is equivalent to *fabs(x)*.

Underflow only happens when both *x* and *y* are subnormal and the (inexact) result is also subnormal.

These functions take precautions against overflow during intermediate steps of the computation.

On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

atan2(), *feclearexcept()*, *fetestexcept()*, *isnan()*, *sqrt()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *hypot()* function is no longer marked as an extension.

The *hypotf()* and *hypotl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/49 is applied, updating the EXAMPLES section.

20362 **NAME**

20363 iconv — codeset conversion function

20364 **SYNOPSIS**

20365 #include <iconv.h>

```
20366 size_t iconv(iconv_t cd, char **restrict inbuf,
20367             size_t *restrict inbytesleft, char **restrict outbuf,
20368             size_t *restrict outbytesleft);
```

20369 **DESCRIPTION**

20370 The *iconv()* function shall convert the sequence of characters from one codeset, in the array
 20371 specified by *inbuf*, into a sequence of corresponding characters in another codeset, in the array
 20372 specified by *outbuf*. The codesets are those specified in the *iconv_open()* call that returned the
 20373 conversion descriptor, *cd*. The *inbuf* argument points to a variable that points to the first
 20374 character in the input buffer and *inbytesleft* indicates the number of bytes to the end of the buffer
 20375 to be converted. The *outbuf* argument points to a variable that points to the first available byte in
 20376 the output buffer and *outbytesleft* indicates the number of the available bytes to the end of the
 20377 buffer.

20378 For state-dependent encodings, the conversion descriptor *cd* is placed into its initial shift state by
 20379 a call for which *inbuf* is a null pointer, or for which *inbuf* points to a null pointer. When *iconv()* is
 20380 called in this way, and if *outbuf* is not a null pointer or a pointer to a null pointer, and *outbytesleft*
 20381 points to a positive value, *iconv()* shall place, into the output buffer, the byte sequence to change
 20382 the output buffer to its initial shift state. If the output buffer is not large enough to hold the
 20383 entire reset sequence, *iconv()* shall fail and set *errno* to [E2BIG]. Subsequent calls with *inbuf* as
 20384 other than a null pointer or a pointer to a null pointer cause the conversion to take place from
 20385 the current state of the conversion descriptor.

20386 If a sequence of input bytes does not form a valid character in the specified codeset, conversion
 20387 shall stop after the previous successfully converted character. If the input buffer ends with an
 20388 incomplete character or shift sequence, conversion shall stop after the previous successfully
 20389 converted bytes. If the output buffer is not large enough to hold the entire converted input,
 20390 conversion shall stop just prior to the input bytes that would cause the output buffer to
 20391 overflow. The variable pointed to by *inbuf* shall be updated to point to the byte following the last
 20392 byte successfully used in the conversion. The value pointed to by *inbytesleft* shall be
 20393 decremented to reflect the number of bytes still not converted in the input buffer. The variable
 20394 pointed to by *outbuf* shall be updated to point to the byte following the last byte of converted
 20395 output data. The value pointed to by *outbytesleft* shall be decremented to reflect the number of
 20396 bytes still available in the output buffer. For state-dependent encodings, the conversion
 20397 descriptor shall be updated to reflect the shift state in effect at the end of the last successfully
 20398 converted byte sequence.

20399 If *iconv()* encounters a character in the input buffer that is valid, but for which an identical
 20400 character does not exist in the target codeset, *iconv()* shall perform an implementation-defined
 20401 conversion on this character.

20402 **RETURN VALUE**

20403 The *iconv()* function shall update the variables pointed to by the arguments to reflect the extent
 20404 of the conversion and return the number of non-identical conversions performed. If the entire
 20405 string in the input buffer is converted, the value pointed to by *inbytesleft* shall be 0. If the input
 20406 conversion is stopped due to any conditions mentioned above, the value pointed to by *inbytesleft*
 20407 shall be non-zero and *errno* shall be set to indicate the condition. If an error occurs, *iconv()* shall
 20408 return (*size_t*)-1 and set *errno* to indicate the error.

20409 **ERRORS**20410 The *iconv()* function shall fail if:20411 [EILSEQ] Input conversion stopped due to an input byte that does not belong to the
20412 input codeset.

20413 [E2BIG] Input conversion stopped due to lack of space in the output buffer.

20414 [EINVAL] Input conversion stopped due to an incomplete character or shift sequence at
20415 the end of the input buffer.20416 The *iconv()* function may fail if:20417 [EBADF] The *cd* argument is not a valid open conversion descriptor.20418 **EXAMPLES**

20419 None.

20420 **APPLICATION USAGE**

20421 The *inbuf* argument indirectly points to the memory area which contains the conversion input
20422 data. The *outbuf* argument indirectly points to the memory area which is to contain the result of
20423 the conversion. The objects indirectly pointed to by *inbuf* and *outbuf* are not restricted to
20424 containing data that is directly representable in the ISO C standard language **char** data type. The
20425 type of *inbuf* and *outbuf*, **char ****, does not imply that the objects pointed to are interpreted as
20426 null-terminated C strings or arrays of characters. Any interpretation of a byte sequence that
20427 represents a character in a given character set encoding scheme is done internally within the
20428 codeset converters. For example, the area pointed to indirectly by *inbuf* and/or *outbuf* can
20429 contain all zero octets that are not interpreted as string terminators but as coded character data
20430 according to the respective codeset encoding scheme. The type of the data (**char**, **short**, **long**, and
20431 so on) read or stored in the objects is not specified, but may be inferred for both the input and
20432 output data by the converters determined by the *fromcode* and *toencode* arguments of *iconv_open()*.

20433 Regardless of the data type inferred by the converter, the size of the remaining space in both
20434 input and output objects (the *inbytesleft* and *outbytesleft* arguments) is always measured in bytes.

20435 For implementations that support the conversion of state-dependent encodings, the conversion
20436 descriptor must be able to accurately reflect the shift-state in effect at the end of the last
20437 successful conversion. It is not required that the conversion descriptor itself be updated, which
20438 would require it to be a pointer type. Thus, implementations are free to implement the
20439 descriptor as a handle (other than a pointer type) by which the conversion information can be
20440 accessed and updated.

20441 **RATIONALE**

20442 None.

20443 **FUTURE DIRECTIONS**

20444 None.

20445 **SEE ALSO**

20446 *iconv_open()*, *iconv_close()*, *mbsrtowcs()*, the Base Definitions volume of IEEE Std 1003.1-200x,
20447 <**iconv.h**>

20448 **CHANGE HISTORY**

20449 First released in Issue 4. Derived from the HP-UX Manual.

20450 **Issue 6**20451 The SYNOPSIS has been corrected to align with the <**iconv.h**> reference page.

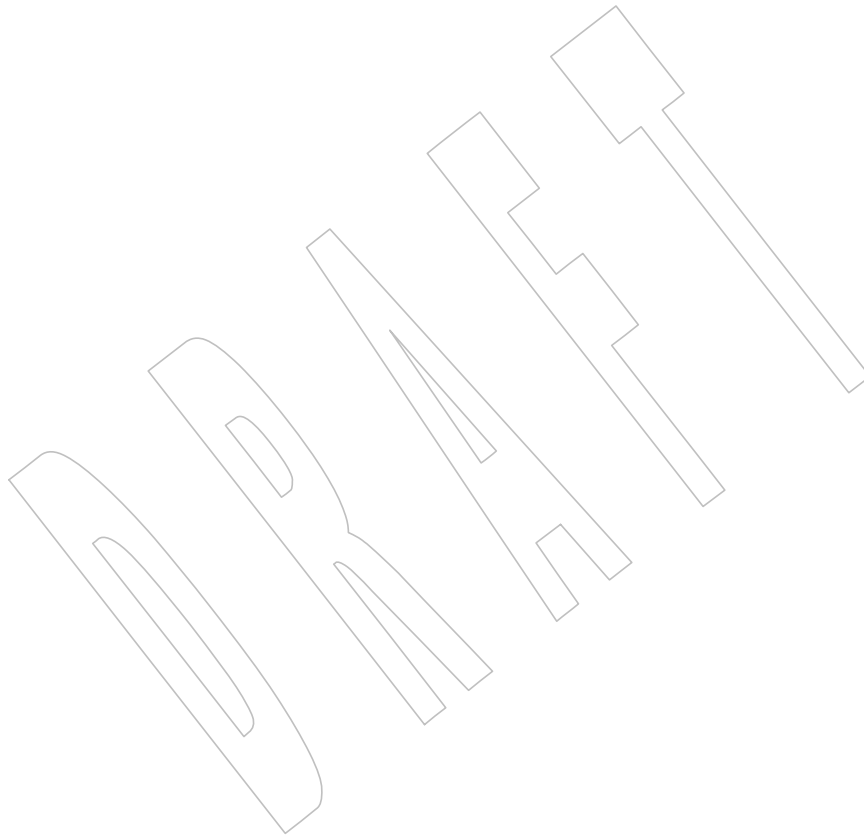
20452 The **restrict** keyword is added to the *iconv()* prototype for alignment with the
20453 ISO/IEC 9899:1999 standard.

20454

Issue 7

20455

The *iconv()* function is moved from the XSI option to the Base.



20456 **NAME**20457 `iconv_close` — codeset conversion deallocation function20458 **SYNOPSIS**20459 `#include <iconv.h>`20460 `int iconv_close(iconv_t cd);`20461 **DESCRIPTION**20462 The `iconv_close()` function shall deallocate the conversion descriptor `cd` and all other associated
20463 resources allocated by `iconv_open()`.20464 If a file descriptor is used to implement the type **iconv_t**, that file descriptor shall be closed.20465 **RETURN VALUE**20466 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and `errno` set to
20467 indicate the error.20468 **ERRORS**20469 The `iconv_close()` function may fail if:

20470 [EBADF] The conversion descriptor is invalid.

20471 **EXAMPLES**

20472 None.

20473 **APPLICATION USAGE**

20474 None.

20475 **RATIONALE**

20476 None.

20477 **FUTURE DIRECTIONS**

20478 None.

20479 **SEE ALSO**20480 `iconv()`, `iconv_open()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<iconv.h>`20481 **CHANGE HISTORY**

20482 First released in Issue 4. Derived from the HP-UX Manual.

20483 **Issue 7**20484 The `iconv_close()` function is moved from the XSI option to the Base.

20485 **NAME**
 20486 `iconv_open` — codeset conversion allocation function

20487 **SYNOPSIS**
 20488 `#include <iconv.h>`

20489 `iconv_t iconv_open(const char *tocode, const char *fromcode);`

20490 **DESCRIPTION**

20491 The `iconv_open()` function shall return a conversion descriptor that describes a conversion from
 20492 the codeset specified by the string pointed to by the `fromcode` argument to the codeset specified
 20493 by the string pointed to by the `tocode` argument. For state-dependent encodings, the conversion
 20494 descriptor shall be in a codeset-dependent initial shift state, ready for immediate use with
 20495 `iconv()`.

20496 Settings of `fromcode` and `tocode` and their permitted combinations are implementation-defined.

20497 A conversion descriptor shall remain valid until it is closed by `iconv_close()` or an implicit close.

20498 If a file descriptor is used to implement conversion descriptors, the `FD_CLOEXEC` flag shall be
 20499 set; see `<fcntl.h>`.

20500 **RETURN VALUE**

20501 Upon successful completion, `iconv_open()` shall return a conversion descriptor for use on
 20502 subsequent calls to `iconv()`. Otherwise, `iconv_open()` shall return `(iconv_t)-1` and set `errno` to
 20503 indicate the error.

20504 **ERRORS**

20505 The `iconv_open()` function may fail if:

20506 [EMFILE] All file descriptors available to the process are currently open.

20507 [ENFILE] Too many files are currently open in the system.

20508 [ENOMEM] Insufficient storage space is available.

20509 [EINVAL] The conversion specified by `fromcode` and `tocode` is not supported by the
 20510 implementation.

20511 **EXAMPLES**

20512 None.

20513 **APPLICATION USAGE**

20514 Some implementations of `iconv_open()` use `malloc()` to allocate space for internal buffer areas.
 20515 The `iconv_open()` function may fail if there is insufficient storage space to accommodate these
 20516 buffers.

20517 Conforming applications must assume that conversion descriptors are not valid after a call to
 20518 one of the `exec` functions.

20519 Application developers should consult the system documentation to determine the supported
 20520 codesets and their naming schemes.

20521 **RATIONALE**

20522 None.

20523 **FUTURE DIRECTIONS**

20524 None.

20525
20526
20527
20528
20529
20530
20531**SEE ALSO**

iconv(), *iconv_close()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<fcntl.h>`, `<iconv.h>`

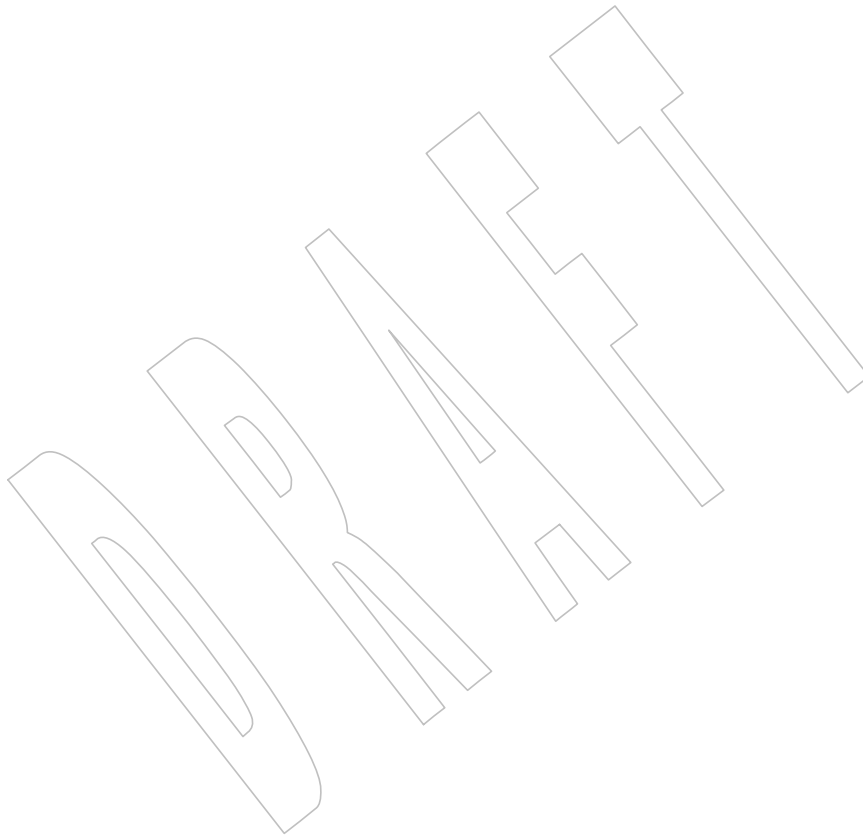
CHANGE HISTORY

First released in Issue 4. Derived from the HP-UX Manual.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

The *iconv_open()* function is moved from the XSI option to the Base.



20532 NAME

20533 `if_freenameindex` — free memory allocated by `if_nameindex`

20534 SYNOPSIS

20535 `#include <net/if.h>`

20536 `void if_freenameindex(struct if_nameindex *ptr);`

20537 DESCRIPTION

20538 The `if_freenameindex()` function shall free the memory allocated by `if_nameindex()`. The `ptr`
 20539 argument shall be a pointer that was returned by `if_nameindex()`. After `if_freenameindex()` has
 20540 been called, the application shall not use the array of which `ptr` is the address.

20541 RETURN VALUE

20542 None.

20543 ERRORS

20544 No errors are defined.

20545 EXAMPLES

20546 None.

20547 APPLICATION USAGE

20548 None.

20549 RATIONALE

20550 None.

20551 FUTURE DIRECTIONS

20552 None.

20553 SEE ALSO

20554 [*getsockopt\(\)*](#), [*if_indextoname\(\)*](#), [*if_nameindex\(\)*](#), [*if_nametoindex\(\)*](#), [*setsockopt\(\)*](#), the Base Definitions
 20555 volume of IEEE Std 1003.1-200x, [**<net/if.h>**](#)

20556 CHANGE HISTORY

20557 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

20558 **NAME**20559 `if_indextoname` — map a network interface index to its corresponding name20560 **SYNOPSIS**20561 `#include <net/if.h>`20562 `char *if_indextoname(unsigned ifindex, char *ifname);`20563 **DESCRIPTION**20564 The `if_indextoname()` function shall map an interface index to its corresponding name.20565 When this function is called, *ifname* shall point to a buffer of at least {IF_NAMESIZE} bytes. The
20566 function shall place in this buffer the name of the interface with index *ifindex*.20567 **RETURN VALUE**20568 If *ifindex* is an interface index, then the function shall return the value supplied in *ifname*, which
20569 points to a buffer now containing the interface name. Otherwise, the function shall return a
20570 NULL pointer and set *errno* to indicate the error.20571 **ERRORS**20572 The `if_indextoname()` function shall fail if:

20573 [ENXIO] The interface does not exist.

20574 **EXAMPLES**

20575 None.

20576 **APPLICATION USAGE**

20577 None.

20578 **RATIONALE**

20579 None.

20580 **FUTURE DIRECTIONS**

20581 None.

20582 **SEE ALSO**20583 [*getsockopt\(\)*](#), [*if_freenameindex\(\)*](#), [*if_nameindex\(\)*](#), [*if_nametoindex\(\)*](#), [*setsockopt\(\)*](#), the Base Definitions
20584 volume of IEEE Std 1003.1-200x, [**<net/if.h>**](#)20585 **CHANGE HISTORY**

20586 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

20587 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/28 is applied, changing {IFNAMSIZ} to
20588 {IF_NAMESIZ} in the DESCRIPTION.

20589 **NAME**20590 `if_nameindex` — return all network interface names and indexes20591 **SYNOPSIS**20592 `#include <net/if.h>`20593 `struct if_nameindex *if_nameindex(void);`20594 **DESCRIPTION**20595 The `if_nameindex()` function shall return an array of `if_nameindex` structures, one structure per
20596 interface. The end of the array is indicated by a structure with an `if_index` field of zero and an
20597 `if_name` field of NULL.20598 Applications should call `if_freenameindex()` to release the memory that may be dynamically
20599 allocated by this function, after they have finished using it.20600 **RETURN VALUE**20601 An array of structures identifying local interfaces. A NULL pointer is returned upon an error,
20602 with `errno` set to indicate the error.20603 **ERRORS**20604 The `if_nameindex()` function may fail if:

20605 [ENOBUFS] Insufficient resources are available to complete the function.

20606 **EXAMPLES**

20607 None.

20608 **APPLICATION USAGE**

20609 None.

20610 **RATIONALE**

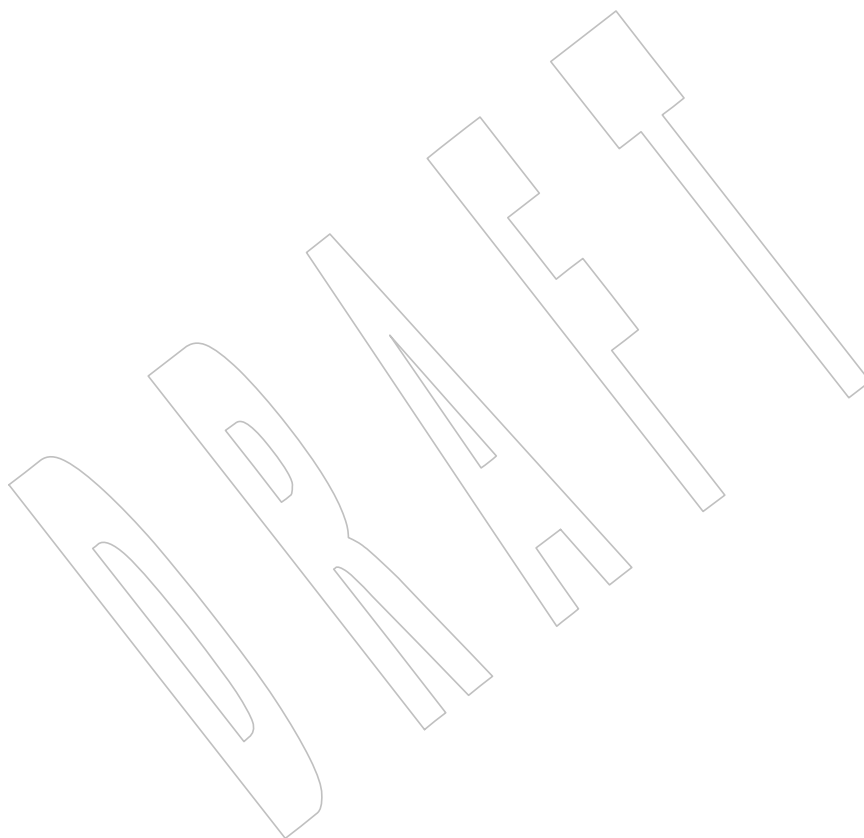
20611 None.

20612 **FUTURE DIRECTIONS**

20613 None.

20614 **SEE ALSO**20615 `getsockopt()`, `if_freenameindex()`, `if_indextoname()`, `if_nametoindex()`, `setsockopt()`, the Base
20616 Definitions volume of IEEE Std 1003.1-200x, `<net/if.h>`20617 **CHANGE HISTORY**

20618 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.



NAME

ilogb, ilogbf, ilogbl — return an unbiased exponent

SYNOPSIS

```
#include <math.h>

int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall return the exponent part of their argument x . Formally, the return value is the integral part of $\log_r |x|$ as a signed integral value, for non-zero x , where r is the radix of the machine's floating-point arithmetic, which is the value of `FLT_RADIX` defined in `<float.h>`.

An application wishing to check for error situations should set `errno` to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `errno` is non-zero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the exponent part of x as a signed integer value. They are equivalent to calling the corresponding `logb()` function and casting the returned value to type `int`.

XSI If x is 0, the value `FP_ILOGB0` shall be returned. On XSI-conformant systems, a domain error shall occur;

CX otherwise, a domain error may occur.

XSI If x is $\pm\text{Inf}$, the value `{INT_MAX}` shall be returned. On XSI-conformant systems, a domain error shall occur;

CX otherwise, a domain error may occur.

XSI If x is a NaN, the value `FP_ILOGBNAN` shall be returned.

20687 shall be raised.

20688 These functions may fail if:

20689 Domain Error The x argument is zero, NaN, or $\pm\text{Inf}$.

20690 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 20691 then $errno$ shall be set to [EDOM]. If the integer expression (math_errhandling
 20692 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 20693 shall be raised.

20694 EXAMPLES

20695 None.

20696 APPLICATION USAGE

20697 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 20698 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

20699 RATIONALE

20700 The errors come from taking the expected floating-point value and converting it to **int**, which is
 20701 an invalid operation in IEEE Std 754-1985 (since overflow, infinity, and NaN are not
 20702 representable in a type **int**), so should be a domain error.

20703 There are no known implementations that overflow. For overflow to happen, {INT_MAX} must
 20704 be less than $\text{LDBL_MAX_EXP} \cdot \log_2(\text{FLT_RADIX})$ or {INT_MIN} must be greater than
 20705 $\text{LDBL_MIN_EXP} \cdot \log_2(\text{FLT_RADIX})$ if subnormals are not supported, or {INT_MIN} must be
 20706 greater than $(\text{LDBL_MIN_EXP} - \text{LDBL_MANT_DIG}) \cdot \log_2(\text{FLT_RADIX})$ if subnormals are
 20707 supported.

20708 FUTURE DIRECTIONS

20709 None.

20710 SEE ALSO

20711 [fclearexcept\(\)](#), [fetestexcept\(\)](#), [logb\(\)](#), [scalbln\(\)](#), the Base Definitions volume of
 20712 IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
 20713 [<float.h>](#), [<math.h>](#)

20714 CHANGE HISTORY

20715 First released in Issue 4, Version 2.

20716 Issue 5

20717 Moved from X/OPEN UNIX extension to BASE.

20718 Issue 6

20719 The $\text{ilogb}()$ function is no longer marked as an extension.

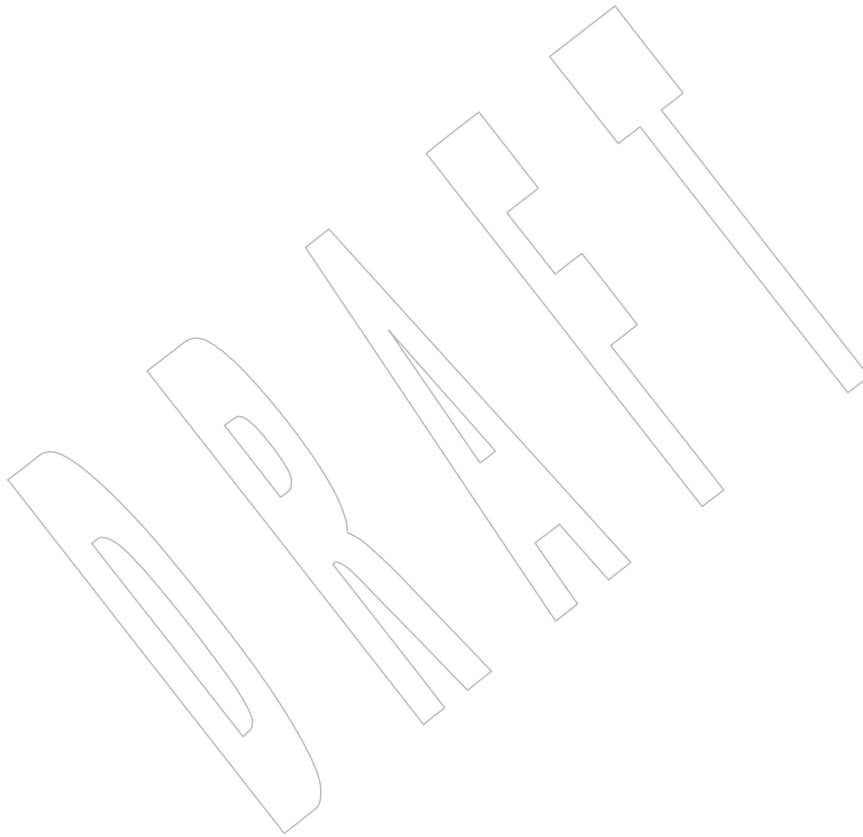
20720 The $\text{ilogbf}()$ and $\text{ilogbl}()$ functions are added for alignment with the ISO/IEC 9899:1999
 20721 standard.

20722 The RETURN VALUE section is revised for alignment with the ISO/IEC 9899:1999 standard.

20723 Functionality relating to the XSI option is marked.

20724 Issue 7

20725 ISO/IEC 9899:1999 standard, Technical Corrigendum 2 #48 (SD5-XSH-ERN-71), #49, and #79
 20726 (SD5-XSH-ERN-72) are applied.



20754 **NAME**
 20755 `imaxdiv` — return quotient and remainder

20756 **SYNOPSIS**
 20757 `#include <inttypes.h>`
 20758 `imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);`

20759 **DESCRIPTION**
 20760 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 20761 conflict between the requirements described here and the ISO C standard is unintentional. This
 20762 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

20763 The *imaxdiv()* function shall compute *numer* / *denom* and *numer* % *denom* in a single operation.

20764 **RETURN VALUE**
 20765 The *imaxdiv()* function shall return a structure of type **imaxdiv_t**, comprising both the quotient
 20766 and the remainder. The structure shall contain (in either order) the members *quot* (the quotient)
 20767 and *rem* (the remainder), each of which has type **intmax_t**.

20768 If either part of the result cannot be represented, the behavior is undefined.

20769 **ERRORS**
 20770 No errors are defined.

20771 **EXAMPLES**
 20772 None.

20773 **APPLICATION USAGE**
 20774 None.

20775 **RATIONALE**
 20776 None.

20777 **FUTURE DIRECTIONS**
 20778 None.

20779 **SEE ALSO**
 20780 *imaxabs()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<inttypes.h>`

20781 **CHANGE HISTORY**
 20782 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

inet_addr()**NAME**

inet_addr, inet_ntoa — IPv4 address manipulation

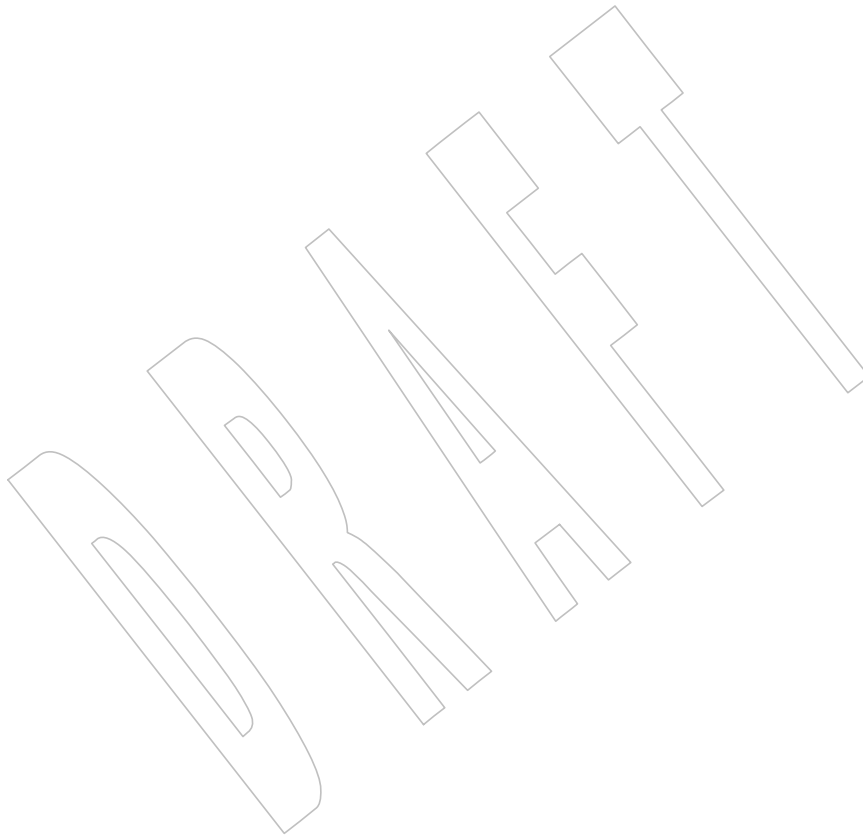
SYNOPSIS

```
#include <arpa/inet.h>

in_addr_t inet_addr(const char *cp);
char *inet_ntoa(struct in_addr in);
```

DESCRIPTION

The *inet_addr()* function shall convert the string pointed to by *cp*, in t



20820

EXAMPLES

20821

None.

20822

APPLICATION USAGE

20823

The return value of *inet_ntoa()* may point to static data that may be overwritten by subsequent calls to *inet_ntoa()*.

20824

20825

RATIONALE

20826

None.

20827

FUTURE DIRECTIONS

20828

None.

20829

SEE ALSO

20830

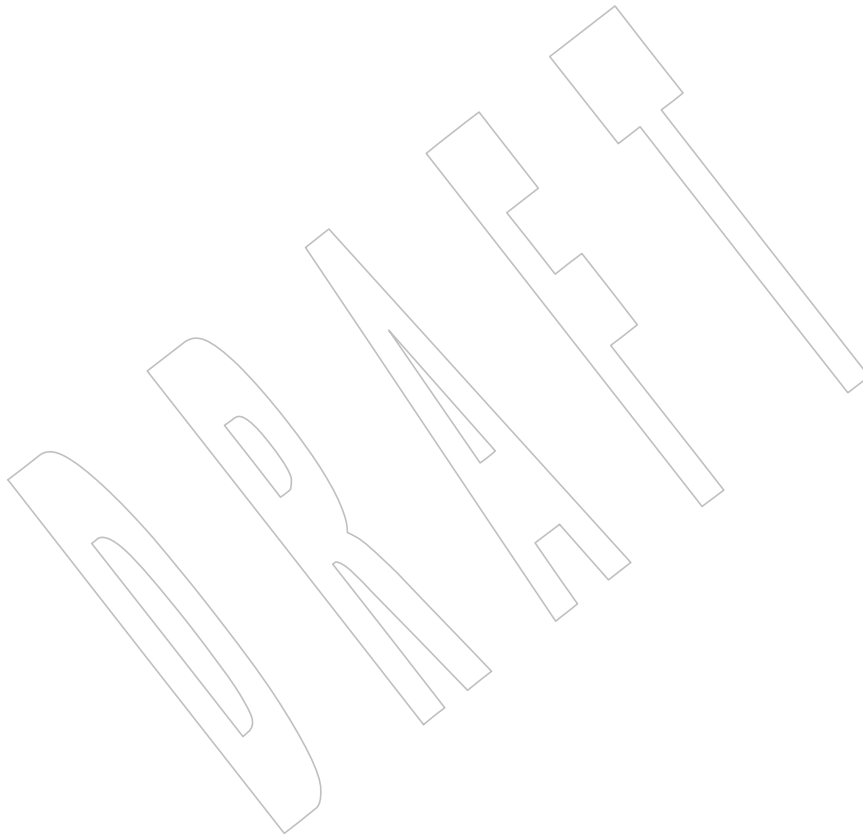
endhostent(), *endnetent()*, the Base Definitions volume of IEEE Std 1003.1-200x, <arpa/inet.h>

20831

CHANGE HISTORY

20832

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.



20833 **NAME**

20834 inet_ntop, inet_pton — convert IPv4 and IPv6 addresses between binary and text form

20835 **SYNOPSIS**

20836 #include <arpa/inet.h>

20837 const char *inet_ntop(int af, const void *restrict src,
20838 char *restrict dst, socklen_t size);

20839 int inet_pton(int af, const char *restrict src, void *restrict dst);

20840 **DESCRIPTION**

20841 The *inet_ntop()* function shall convert a numeric address into a text string suitable for
 20842 IP6 presentation. The *af* argument shall specify the family of the address. This can be AF_INET or
 20843 AF_INET6. The *src* argument points to a buffer holding an IPv4 address if the *af* argument is
 20844 IP6 AF_INET, or an IPv6 address if the *af* argument is AF_INET6; the address must be in network
 20845 byte order. The *dst* argument points to a buffer where the function stores the resulting text string;
 20846 it shall not be NULL. The *size* argument specifies the size of this buffer, which shall be large
 20847 IP6 enough to hold the text string (INET_ADDRSTRLEN characters for IPv4,
 20848 INET6_ADDRSTRLEN characters for IPv6).

20849 The *inet_pton()* function shall convert an address in its standard text presentation form into its
 20850 IP6 numeric binary form. The *af* argument shall specify the family of the address. The AF_INET and
 20851 AF_INET6 address families shall be supported. The *src* argument points to the string being
 20852 passed in. The *dst* argument points to a buffer into which the function stores the numeric
 20853 IP6 address; this shall be large enough to hold the numeric address (32 bits for AF_INET, 128 bits
 20854 for AF_INET6).

20855 If the *af* argument of *inet_pton()* is AF_INET, the *src* string shall be in the standard IPv4 dotted-
 20856 decimal form:

20857 ddd.ddd.ddd.ddd

20858 where "ddd" is a one to three digit decimal number between 0 and 255 (see *inet_addr()*). The
 20859 *inet_pton()* function does not accept other formats (such as the octal numbers, hexadecimal
 20860 numbers, and fewer than four numbers that *inet_addr()* accepts).

20861 IP6 If the *af* argument of *inet_pton()* is AF_INET6, the *src* string shall be in one of the following
 20862 standard IPv6 text forms:

20863 1. The preferred form is "x:x:x:x:x:x:x:x", where the 'x's are the hexadecimal values
 20864 of the eight 16-bit pieces of the address. Leading zeros in individual fields can be
 20865 omitted, but there shall be at least one numeral in every field.

20866 2. A string of contiguous zero fields in the preferred form can be shown as ":::". The ":::"
 20867 can only appear once in an address. Unspecified addresses ("0:0:0:0:0:0:0:0") may
 20868 be represented simply as "::".

20869 3. A third form that is sometimes more convenient when dealing with a mixed environment
 20870 of IPv4 and IPv6 nodes is "x:x:x:x:x:x.d.d.d.d", where the 'x's are the
 20871 hexadecimal values of the six high-order 16-bit pieces of the address, and the 'd's are the
 20872 decimal values of the four low-order 8-bit pieces of the address (standard IPv4
 20873 representation).

20874 **Note:** A more extensive description of the standard representations of IPv6 addresses can be found in
 20875 RFC 2373.

20876
20877
20878

20879
20880
20881
20882

20883
20884

20885
20886

20887

20888
20889

20890
20891

20892
20893

20894
20895

20896
20897

20898
20899

20900

20901
20902

20903
20904
20905

RETURN VALUE

The *inet_ntop()* function shall return a pointer to the buffer containing the text string if the conversion succeeds, and NULL otherwise, and set *errno* to indicate the error.

The *inet_pton()* function shall return 1 if the conversion succeeds, with the address pointed to by *dst* in network byte order. It shall return 0 if the input is not a valid IPv4 dotted-decimal string or a valid IPv6 address string, or -1 with *errno* set to [EAFNOSUPPORT] if the *af* argument is unknown.

ERRORS

The *inet_ntop()* and *inet_pton()* functions shall fail if:

[EAFNOSUPPORT]

The *af* argument is invalid.

[ENOSPC]

The size of the *inet_ntop()* result buffer is inadequate.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<arpa/inet.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

IPv6 extensions are marked.

The **restrict** keyword is added to the *inet_ntop()* and *inet_pton()* prototypes for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/29 is applied, adding “the address must be in network byte order” to the end of the fourth sentence of the first paragraph in the DESCRIPTION.

initstate()

DRAFT

20947 **ERRORS**

20948 No errors are defined.

20949 **EXAMPLES**

20950 None.

20951 **APPLICATION USAGE**

20952 After initialization, a state array can be restarted at a different point in one of two ways:

- 20953 1. The *initstate()* function can be used, with the desired seed, state array, and size of the
- 20954 array.
- 20955 2. The *setstate()* function, with the desired state, can be used, followed by *srandom()* with
- 20956 the desired seed. The advantage of using both of these functions is that the size of the
- 20957 state array does not have to be saved once it is initialized.

20958 Although some implementations of *random()* have written messages to standard error, such

20959 implementations do not conform to IEEE Std 1003.1-200x.

20960 Issue 5 restored the historical behavior of this function.

20961 Threaded applications should use *erand48()*, *nrand48()*, or *jrand48()* instead of *random()* when

20962 an independent random number sequence in multiple threads is required.

20963 **RATIONALE**

20964 None.

20965 **FUTURE DIRECTIONS**

20966 None.

20967 **SEE ALSO**20968 *drand48()*, *rand()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdlib.h>`20969 **CHANGE HISTORY**

20970 First released in Issue 4, Version 2.

20971 **Issue 5**

20972 Moved from X/OPEN UNIX extension to BASE.

20973 In the DESCRIPTION, the phrase “values smaller than 8” is replaced with “values greater than

20974 or equal to 8, or less than 32”, “*size*<8” is replaced with “ $8 \leq \textit{size} < 32$ ”, and a new first paragraph

20975 is added to the RETURN VALUE section. A note is added to the APPLICATION USAGE

20976 indicating that these changes restore the historical behavior of the function.

20977 **Issue 6**

20978 In the DESCRIPTION, duplicate text “For values greater than or equal to 8 . . .” is removed.

20979 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/30 is applied, removing *rand_r()* from the

20980 list of suggested functions in the APPLICATION USAGE section.

20981 **NAME**
 20982 `insque, remque` — insert or remove an element in a queue

20983 **SYNOPSIS**

```
20984 XSI #include <search.h>
20985 void insque(void *element, void *pred);
20986 void remque(void *element);
```

20987 **DESCRIPTION**

20988 The `insque()` and `remque()` functions shall manipulate queues built from doubly-linked lists. The
 20989 queue can be either circular or linear. An application using `insque()` or `remque()` shall ensure it
 20990 defines a structure in which the first two members of the structure are pointers to the same type
 20991 of structure, and any further members are application-specific. The first member of the structure
 20992 is a forward pointer to the next entry in the queue. The second member is a backward pointer to
 20993 the previous entry in the queue. If the queue is linear, the queue is terminated with null
 20994 pointers. The names of the structure and of the pointer members are not subject to any special
 20995 restriction.

20996 The `insque()` function shall insert the element pointed to by `element` into a queue immediately
 20997 after the element pointed to by `pred`.

20998 The `remque()` function shall remove the element pointed to by `element` from a queue.

20999 If the queue is to be used as a linear list, invoking `insque(&element, NULL)`, where `element` is the
 21000 initial element of the queue, shall initialize the forward and backward pointers of `element` to null
 21001 pointers.

21002 If the queue is to be used as a circular list, the application shall ensure it initializes the forward
 21003 pointer and the backward pointer of the initial element of the queue to the element's own
 21004 address.

21005 **RETURN VALUE**

21006 The `insque()` and `remque()` functions do not return a value.

21007 **ERRORS**

21008 No errors are defined.

21009 **EXAMPLES**

21010 **Creating a Linear Linked List**

21011 The following example creates a linear linked list.

```
21012 #include <search.h>
21013 ...
21014 struct myque element1;
21015 struct myque element2;
21016
21017 char *data1 = "DATA1";
21018 char *data2 = "DATA2";
21019 ...
21020 element1.data = data1;
21021 element2.data = data2;
21022
21023 insque (&element1, NULL);
21024 insque (&element2, &element1);
```

21023

Creating a Circular Linked List

21024

The following example creates a circular linked list.

21025

```
#include <search.h>
```

21026

```
...
```

21027

```
struct myque element1;
```

21028

```
struct myque element2;
```

21029

```
char *data1 = "DATA1";
```

21030

```
char *data2 = "DATA2";
```

21031

```
...
```

21032

```
element1.data = data1;
```

21033

```
element2.data = data2;
```

21034

```
element1.fwd = &element1;
```

21035

```
element1.bck = &element1;
```

21036

```
insque (&element2, &element1);
```

21037

Removing an Element

21038

The following example removes the element pointed to by *element1*.

21039

```
#include <search.h>
```

21040

```
...
```

21041

```
struct myque element1;
```

21042

```
...
```

21043

```
remque (&element1);
```

21044

APPLICATION USAGE

21045

The historical implementations of these functions described the arguments as being of type **struct qelem *** rather than as being of type **void *** as defined here. In those implementations, **struct qelem** was commonly defined in **<search.h>** as:

21048

```
struct qelem {
```

21049

```
    struct qelem *q_forw;
```

21050

```
    struct qelem *q_back;
```

21051

```
};
```

21052

Applications using these functions, however, were never able to use this structure directly since it provided no room for the actual data contained in the elements. Most applications defined structures that contained the two pointers as the initial elements and also provided space for, or pointers to, the object's data. Applications that used these functions to update more than one type of table also had the problem of specifying two or more different structures with the same name, if they literally used **struct qelem** as specified.

21058

As described here, the implementations were actually expecting a structure type where the first two members were forward and backward pointers to structures. With C compilers that didn't provide function prototypes, applications used structures as specified in the DESCRIPTION above and the compiler did what the application expected.

21062

If this method had been carried forward with an ISO C standard compiler and the historical function prototype, most applications would have to be modified to cast pointers to the structures actually used to be pointers to **struct qelem** to avoid compilation warnings. By specifying **void *** as the argument type, applications do not need to change (unless they specifically referenced **struct qelem** and depended on it being defined in **<search.h>**).

21063

21064

21065

21066

insque()

21067 **RATIONALE**
21068 None.

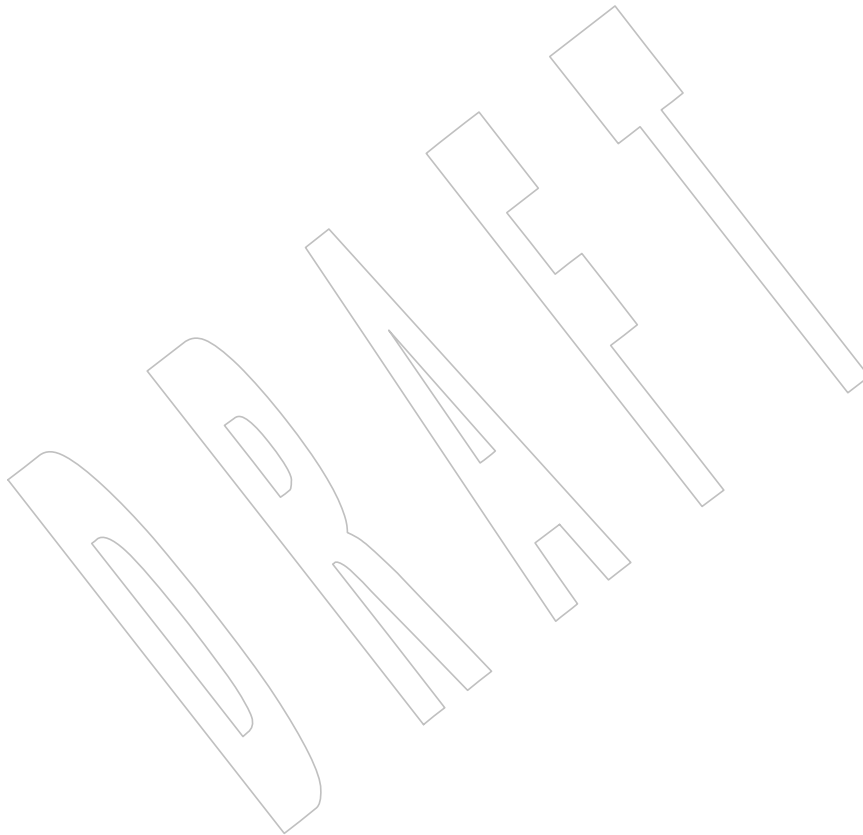
21069 **FUTURE DIRECTIONS**
21070 None.

21071 **SEE ALSO**
21072 The Base Definitions volume of IEEE Std 1003.1-200x, <[search.h](#)>

21073 **CHANGE HISTORY**
21074 First released in Issue 4, Version 2.

21075 **Issue 5**
21076 Moved from X/OPEN UNIX extension to BASE.

21077 **Issue 6**
21078 The normative text is updated to avoid use of the term “must” for application requirements.



NAME

ioctl — control a STREAMS device (**STREAMS**)

SYNOPSIS

```
OB XSR #include <stropts.h>
int ioctl(int fildev, int request, ... /* arg */);
```

DESCRIPTION

The *ioctl()* function shall perform a variety of control functions on STREAMS devices. For non-STREAMS devices, the functions performed by this call are unspecified. The *request* argument and an optional third argument (with varying type) shall be passed to and interpreted by the appropriate part of the STREAM associated with *fildev*.

The *fildev* argument is an open file descriptor that refers to a device.

The *request* ar

21117	I_FLUSH	Flushes read and/or write queues, depending on the value of <i>arg</i> . Valid <i>arg</i> values are:
21118		
21119	FLUSHR	Flush all read queues.
21120	FLUSHW	Flush all write queues.
21121	FLUSHRW	Flush all read and all write queues.
21122		The <i>ioctl()</i> function with the I_FLUSH command shall fail if:
21123	[EINVAL]	Invalid <i>arg</i> value.
21124	[EAGAIN] or [ENOSR]	Unable to allocate buffers for flush message.
21125		
21126	[ENXIO]	Hangup received on <i>fildev</i> .
21127	I_FLUSHBAND	Flushes a particular band of messages. The <i>arg</i> argument points to a bandinfo structure. The <i>bi_flag</i> member may be one of FLUSHR, FLUSHW, or FLUSHRW as described above. The <i>bi_pri</i> member determines the priority band to be flushed.
21128		
21129		
21130		
21131	I_SETSIG	Requests that the STREAMS implementation send the SIGPOLL signal to the calling process when a particular event has occurred on the STREAM associated with <i>fildev</i> . I_SETSIG supports an asynchronous processing capability in STREAMS. The value of <i>arg</i> is a bitmask that specifies the events for which the process should be signaled. It is the bitwise-inclusive OR of any combination of the following constants:
21132		
21133		
21134		
21135		
21136		
21137	S_RDNORM	A normal (priority band set to 0) message has arrived at the head of a STREAM head read queue. A signal shall be generated even if the message is of zero length.
21138		
21139		
21140	S_RDBAND	A message with a non-zero priority band has arrived at the head of a STREAM head read queue. A signal shall be generated even if the message is of zero length.
21141		
21142		
21143	S_INPUT	A message, other than a high-priority message, has arrived at the head of a STREAM head read queue. A signal shall be generated even if the message is of zero length.
21144		
21145		
21146	S_HIPRI	A high-priority message is present on a STREAM head read queue. A signal shall be generated even if the message is of zero length.
21147		
21148		
21149	S_OUTPUT	The write queue for normal data (priority band 0) just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) normal data downstream.
21150		
21151		
21152		
21153	S_WRNORM	Equivalent to S_OUTPUT.
21154	S_WRBAND	The write queue for a non-zero priority band just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) priority data downstream.
21155		
21156		
21157		
21158	S_MSG	A STREAMS signal message that contains the SIGPOLL signal has reached the front of the STREAM head read queue.
21159		
21160		

21161		S_ERROR	Notification of an error condition has reached the STREAM head.
21162			
21163		S_HANGUP	Notification of a hangup has reached the STREAM head.
21164		S_BANDURG	When used in conjunction with S_RDBAND, SIGURG is generated instead of SIGPOLL when a priority message reaches the front of the STREAM head read queue.
21165			
21166			
21167			If <i>arg</i> is 0, the calling process shall be unregistered and shall not receive further SIGPOLL signals for the stream associated with <i>fdes</i> .
21168			
21169			Processes that wish to receive SIGPOLL signals shall ensure that they explicitly register to receive them using I_SETSIG. If several processes register to receive this signal for the same event on the same STREAM, each process shall be signaled when the event occurs.
21170			
21171			
21172			
21173			The <i>ioctl()</i> function with the I_SETSIG command shall fail if:
21174		[EINVAL]	The value of <i>arg</i> is invalid.
21175		[EINVAL]	The value of <i>arg</i> is 0 and the calling process is not registered to receive the SIGPOLL signal.
21176			
21177		[EAGAIN]	There were insufficient resources to store the signal request.
21178	I_GETSIG		Returns the events for which the calling process is currently registered to be sent a SIGPOLL signal. The events are returned as a bitmask in an int pointed to by <i>arg</i> , where the events are those specified in the description of I_SETSIG above.
21179			
21180			
21181			
21182			The <i>ioctl()</i> function with the I_GETSIG command shall fail if:
21183		[EINVAL]	Process is not registered to receive the SIGPOLL signal.
21184	I_FIND		Compares the names of all modules currently present in the STREAM to the name pointed to by <i>arg</i> , and returns 1 if the named module is present in the STREAM, or returns 0 if the named module is not present.
21185			
21186			
21187			The <i>ioctl()</i> function with the I_FIND command shall fail if:
21188		[EINVAL]	<i>arg</i> does not contain a valid module name.
21189	I_PEEK		Retrieves the information in the first message on the STREAM head read queue without taking the message off the queue. It is analogous to <i>getmsg()</i> except that this command does not remove the message from the queue. The <i>arg</i> argument points to a strpeek structure.
21190			
21191			
21192			
21193			The application shall ensure that the <i>maxlen</i> member in the ctlbuf and databuf structures is set to the number of bytes of control information and/or data information, respectively, to retrieve. The <i>flags</i> member may be marked RS_HIPRI or 0, as described by <i>getmsg()</i> . If the process sets <i>flags</i> to RS_HIPRI, for example, I_PEEK shall only look for a high-priority message on the STREAM head read queue.
21194			
21195			
21196			
21197			
21198			
21199			I_PEEK returns 1 if a message was retrieved, and returns 0 if no message was found on the STREAM head read queue, or if the RS_HIPRI flag was set in <i>flags</i> and a high-priority message was not present on the STREAM head read queue. It does not wait for a message to arrive. On return, ctlbuf specifies information in the control buffer, databuf specifies information in the data buffer, and <i>flags</i> contains the value RS_HIPRI or 0.
21200			
21201			
21202			
21203			
21204			

21205	L_SRDOPT	Sets the read mode using the value of the argument <i>arg</i> . Read modes are described in <i>read()</i> . Valid <i>arg</i> flags are:
21206		
21207		RNORM Byte-stream mode, the default.
21208		RMSGD Message-discard mode.
21209		RMSGN Message-nondiscard mode.
21210		The bitwise-inclusive OR of RMSGD and RMSGN shall return [EINVAL]. The bitwise-inclusive OR of RNORM and either RMSGD or RMSGN shall result in the other flag overriding RNORM which is the default.
21211		
21212		
21213		In addition, treatment of control messages by the STREAM head may be changed by setting any of the following flags in <i>arg</i> :
21214		
21215		RPROTNORM Fail <i>read()</i> with [EBADMSG] if a message containing a control part is at the front of the STREAM head read queue.
21216		
21217	RPROTDAT Deliver the control part of a message as data when a process issues a <i>read()</i> .	
21218		
21219	RPROTDIS Discard the control part of a message, delivering any data portion, when a process issues a <i>read()</i> .	
21220		
21221	The <i>ioctl()</i> function with the L_SRDOPT command shall fail if:	
21222	[EINVAL] The <i>arg</i> argument is not valid.	
21223	L_GRDOPT	Returns the current read mode setting, as described above, in an int pointed to by the argument <i>arg</i> . Read modes are described in <i>read()</i> .
21224		
21225	L_NREAD	Counts the number of data bytes in the data part of the first message on the STREAM head read queue and places this value in the int pointed to by <i>arg</i> . The return value for the command shall be the number of messages on the STREAM head read queue. For example, if 0 is returned in <i>arg</i> , but the <i>ioctl()</i> return value is greater than 0, this indicates that a zero-length message is next on the queue.
21226		
21227		
21228		
21229		
21230		
21231	L_FDINSERT	Creates a message from specified buffer(s), adds information about another STREAM, and sends the message downstream. The message contains a control part and an optional data part. The data and control parts to be sent are distinguished by placement in separate buffers, as described below. The <i>arg</i> argument points to a strfdinsert structure.
21232		
21233		
21234		
21235		
21236		The application shall ensure that the <i>len</i> member in the ctlbuf strbuf structure is set to the size of a t_uscalar_t plus the number of bytes of control information to be sent with the message. The <i>fildev</i> member specifies the file descriptor of the other STREAM, and the <i>offset</i> member, which must be suitably aligned for use as a t_uscalar_t , specifies the offset from the start of the control buffer where L_FDINSERT shall store a t_uscalar_t whose interpretation is specific to the STREAM end. The application shall ensure that the <i>len</i> member in the databuf strbuf structure is set to the number of bytes of data information to be sent with the message, or to 0 if no data part is to be sent.
21237		
21238		
21239		
21240		
21241		
21242		
21243		
21244		
21245		
21246		The <i>flags</i> member specifies the type of message to be created. A normal message is created if <i>flags</i> is set to 0, and a high-priority message is created if <i>flags</i> is set to RS_HIPRI. For non-priority messages, L_FDINSERT shall block if the STREAM write queue is full due to internal flow control conditions. For priority messages, L_FDINSERT does not block on this condition. For non-priority messages, L_FDINSERT does not block when the write queue is full
21247		
21248		
21249		
21250		
21251		

21252 and O_NONBLOCK is set. Instead, it fails and sets *errno* to [EAGAIN].

21253 I_FDINSERT also blocks, unless prevented by lack of internal resources,
21254 waiting for the availability of message blocks in the STREAM, regardless of
21255 priority or whether O_NONBLOCK has been specified. No partial message is
21256 sent.

21257 The *ioctl()* function with the I_FDINSERT command shall fail if:

21258 [EAGAIN] A non-priority message is specified, the O_NONBLOCK
21259 flag is set, and the STREAM write queue is full due to
21260 internal flow control conditions.

21261 [EAGAIN] or [ENOSR]
21262 Buffers cannot be allocated for the message that is to be
21263 created.

21264 [EINVAL] One of the following:

- 21265 — The *fildev* member of the **strfdinsert** structure is not a
21266 valid, open STREAM file descriptor.
- 21267 — The size of a **t_uscalar_t** plus *offset* is greater than the
21268 *len* member for the buffer specified through **ctlbuf**.
- 21269 — The *offset* member does not specify a properly-aligned
21270 location in the data buffer.
- 21271 — An undefined value is stored in *flags*.

21272 [ENXIO] Hangup received on the STREAM identified by either the
21273 *fildev* argument or the *fildev* member of the **strfdinsert**
21274 structure.

21275 [ERANGE] The *len* member for the buffer specified through **databuf**
21276 does not fall within the range specified by the maximum
21277 and minimum packet sizes of the topmost STREAM
21278 module; or the *len* member for the buffer specified through
21279 **databuf** is larger than the maximum configured size of the
21280 data part of a message; or the *len* member for the buffer
21281 specified through **ctlbuf** is larger than the maximum
21282 configured size of the control part of a message.

21283 I_STR Constructs an internal STREAMS *ioctl()* message from the data pointed to by
21284 *arg*, and sends that message downstream.

21285 This mechanism is provided to send *ioctl()* requests to downstream modules
21286 and drivers. It allows information to be sent with *ioctl()*, and returns to the
21287 process any information sent upstream by the downstream recipient. I_STR
21288 shall block until the system responds with either a positive or negative
21289 acknowledgement message, or until the request times out after some period of
21290 time. If the request times out, it shall fail with *errno* set to [ETIME].

21291 At most, one I_STR can be active on a STREAM. Further I_STR calls shall
21292 block until the active I_STR completes at the STREAM head. The default
21293 timeout interval for these requests is 15 seconds. The O_NONBLOCK flag has
21294 no effect on this call.

21295 To send requests downstream, the application shall ensure that *arg* points to a
21296 **strioc** structure.

21297 The *ic_cmd* member is the internal *ioctl()* command intended for a

21298		downstream module or driver and <i>ic_timeout</i> is the number of seconds
21299		(-1=infinite, 0=use implementation-defined timeout interval, >0=as specified)
21300		an I_STR request shall wait for acknowledgement before timing out. <i>ic_len</i> is
21301		the number of bytes in the data argument, and <i>ic_dp</i> is a pointer to the data
21302		argument. The <i>ic_len</i> member has two uses: on input, it contains the length of
21303		the data argument passed in, and on return from the command, it contains the
21304		number of bytes being returned to the process (the buffer pointed to by <i>ic_dp</i>
21305		should be large enough to contain the maximum amount of data that any
21306		module or the driver in the STREAM can return).
21307		The STREAM head shall convert the information pointed to by the strioc
21308		structure to an internal <i>ioctl()</i> command message and send it downstream.
21309		The <i>ioctl()</i> function with the I_STR command shall fail if:
21310		[EAGAIN] or [ENOSR]
21311		Unable to allocate buffers for the <i>ioctl()</i> message.
21312		[EINVAL] The <i>ic_len</i> member is less than 0 or larger than the
21313		maximum configured size of the data part of a message, or
21314		<i>ic_timeout</i> is less than -1.
21315		[ENXIO] Hangup received on <i>fildev</i> .
21316		[ETIME] A downstream <i>ioctl()</i> timed out before acknowledgement
21317		was received.
21318		An I_STR can also fail while waiting for an acknowledgement if a message
21319		indicating an error or a hangup is received at the STREAM head. In addition,
21320		an error code can be returned in the positive or negative acknowledgement
21321		message, in the event the <i>ioctl()</i> command sent downstream fails. For these
21322		cases, I_STR shall fail with <i>errno</i> set to the value in the message.
21323	I_SWROPT	Sets the write mode using the value of the argument <i>arg</i> . Valid bit settings for
21324		<i>arg</i> are:
21325		SNDZERO Send a zero-length message downstream when a <i>write()</i> of 0
21326		bytes occurs. To not send a zero-length message when a
21327		<i>write()</i> of 0 bytes occurs, the application shall ensure that
21328		this bit is not set in <i>arg</i> (for example, <i>arg</i> would be set to 0).
21329		The <i>ioctl()</i> function with the I_SWROPT command shall fail if:
21330		[EINVAL] <i>arg</i> is not the above value.
21331	I_GWROPT	Returns the current write mode setting, as described above, in the int that is
21332		pointed to by the argument <i>arg</i> .
21333	I_SENDFD	Creates a new reference to the open file description associated with the file
21334		descriptor <i>arg</i> , and writes a message on the STREAMS-based pipe <i>fildev</i>
21335		containing this reference, together with the user ID and group ID of the calling
21336		process.
21337		The <i>ioctl()</i> function with the I_SENDFD command shall fail if:
21338		[EAGAIN] The sending STREAM is unable to allocate a message block
21339		to contain the file pointer; or the read queue of the receiving
21340		STREAM head is full and cannot accept the message sent by
21341		I_SENDFD.

21342		[EBADF]	The <i>arg</i> argument is not a valid, open file descriptor.
21343		[EINVAL]	The <i>fildes</i> argument is not connected to a STREAM pipe.
21344		[ENXIO]	Hangup received on <i>fildes</i> .
21345	I_RECVFD		Retrieves the reference to an open file description from a message written to a STREAMS-based pipe using the I_SENDFD command, and allocates a new file descriptor in the calling process that refers to this open file description. The <i>arg</i> argument is a pointer to a strrecvfd data structure as defined in <stropts.h> .
21346			
21347			
21348			
21349			
21350			The <i>fd</i> member is a file descriptor. The <i>uid</i> and <i>gid</i> members are the effective user ID and effective group ID, respectively, of the sending process.
21351			
21352			If O_NONBLOCK is not set, I_RECVFD shall block until a message is present at the STREAM head. If O_NONBLOCK is set, I_RECVFD shall fail with <i>errno</i> set to [EAGAIN] if no message is present at the STREAM head.
21353			
21354			
21355			If the message at the STREAM head is a message sent by an I_SENDFD, a new file descriptor shall be allocated for the open file descriptor referenced in the message. The new file descriptor is placed in the <i>fd</i> member of the strrecvfd structure pointed to by <i>arg</i> .
21356			
21357			
21358			
21359			The <i>ioctl()</i> function with the I_RECVFD command shall fail if:
21360		[EAGAIN]	A message is not present at the STREAM head read queue and the O_NONBLOCK flag is set.
21361			
21362		[EBADMSG]	The message at the STREAM head read queue is not a message containing a passed file descriptor.
21363			
21364		[EMFILE]	All file descriptors available to the process are currently open.
21365			
21366		[ENXIO]	Hangup received on <i>fildes</i> .
21367	I_LIST		Allows the process to list all the module names on the STREAM, up to and including the topmost driver name. If <i>arg</i> is a null pointer, the return value shall be the number of modules, including the driver, that are on the STREAM pointed to by <i>fildes</i> . This lets the process allocate enough space for the module names. Otherwise, it should point to a str_list structure.
21368			
21369			
21370			
21371			
21372			The <i>sl_nmods</i> member indicates the number of entries the process has allocated in the array. Upon return, the <i>sl_modlist</i> member of the str_list structure shall contain the list of module names, and the number of entries that have been filled into the <i>sl_modlist</i> array is found in the <i>sl_nmods</i> member (the number includes the number of modules including the driver). The return value from <i>ioctl()</i> shall be 0. The entries are filled in starting at the top of the STREAM and continuing downstream until either the end of the STREAM is reached, or the number of requested modules (<i>sl_nmods</i>) is satisfied.
21373			
21374			
21375			
21376			
21377			
21378			
21379			
21380			The <i>ioctl()</i> function with the I_LIST command shall fail if:
21381		[EINVAL]	The <i>sl_nmods</i> member is less than 1.
21382		[EAGAIN] or [ENOSR]	
21383			Unable to allocate buffers.
21384	I_ATMARK		Allows the process to see if the message at the head of the STREAM head read queue is marked by some module downstream. The <i>arg</i> argument determines how the checking is done when there may be multiple marked messages on the STREAM head read queue. It may take on the following values:
21385			
21386			
21387			

21388		ANYMARK	Check if the message is marked.
21389		LASTMARK	Check if the message is the last one marked on the queue.
21390			The bitwise-inclusive OR of the flags ANYMARK and LASTMARK is permitted.
21391			
21392			The return value shall be 1 if the mark condition is satisfied; otherwise, the value shall be 0.
21393			
21394			The <i>ioctl()</i> function with the I_ATMARK command shall fail if:
21395		[EINVAL]	Invalid <i>arg</i> value.
21396	I_CKBAND		Checks if the message of a given priority band exists on the STREAM head read queue. This shall return 1 if a message of the given priority exists, 0 if no such message exists, or -1 on error. <i>arg</i> should be of type int .
21397			
21398			
21399			The <i>ioctl()</i> function with the I_CKBAND command shall fail if:
21400		[EINVAL]	Invalid <i>arg</i> value.
21401	I_GETBAND		Returns the priority band of the first message on the STREAM head read queue in the integer referenced by <i>arg</i> .
21402			
21403			The <i>ioctl()</i> function with the I_GETBAND command shall fail if:
21404		[ENODATA]	No message on the STREAM head read queue.
21405	I_CANPUT		Checks if a certain band is writable. <i>arg</i> is set to the priority band in question. The return value shall be 0 if the band is flow-controlled, 1 if the band is writable, or -1 on error.
21406			
21407			
21408			The <i>ioctl()</i> function with the I_CANPUT command shall fail if:
21409		[EINVAL]	Invalid <i>arg</i> value.
21410	I_SETCLTIME		This request allows the process to set the time the STREAM head shall delay when a STREAM is closing and there is data on the write queues. Before closing each module or driver, if there is data on its write queue, the STREAM head shall delay for the specified amount of time to allow the data to drain. If, after the delay, data is still present, it shall be flushed. The <i>arg</i> argument is a pointer to an integer specifying the number of milliseconds to delay, rounded up to the nearest valid value. If I_SETCLTIME is not performed on a STREAM, an implementation-defined default timeout interval is used.
21411			
21412			
21413			
21414			
21415			
21416			
21417			
21418			The <i>ioctl()</i> function with the I_SETCLTIME command shall fail if:
21419		[EINVAL]	Invalid <i>arg</i> value.
21420	I_GETCLTIME		Returns the close time delay in the integer pointed to by <i>arg</i> .

21421 **Multiplexed STREAMS Configurations**

21422 The following commands are used for connecting and disconnecting multiplexed STREAMS configurations. These commands use an implementation-defined default timeout interval.

21424	I_LINK		Connects two STREAMS, where <i>fdes</i> is the file descriptor of the STREAM connected to the multiplexing driver, and <i>arg</i> is the file descriptor of the STREAM connected to another driver. The STREAM designated by <i>arg</i> is connected below the multiplexing driver. I_LINK requires the multiplexing driver to send an acknowledgement message to the STREAM head regarding the connection. This call shall return a multiplexer ID number (an identifier used to disconnect the multiplexer; see I_UNLINK) on success, and -1 on
21425			
21426			
21427			
21428			
21429			
21430			

21431		failure.
21432		The <i>ioctl()</i> function with the I_LINK command shall fail if:
21433	[ENXIO]	Hangup received on <i>fildev</i> .
21434	[ETIME]	Timeout before acknowledgement message was received at STREAM head.
21435		
21436	[EAGAIN] or [ENOSR]	Unable to allocate STREAMS storage to perform the I_LINK.
21437		
21438		
21439	[EBADF]	The <i>arg</i> argument is not a valid, open file descriptor.
21440	[EINVAL]	The <i>fildev</i> argument does not support multiplexing; or <i>arg</i> is not a STREAM or is already connected downstream from a multiplexer; or the specified I_LINK operation would connect the STREAM head in more than one place in the multiplexed STREAM.
21441		
21442		
21443		
21444		
21445		An I_LINK can also fail while waiting for the multiplexing driver to acknowledge the request, if a message indicating an error or a hangup is received at the STREAM head of <i>fildev</i> . In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_LINK fails with <i>errno</i> set to the value in the message.
21446		
21447		
21448		
21449		
21450	I_UNLINK	Disconnects the two STREAMs specified by <i>fildev</i> and <i>arg</i> . <i>fildev</i> is the file descriptor of the STREAM connected to the multiplexing driver. The <i>arg</i> argument is the multiplexer ID number that was returned by the I_LINK <i>ioctl()</i> command when a STREAM was connected downstream from the multiplexing driver. If <i>arg</i> is MUXID_ALL, then all STREAMs that were connected to <i>fildev</i> shall be disconnected. As in I_LINK, this command requires acknowledgement.
21451		
21452		
21453		
21454		
21455		
21456		
21457		The <i>ioctl()</i> function with the I_UNLINK command shall fail if:
21458	[ENXIO]	Hangup received on <i>fildev</i> .
21459	[ETIME]	Timeout before acknowledgement message was received at STREAM head.
21460		
21461	[EAGAIN] or [ENOSR]	Unable to allocate buffers for the acknowledgement message.
21462		
21463		
21464	[EINVAL]	Invalid multiplexer ID number.
21465		An I_UNLINK can also fail while waiting for the multiplexing driver to acknowledge the request if a message indicating an error or a hangup is received at the STREAM head of <i>fildev</i> . In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_UNLINK shall fail with <i>errno</i> set to the value in the message.
21466		
21467		
21468		
21469		
21470	I_PLINK	Creates a <i>persistent connection</i> between two STREAMs, where <i>fildev</i> is the file descriptor of the STREAM connected to the multiplexing driver, and <i>arg</i> is the file descriptor of the STREAM connected to another driver. This call shall create a persistent connection which can exist even if the file descriptor <i>fildev</i> associated with the upper STREAM to the multiplexing driver is closed. The STREAM designated by <i>arg</i> gets connected via a persistent connection below the multiplexing driver. I_PLINK requires the multiplexing driver to send an acknowledgement message to the STREAM head. This call shall return a
21471		
21472		
21473		
21474		
21475		
21476		
21477		

21478 multiplexer ID number (an identifier that may be used to disconnect the
21479 multiplexer; see I_PUNLINK) on success, and -1 on failure.

21480 The *ioctl()* function with the I_PLINK command shall fail if:

- 21481 [ENXIO] Hangup received on *fildev*.
- 21482 [ETIME] Timeout before acknowledgement message was received at
21483 STREAM head.
- 21484 [EAGAIN] or [ENOSR]
21485 Unable to allocate STREAMS storage to perform the
21486 I_PLINK.
- 21487 [EBADF] The *arg* argument is not a valid, open file descriptor.
- 21488 [EINVAL] The *fildev* argument does not support multiplexing; or *arg* is
21489 not a STREAM or is already connected downstream from a
21490 multiplexer; or the specified I_PLINK operation would
21491 connect the STREAM head in more than one place in the
21492 multiplexed STREAM.

21493 An I_PLINK can also fail while waiting for the multiplexing driver to
21494 acknowledge the request, if a message indicating an error or a hangup is
21495 received at the STREAM head of *fildev*. In addition, an error code can be
21496 returned in the positive or negative acknowledgement message. For these
21497 cases, I_PLINK shall fail with *errno* set to the value in the message.

21498 I_PUNLINK Disconnects the two STREAMs specified by *fildev* and *arg* from a persistent
21499 connection. The *fildev* argument is the file descriptor of the STREAM
21500 connected to the multiplexing driver. The *arg* argument is the multiplexer ID
21501 number that was returned by the I_PLINK *ioctl()* command when a STREAM
21502 was connected downstream from the multiplexing driver. If *arg* is
21503 MUXID_ALL, then all STREAMs which are persistent connections to *fildev*
21504 shall be disconnected. As in I_PLINK, this command requires the multiplexing
21505 driver to acknowledge the request.

21506 The *ioctl()* function with the I_PUNLINK command shall fail if:

- 21507 [ENXIO] Hangup received on *fildev*.
- 21508 [ETIME] Timeout before acknowledgement message was received at
21509 STREAM head.
- 21510 [EAGAIN] or [ENOSR]
21511 Unable to allocate buffers for the acknowledgement
21512 message.
- 21513 [EINVAL] Invalid multiplexer ID number.

21514 An I_PUNLINK can also fail while waiting for the multiplexing driver to
21515 acknowledge the request if a message indicating an error or a hangup is
21516 received at the STREAM head of *fildev*. In addition, an error code can be
21517 returned in the positive or negative acknowledgement message. For these
21518 cases, I_PUNLINK shall fail with *errno* set to the value in the message.

21519 RETURN VALUE

21520 Upon successful completion, *ioctl()* shall return a value other than -1 that depends upon the
21521 STREAMS device control function. Otherwise, it shall return -1 and set *errno* to indicate the
21522 error.

21523 ERRORS

21524 Under the following general conditions, *ioctl()* shall fail if:

- 21525 [EBADF] The *fildev* argument is not a valid open file descriptor.
- 21526 [EINTR] A signal was caught during the *ioctl()* operation.
- 21527 [EINVAL] The STREAM or multiplexer referenced by *fildev* is linked (directly or
21528 indirectly) downstream from a multiplexer.

21529 If an underlying device driver detects an error, then *ioctl()* shall fail if:

- 21530 [EINVAL] The *request* or *arg* argument is not valid for this device.
- 21531 [EIO] Some physical I/O error has occurred.
- 21532 [ENOTTY] The file associated with the *fildev* argument is not a STREAMS device that
21533 accepts control functions.
- 21534 [ENXIO] The *request* and *arg* arguments are valid for this device driver, but the service
21535 requested cannot be performed on this particular sub-device.
- 21536 [ENODEV] The *fildev* argument refers to a valid STREAMS device, but the corresponding
21537 device driver does not support the *ioctl()* function.

21538 If a STREAM is connected downstream from a multiplexer, any *ioctl()* command except
21539 I_UNLINK and I_PUNLINK shall set *errno* to [EINVAL].

21540 EXAMPLES

21541 None.

21542 APPLICATION USAGE

21543 The implementation-defined timeout interval for STREAMS has historically been 15 seconds.

21544 RATIONALE

21545 None.

21546 FUTURE DIRECTIONS

21547 The *ioctl()* function may be removed in a future version.

21548 SEE ALSO

21549 Section 2.6 (on page 38), *close()*, *fcntl()*, *getmsg()*, *open()*, *pipe()*, *poll()*, *putmsg()*, *read()*,
21550 *sigaction()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stropts.h>

21551 CHANGE HISTORY

21552 First released in Issue 4, Version 2.

21553 Issue 5

21554 Moved from X/OPEN UNIX extension to BASE.

21555 Issue 6

21556 The Open Group Corrigendum U028/4 is applied, correcting text in the I_FDINSERT [EINVAL]
21557 case to refer to *ctlbuf*.

21558 This function is marked as part of the XSI STREAMS Option Group.

21559 The normative text is updated to avoid use of the term “must” for application requirements.

21560 Issue 7

21561 SD5-XSH-ERN-100 is applied, correcting the definition of the [ENOTTY] error condition.

21562 The *ioctl()* function is marked obsolescent.

21563 **NAME**
 21564 `isalnum, isalnum_l` — test for an alphanumeric character

SYNOPSIS

```
21565 #include <ctype.h>
21566
21567 int isalnum(int c);
21568 CX int isalnum_l(int c, locale_t locale);
```

DESCRIPTION

21569 CX For `isalnum()`: The functionality described on this reference page is aligned with the ISO C
 21570 standard. Any conflict between the requirements described here and the ISO C standard is
 21571 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.
 21572

21573 CX The `isalnum()` and `isalnum_l()` functions shall test whether `c` is a character of class **alpha** or
 21574 **digit** in the current locale of the process, or in the locale represented by `locale`, respectively; see
 21575 the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

21576 The `c` argument is an **int**, the value of which the application shall ensure is representable as an
 21577 **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the
 21578 behavior is undefined.

RETURN VALUE

21579 CX The `isalnum()` and `isalnum_l()` functions shall return non-zero if `c` is an alphanumeric character;
 21580 otherwise, they shall return 0.
 21581

ERRORS

21582 The `isalnum_l()` function may fail if:

21583
 21584 CX **[EINVAL]** `locale` is not a valid locale object handle.

EXAMPLES

21585 None.
 21586

APPLICATION USAGE

21587 To ensure applications portability, especially across natural languages, only these functions and
 21588 the functions in the reference pages listed in the SEE ALSO section should be used for character
 21589 classification.
 21590

RATIONALE

21591 None.
 21592

FUTURE DIRECTIONS

21593 None.
 21594

SEE ALSO

21595 `isalpha()`, `isblank()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`,
 21596 `isxdigit()`, `setlocale()`, `uselocale()`, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7,
 21597 Locale, `<ctype.h>`, `<stdio.h>`
 21598

CHANGE HISTORY

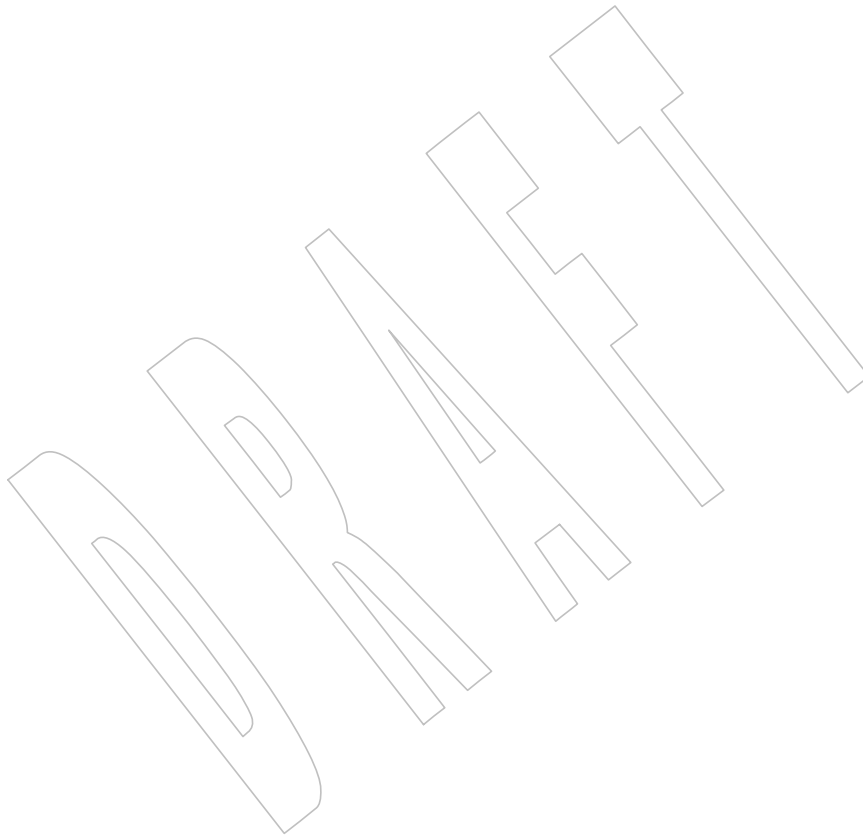
21599 First released in Issue 1. Derived from Issue 1 of the SVID.
 21600

Issue 6

21601 The normative text is updated to avoid use of the term “must” for application requirements.
 21602

21603
21604
21605**Issue 7**

The *isalnum_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



21606 **NAME**
 21607 `isalpha, isalpha_l` — test for an alphabetic character

SYNOPSIS

```
21609 #include <ctype.h>
21610 int isalpha(int c);
21611 CX int isalpha_l(int c, locale_t locale);
```

DESCRIPTION

21612 CX For `isalpha()`: The functionality described on this reference page is aligned with the ISO C
 21613 standard. Any conflict between the requirements described here and the ISO C standard is
 21614 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.
 21615

21616 CX The `isalpha()` and `isalpha_l()` functions shall test whether `c` is a character of class **alpha** in the
 21617 current locale of the process, or in the locale represented by `locale`, respectively; see the Base
 21618 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

21619 The `c` argument is an **int**, the value of which the application shall ensure is representable as an
 21620 **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the
 21621 behavior is undefined.

RETURN VALUE

21622 CX The `isalpha()` and `isalpha_l()` functions shall return non-zero if `c` is an alphabetic character;
 21624 otherwise, they shall return 0.

ERRORS

21625 The `isalpha_l()` function may fail if:

21626 CX **[EINVAL]** `locale` is not a valid locale object handle.

EXAMPLES

21628 None.
 21629

APPLICATION USAGE

21630 To ensure applications portability, especially across natural languages, only these functions and
 21631 the functions in the reference pages listed in the SEE ALSO section should be used for character
 21632 classification.
 21633

RATIONALE

21634 None.
 21635

FUTURE DIRECTIONS

21636 None.
 21637

SEE ALSO

21638 `isalnum()`, `isblank()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`,
 21639 `isxdigit()`, `setlocale()`, `uselocale()`, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7,
 21640 Locale, `<ctype.h>`, `<locale.h>`, `<stdio.h>`
 21641

CHANGE HISTORY

21642 First released in Issue 1. Derived from Issue 1 of the SVID.
 21643

Issue 6

21644 The normative text is updated to avoid use of the term “must” for application requirements.
 21645

21646
21647
21648**Issue 7**

The *isalpha_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



21649 **NAME**21650 `isascii` — test for a 7-bit US-ASCII character21651 **SYNOPSIS**

```
21652 OB XSI #include <ctype.h>
21653 int isascii(int c);
```

21654 **DESCRIPTION**21655 The `isascii()` function shall test whether `c` is a 7-bit US-ASCII character code.21656 The `isascii()` function is defined on all integer values.21657 **RETURN VALUE**

21658 The `isascii()` function shall return non-zero if `c` is a 7-bit US-ASCII character code between 0 and
 21659 octal 0177 inclusive; otherwise, it shall return 0.

21660 **ERRORS**

21661 No errors are defined.

21662 **EXAMPLES**

21663 None.

21664 **APPLICATION USAGE**21665 The `isascii()` function cannot be used portably in a localized application.21666 **RATIONALE**

21667 None.

21668 **FUTURE DIRECTIONS**21669 The `isascii()` function may be removed in a future version.21670 **SEE ALSO**21671 The Base Definitions volume of IEEE Std 1003.1-200x, `<ctype.h>`21672 **CHANGE HISTORY**

21673 First released in Issue 1. Derived from Issue 1 of the SVID.

21674 **Issue 7**21675 The `isascii()` function is marked obsolescent.

21676 **NAME**
 21677 `isastream` — test a file descriptor (**STREAMS**)

21678 **SYNOPSIS**

```
21679 OB XSR #include <stropts.h>
21680 int isastream(int fildev);
```

21681 **DESCRIPTION**

21682 The `isastream()` function shall test whether *fildev*, an open file descriptor, is associated with a
 21683 STREAMS-based file.

21684 **RETURN VALUE**

21685 Upon successful completion, `isastream()` shall return 1 if *fildev* refers to a STREAMS-based file
 21686 and 0 if not. Otherwise, `isastream()` shall return `-1` and set `errno` to indicate the error.

21687 **ERRORS**

21688 The `isastream()` function shall fail if:

21689 [EBADF] The *fildev* argument is not a valid open file descriptor.

21690 **EXAMPLES**

21691 None.

21692 **APPLICATION USAGE**

21693 None.

21694 **RATIONALE**

21695 None.

21696 **FUTURE DIRECTIONS**

21697 The `isastream()` function may be removed in a future version.

21698 **SEE ALSO**

21699 The Base Definitions volume of IEEE Std 1003.1-200x, `<stropts.h>`

21700 **CHANGE HISTORY**

21701 First released in Issue 4, Version 2.

21702 **Issue 5**

21703 Moved from X/OPEN UNIX extension to BASE.

21704 **Issue 7**

21705 The `isastream()` function is marked obsolescent.

21706 **NAME**

21707 isatty — test for a terminal device

21708 **SYNOPSIS**

21709 #include <unistd.h>

21710 int isatty(int *fd*);

21711 **DESCRIPTION**

21712 The *isatty()* function shall test whether *fd*, an open file descriptor, is associated with a

21713 terminal device.

21714 **RETURN VALUE**

21715 The *isatty()* function shall return 1 if *fd* is associated with a terminal; otherwise, it shall return

21716 0 and may set *errno* to indicate the error.

21717 **ERRORS**

21718 The *isatty()* function may fail if:

21719 [EBADF] The *fd* argument is not a valid open file descriptor.

21720 [ENOTTY] The file associated with the *fd* argument is not a terminal.

21721 **EXAMPLES**

21722 None.

21723 **APPLICATION USAGE**

21724 The *isatty()* function does not necessarily indicate that a human being is available for interaction

21725 via *fd*. It is quite possible that non-terminal devices are connected to the communications

21726 line.

21727 **RATIONALE**

21728 None.

21729 **FUTURE DIRECTIONS**

21730 None.

21731 **SEE ALSO**

21732 The Base Definitions volume of IEEE Std 1003.1-200x, <unistd.h>

21733 **CHANGE HISTORY**

21734 First released in Issue 1. Derived from Issue 1 of the SVID.

21735 **Issue 6**

21736 The following new requirements on POSIX implementations derive from alignment with the

21737 Single UNIX Specification:

21738

21738 • The optional setting of *errno* to indicate an error is added.

21739 • The [EBADF] and [ENOTTY] optional error conditions are added.

21740 **Issue 7**

21741 SD5-XSH-ERN-100 is applied, correcting the definition of the [ENOTTY] error condition.

21742 **NAME**

21743 isblank, isblank_l — test for a blank character

21744 **SYNOPSIS**

21745 #include <ctype.h>

21746 int isblank(int c);

21747 CX int isblank_l(int c, locale_t locale);

21748 **DESCRIPTION**

21749 CX For *isblank()*: The functionality described on this reference page is aligned with the ISO C
 21750 standard. Any conflict between the requirements described here and the ISO C standard is
 21751 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21752 CX The *isblank()* and *isblank_l()* functions shall test whether *c* is a character of class **blank** in the
 21753 current locale of the process, or in the locale represented by *locale*, respectively; see the Base
 21754 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

21755 The *c* argument is a type **int**, the value of which the application shall ensure is a character
 21756 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
 21757 any other value, the behavior is undefined.

21758 **RETURN VALUE**

21759 CX The *isblank()* and *isblank_l()* functions shall return non-zero if *c* is a <blank>; otherwise, they
 21760 shall return 0.

21761 **ERRORS**21762 The *isblank_l()* function may fail if:

21763 CX [EINVAL] *locale* is not a valid locale object handle.

21764 **EXAMPLES**

21765 None.

21766 **APPLICATION USAGE**

21767 To ensure applications portability, especially across natural languages, only these functions and
 21768 the functions in the reference pages listed in the SEE ALSO section should be used for character
 21769 classification.

21770 **RATIONALE**

21771 None.

21772 **FUTURE DIRECTIONS**

21773 None.

21774 **SEE ALSO**

21775 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*,
 21776 *isxdigit()*, *setlocale()*, *uselocale()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7,
 21777 Locale, <ctype.h>, <locale.h>

21778 **CHANGE HISTORY**

21779 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21780 **Issue 7**

21781 The *isblank_l()* function is added from The Open Group Technical Standard, 2006, Extended API
 21782 Set Part 4.

21783 **NAME**
 21784 `isctrl, isctrl_l` — test for a control character

21785 **SYNOPSIS**
 21786 `#include <ctype.h>`
 21787 `int isctrl(int c);`
 21788 CX `int isctrl_l(int c, locale_t locale);`

21789 **DESCRIPTION**
 21790 CX For `isctrl()`: The functionality described on this reference page is aligned with the ISO C
 21791 standard. Any conflict between the requirements described here and the ISO C standard is
 21792 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21793 CX The `isctrl()` and `isctrl_l()` functions shall test whether `c` is a character of class **cntrl** in the
 21794 current locale of the process, or in the locale represented by `locale`, respectively; see the Base
 21795 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

21796 The `c` argument is a type **int**, the value of which the application shall ensure is a character
 21797 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
 21798 any other value, the behavior is undefined.

21799 **RETURN VALUE**
 21800 CX The `isctrl()` and `isctrl_l()` functions shall return non-zero if `c` is a control character; otherwise,
 21801 they shall return 0.

21802 **ERRORS**
 21803 The `isctrl_l()` function may fail if:

21804 CX **[EINVAL]** `locale` is not a valid locale object handle.

21805 **EXAMPLES**
 21806 None.

21807 **APPLICATION USAGE**
 21808 To ensure applications portability, especially across natural languages, only these functions and
 21809 the functions in the reference pages listed in the SEE ALSO section should be used for character
 21810 classification.

21811 **RATIONALE**
 21812 None.

21813 **FUTURE DIRECTIONS**
 21814 None.

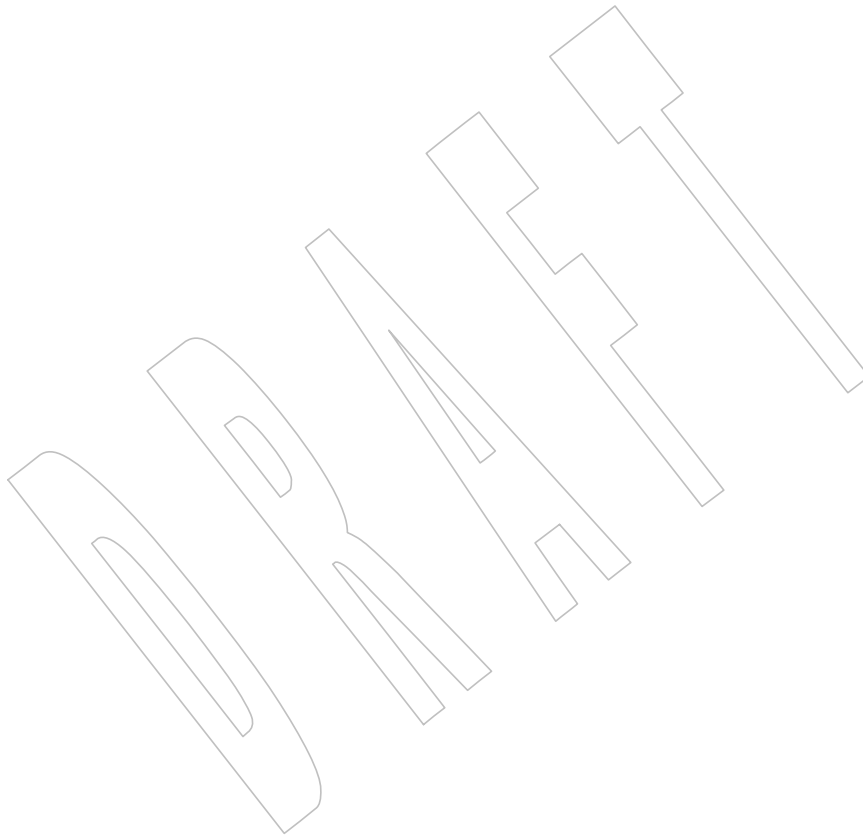
21815 **SEE ALSO**
 21816 [*isalnum\(\)*](#), [*isalpha\(\)*](#), [*isblank\(\)*](#), [*isdigit\(\)*](#), [*isgraph\(\)*](#), [*islower\(\)*](#), [*isprint\(\)*](#), [*ispunct\(\)*](#), [*isspace\(\)*](#),
 21817 [*isupper\(\)*](#), [*isxdigit\(\)*](#), [*setlocale\(\)*](#), [*uselocale\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x,
 21818 Chapter 7, Locale, `<ctype.h>`, `<locale.h>`

21819 **CHANGE HISTORY**
 21820 First released in Issue 1. Derived from Issue 1 of the SVID.

21821 **Issue 6**
 21822 The normative text is updated to avoid use of the term “must” for application requirements.

21823
21824
21825**Issue 7**

The *iscntrl_I()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



21826 **NAME**
 21827 isdigit, isdigit_l — test for a decimal digit

21828 **SYNOPSIS**

21829 #include <ctype.h>
 21830 int isdigit(int c);
 21831 CX int isdigit_l(int c, locale_t locale);

21832 **DESCRIPTION**

21833 CX For *isdigit()*: The functionality described on this reference page is aligned with the ISO C
 21834 standard. Any conflict between the requirements described here and the ISO C standard is
 21835 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21836 CX The *isdigit()* and *isdigit_l()* functions shall test whether *c* is a character of class **digit** in the
 21837 CX current locale of the process, or in the locale represented by *locale*, respectively; see the Base
 21838 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

21839 The *c* argument is an **int**, the value of which the application shall ensure is a character
 21840 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
 21841 any other value, the behavior is undefined.

21842 **RETURN VALUE**

21843 CX The *isdigit()* and *isdigit_l()* functions shall return non-zero if *c* is a decimal digit; otherwise,
 21844 they shall return 0.

21845 **ERRORS**

21846 The *isdigit_l()* function may fail if:

21847 CX [EINVAL] *locale* is not a valid locale object handle.

21848 **EXAMPLES**

21849 None.

21850 **APPLICATION USAGE**

21851 To ensure applications portability, especially across natural languages, only these functions and
 21852 the functions in the reference pages listed in the SEE ALSO section should be used for character
 21853 classification.

21854 **RATIONALE**

21855 None.

21856 **FUTURE DIRECTIONS**

21857 None.

21858 **SEE ALSO**

21859 *isalnum()*, *isalpha()*, *isblank()*, *iscntrl()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*,
 21860 *isupper()*, *isxdigit()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale,
 21861 <ctype.h>, <locale.h>

21862 **CHANGE HISTORY**

21863 First released in Issue 1. Derived from Issue 1 of the SVID.

21864 **Issue 6**

21865 The normative text is updated to avoid use of the term “must” for application requirements.

21866
21867
21868**Issue 7**

The *isdigit_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



21869 **NAME**21870 `isfinite` — test for finite value21871 **SYNOPSIS**21872 `#include <math.h>`21873 `int isfinite(real-floating x);`21874 **DESCRIPTION**

21875 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 21876 conflict between the requirements described here and the ISO C standard is unintentional. This
 21877 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21878 The *isfinite()* macro shall determine whether its argument has a finite value (zero, subnormal, or
 21879 normal, and not infinite or NaN). First, an argument represented in a format wider than its
 21880 semantic type is converted to its semantic type. Then determination is based on the type of the
 21881 argument.

21882 **RETURN VALUE**21883 The *isfinite()* macro shall return a non-zero value if and only if its argument has a finite value.21884 **ERRORS**

21885 No errors are defined.

21886 **EXAMPLES**

21887 None.

21888 **APPLICATION USAGE**

21889 None.

21890 **RATIONALE**

21891 None.

21892 **FUTURE DIRECTIONS**

21893 None.

21894 **SEE ALSO**

21895 *fpclassify()*, *isinf()*, *isnan()*, *isnormal()*, *signbit()*, the Base Definitions volume of
 21896 IEEE Std 1003.1-200x **<math.h>**

21897 **CHANGE HISTORY**

21898 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21899 **NAME**
 21900 isgraph, isgraph_l — test for a visible character

21901 **SYNOPSIS**
 21902 #include <ctype.h>

21903 int isgraph(int c);
 21904 CX int isgraph_l(int c, locale_t locale);

21905 DESCRIPTION

21906 CX For *isgraph()*: The functionality described on this reference page is aligned with the ISO C
 21907 standard. Any conflict between the requirements described here and the ISO C standard is
 21908 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21909 CX The *isgraph()* and *isgraph_l()* functions shall test whether *c* is a character of class **graph** in the
 21910 current locale of the process, or in the locale represented by *locale*, respectively; see the Base
 21911 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

21912 The *c* argument is an **int**, the value of which the application shall ensure is a character
 21913 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
 21914 any other value, the behavior is undefined.

21915 RETURN VALUE

21916 CX The *isgraph()* and *isgraph_l()* functions shall return non-zero if *c* is a character with a visible
 21917 representation; otherwise, they shall return 0.

21918 ERRORS

21919 The *isgraph_l()* function may fail if:

21920 CX [EINVAL] *locale* is not a valid locale object handle.

21921 EXAMPLES

21922 None.

21923 APPLICATION USAGE

21924 To ensure applications portability, especially across natural languages, only these functions and
 21925 the functions in the reference pages listed in the SEE ALSO section should be used for character
 21926 classification.

21927 RATIONALE

21928 None.

21929 FUTURE DIRECTIONS

21930 None.

21931 SEE ALSO

21932 *isalnum()*, *isalpha()*, *isblank()*, *iscntrl()*, *isdigit()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*,
 21933 *isxdigit()*, *setlocale()*, *uselocale()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7,
 21934 Locale, <ctype.h>, <locale.h>

21935 CHANGE HISTORY

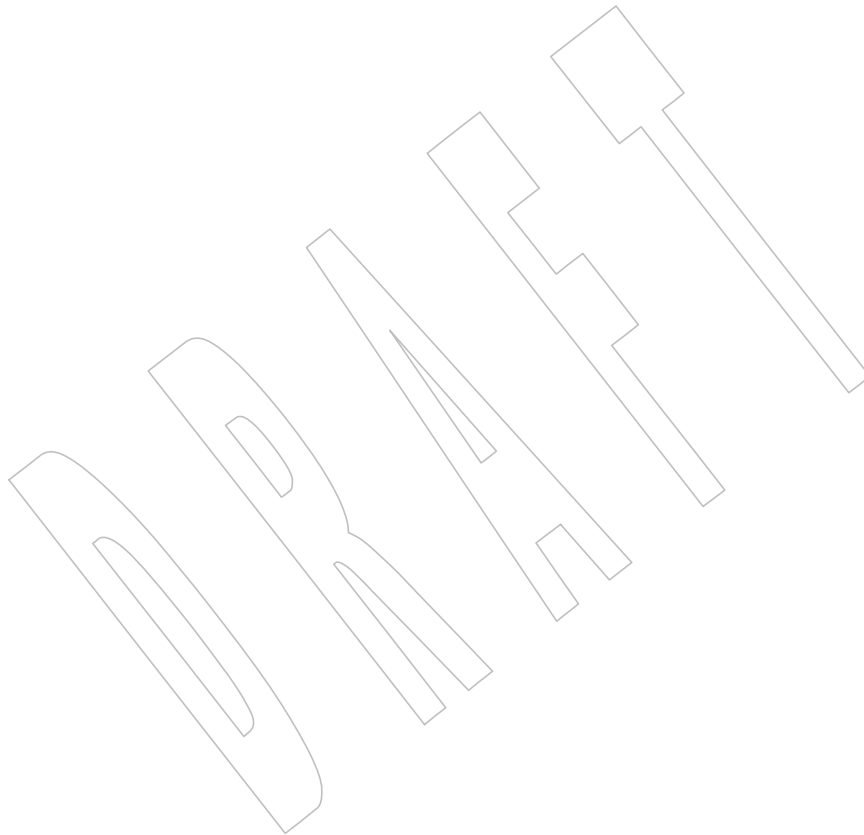
21936 First released in Issue 1. Derived from Issue 1 of the SVID.

21937 Issue 6

21938 The normative text is updated to avoid use of the term “must” for application requirements.

isgraph()*System Interfaces***Issue 7**

The *isgraph_l()* function is added from The Open Group Technical Standard, 2006, Extended API



21942 **NAME**21943 `isgreater` — test if x greater than y 21944 **SYNOPSIS**21945 `#include <math.h>`21946 `int isgreater(real-floating x , real-floating y);`21947 **DESCRIPTION**21948 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21949 conflict between the requirements described here and the ISO C standard is unintentional. This
21950 volume of IEEE Std 1003.1-200x defers to the ISO C standard.21951 The `isgreater()` macro shall determine whether its first argument is greater than its second
21952 argument. The value of `isgreater(x , y)` shall be equal to $(x) > (y)$; however, unlike $(x) > (y)$,
21953 `isgreater(x , y)` shall not raise the invalid floating-point exception when x and y are unordered.21954 **RETURN VALUE**21955 Upon successful completion, the `isgreater()` macro shall return the value of $(x) > (y)$.21956 If x or y is NaN, 0 shall be returned.21957 **ERRORS**

21958 No errors are defined.

21959 **EXAMPLES**

21960 None.

21961 **APPLICATION USAGE**21962 The relational and equality operators support the usual mathematical relationships between
21963 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21964 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21965 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21966 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21967 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21968 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21969 indicates that the argument shall be an expression of **real-floating** type.21970 **RATIONALE**

21971 None.

21972 **FUTURE DIRECTIONS**

21973 None.

21974 **SEE ALSO**21975 [*isgreaterequal\(\)*](#), [*isless\(\)*](#), [*islessequal\(\)*](#), [*islessgreater\(\)*](#), [*isunordered\(\)*](#), the Base Definitions volume of
21976 IEEE Std 1003.1-200x `<math.h>`21977 **CHANGE HISTORY**

21978 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21979 **NAME**21980 `isgreaterequal` — test if x is greater than or equal to y 21981 **SYNOPSIS**21982 `#include <math.h>`21983 `int isgreaterequal(real-floating x, real-floating y);`21984 **DESCRIPTION**21985 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21986 conflict between the requirements described here and the ISO C standard is unintentional. This
21987 volume of IEEE Std 1003.1-200x defers to the ISO C standard.21988 The `isgreaterequal()` macro shall determine whether its first argument is greater than or equal to
21989 its second argument. The value of `isgreaterequal(x, y)` shall be equal to $(x) \geq (y)$; however, unlike
21990 $(x) \geq (y)$, `isgreaterequal(x, y)` shall not raise the invalid floating-point exception when x and y are
21991 unordered.21992 **RETURN VALUE**21993 Upon successful completion, the `isgreaterequal()` macro shall return the value of $(x) \geq (y)$.21994 If x or y is NaN, 0 shall be returned.21995 **ERRORS**

21996 No errors are defined.

21997 **EXAMPLES**

21998 None.

21999 **APPLICATION USAGE**22000 The relational and equality operators support the usual mathematical relationships between
22001 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
22002 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
22003 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
22004 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
22005 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
22006 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
22007 indicates that the argument shall be an expression of **real-floating** type.22008 **RATIONALE**

22009 None.

22010 **FUTURE DIRECTIONS**

22011 None.

22012 **SEE ALSO**22013 [*isgreater\(\)*](#), [*isless\(\)*](#), [*islessequal\(\)*](#), [*islessgreater\(\)*](#), [*isunordered\(\)*](#), the Base Definitions volume of
22014 IEEE Std 1003.1-200x `<math.h>`22015 **CHANGE HISTORY**

22016 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

22017 **NAME**

22018 isinf — test for infinity

22019 **SYNOPSIS**

22020 #include <math.h>

22021 int isinf(real-floating x);

22022 **DESCRIPTION**

22023 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 22024 conflict between the requirements described here and the ISO C standard is unintentional. This
 22025 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22026 The *isinf()* macro shall determine whether its argument value is an infinity (positive or
 22027 negative). First, an argument represented in a format wider than its semantic type is converted
 22028 to its semantic type. Then determination is based on the type of the argument.

22029 **RETURN VALUE**22030 The *isinf()* macro shall return a non-zero value if and only if its argument has an infinite value.22031 **ERRORS**

22032 No errors are defined.

22033 **EXAMPLES**

22034 None.

22035 **APPLICATION USAGE**

22036 None.

22037 **RATIONALE**

22038 None.

22039 **FUTURE DIRECTIONS**

22040 None.

22041 **SEE ALSO**

22042 *fpclassify()*, *isfinite()*, *isnan()*, *isnormal()*, *signbit()*, the Base Definitions volume of
 22043 IEEE Std 1003.1-200x <math.h>

22044 **CHANGE HISTORY**

22045 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

22046 **NAME**22047 `isless` — test if x is less than y 22048 **SYNOPSIS**22049 `#include <math.h>`22050 `int isless(real-floating x , real-floating y);`22051 **DESCRIPTION**22052 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22053 conflict between the requirements described here and the ISO C standard is unintentional. This
22054 volume of IEEE Std 1003.1-200x defers to the ISO C standard.22055 The `isless()` macro shall determine whether its first argument is less than its second argument.
22056 The value of `isless(x , y)` shall be equal to $(x) < (y)$; however, unlike $(x) < (y)$, `isless(x , y)` shall not
22057 raise the invalid floating-point exception when x and y are unordered.22058 **RETURN VALUE**22059 Upon successful completion, the `isless()` macro shall return the value of $(x) < (y)$.22060 If x or y is NaN, 0 shall be returned.22061 **ERRORS**

22062 No errors are defined.

22063 **EXAMPLES**

22064 None.

22065 **APPLICATION USAGE**22066 The relational and equality operators support the usual mathematical relationships between
22067 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
22068 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
22069 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
22070 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
22071 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
22072 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
22073 indicates that the argument shall be an expression of **real-floating** type.22074 **RATIONALE**

22075 None.

22076 **FUTURE DIRECTIONS**

22077 None.

22078 **SEE ALSO**22079 [isgreater\(\)](#), [isgreaterequal\(\)](#), [islessequal\(\)](#), [islessgreater\(\)](#), [isunordered\(\)](#), the Base Definitions volume
22080 of IEEE Std 1003.1-200x, **<math.h>**22081 **CHANGE HISTORY**

22082 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

islessequal — test if x is less than or equal to y

SYNOPSIS

```
#include <math.h>

int islessequal(real-floating x, real-floating y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *islessequal()* macro shall determine whether its first argument is less than or equal to its second argument. The value of *islessequal*(x , y) shall be equal to $(x) \leq (y)$; however, unlike $(x) \leq (y)$, *islessequal*(x , y) shall not raise the invalid floating-point exception when x and y are unordered.

RETURN VALUE

Upon successful completion, the *islessequal()* macro shall return the value of $(x) \leq (y)$.

If x or y is NaN, 0 shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

22121 **NAME**22122 `islessgreater` — test if x is less than or greater than y 22123 **SYNOPSIS**22124 `#include <math.h>`22125 `int islessgreater(real-floating x , real-floating y);`22126 **DESCRIPTION**22127 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22128 conflict between the requirements described here and the ISO C standard is unintentional. This
22129 volume of IEEE Std 1003.1-200x defers to the ISO C standard.22130 The `islessgreater()` macro shall determine whether its first argument is less than or greater than
22131 its second argument. The `islessgreater(x , y)` macro is similar to $(x) < (y) \parallel (x) > (y)$; however,
22132 `islessgreater(x , y)` shall not raise the invalid floating-point exception when x and y are unordered
22133 (nor shall it evaluate x and y twice).22134 **RETURN VALUE**22135 Upon successful completion, the `islessgreater()` macro shall return the value of
22136 $(x) < (y) \parallel (x) > (y)$.22137 If x or y is NaN, 0 shall be returned.22138 **ERRORS**

22139 No errors are defined.

22140 **EXAMPLES**

22141 None.

22142 **APPLICATION USAGE**22143 The relational and equality operators support the usual mathematical relationships between
22144 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
22145 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
22146 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
22147 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
22148 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
22149 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
22150 indicates that the argument shall be an expression of **real-floating** type.22151 **RATIONALE**

22152 None.

22153 **FUTURE DIRECTIONS**

22154 None.

22155 **SEE ALSO**22156 `isgreater()`, `isgreaterequal()`, `isless()`, `islessequal()`, `isunordered()`, the Base Definitions volume of
22157 IEEE Std 1003.1-200x `<math.h>`22158 **CHANGE HISTORY**

22159 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

islower, islower_l — test for a lowercase letter

SYNOPSIS

```
#include <ctype.h>

int islower(int c);
CX int islower_l(int c, locale_t locale);
```

DESCRIPTION

CX For *islower()*: The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *islower()* and *islower_l()* functions shall test whether *c* is a character of class **lower** in the current locale of the process, or in the locale represented by *locale*, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *c* argument is an **int** **5Es**

```

22204         keystr[len++] = c;
22205     }
22206     ...

22207     /* Example 2 -- using islower_l() */
22208     #include <ctype.h>
22209     #include <stdlib.h>
22210     #include <locale.h>
22211     ...
22212     char *keystr;
22213     int elementlen, len;
22214     char c;
22215     ...
22216     locale_t loc = newlocale (LC_ALL_MASK, "", (locale_t) 0);
22217     ...
22218     len = 0;
22219     while (len < elementlen) {
22220         c = (char) (rand() % 256);
22221         ...
22222         if (islower_l(c, loc))
22223             keystr[len++] = c;
22224         }
22225     ...

```

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only these functions and the functions in the reference pages listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum(), *isalpha()*, *isblank()*, *iscntrl()*, *isdigit()*, *isgraph()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*, *setlocale()*, *uselocale()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale, **<ctype.h>**, **<locale.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The normative text is updated to avoid use of the term “must” for application requirements.

An example is added.

Issue 7

The *islower_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

22246 **NAME**

22247 isnan — test for a NaN

22248 **SYNOPSIS**

22249 #include <math.h>

22250 int isnan(real-floating x);

22251 **DESCRIPTION**

22252 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 22253 conflict between the requirements described here and the ISO C standard is unintentional. This
 22254 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22255 The *isnan()* macro shall determine whether its argument value is a NaN. First, an argument
 22256 represented in a format wider than its semantic type is converted to its semantic type. Then
 22257 determination is based on the type of the argument.

22258 **RETURN VALUE**22259 The *isnan()* macro shall return a non-zero value if and only if its argument has a NaN value.22260 **ERRORS**

22261 No errors are defined.

22262 **EXAMPLES**

22263 None.

22264 **APPLICATION USAGE**

22265 None.

22266 **RATIONALE**

22267 None.

22268 **FUTURE DIRECTIONS**

22269 None.

22270 **SEE ALSO**

22271 *fpclassify()*, *isfinite()*, *isinf()*, *isnormal()*, *signbit()*, the Base Definitions volume of
 22272 IEEE Std 1003.1-200x, <math.h>

22273 **CHANGE HISTORY**

22274 First released in Issue 3.

22275 **Issue 5**

22276 The DESCRIPTION is updated to indicate the return value when NaN is not supported. This
 22277 text was previously published in the APPLICATION USAGE section.

22278 **Issue 6**

22279 Re-written for alignment with the ISO/IEC 9899:1999 standard.

NAME

isnormal — test for a normal value

SYNOPSIS

```
#include <math.h>

int isnormal(real-floating x);
```

DESCRIPTION

cx The functionality described on this reference page is aligned with the ISO C standard. Any

22311 **NAME**
 22312 isprint, isprint_l — test for a printable character

22313 **SYNOPSIS**

22314 #include <ctype.h>
 22315 int isprint(int c);
 22316 CX int isprint_l(int c, locale_t locale);

22317 **DESCRIPTION**

22318 CX For *isprint()*: The functionality described on this reference page is aligned with the ISO C
 22319 standard. Any conflict between the requirements described here and the ISO C standard is
 22320 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22321 CX The *isprint()* and *isprint_l()* functions shall test whether *c* is a character of class **print** in the
 22322 current locale of the process, or in the locale represented by *locale*, respectively; see the Base
 22323 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

22324 The *c* argument is an **int**, the value of which the application shall ensure is a character
 22325 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
 22326 any other value, the behavior is undefined.

22327 **RETURN VALUE**

22328 CX The *isprint()* and *isprint_l()* functions shall return non-zero if *c* is a printable character;
 22329 otherwise, they shall return 0.

22330 **ERRORS**

22331 The *isprint_l()* function may fail if:

22332 CX [EINVAL] *locale* is not a valid locale object handle.

22333 **EXAMPLES**

22334 None.

22335 **APPLICATION USAGE**

22336 To ensure applications portability, especially across natural languages, only these functions and
 22337 the functions in the reference pages listed in the SEE ALSO section should be used for character
 22338 classification.

22339 **RATIONALE**

22340 None.

22341 **FUTURE DIRECTIONS**

22342 None.

22343 **SEE ALSO**

22344 *isalnum()*, *isalpha()*, *isblank()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *ispunct()*, *isspace()*, *isupper()*,
 22345 *isxdigit()*, *setlocale()*, *uselocale()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7,
 22346 Locale, <ctype.h>, <locale.h>

22347 **CHANGE HISTORY**

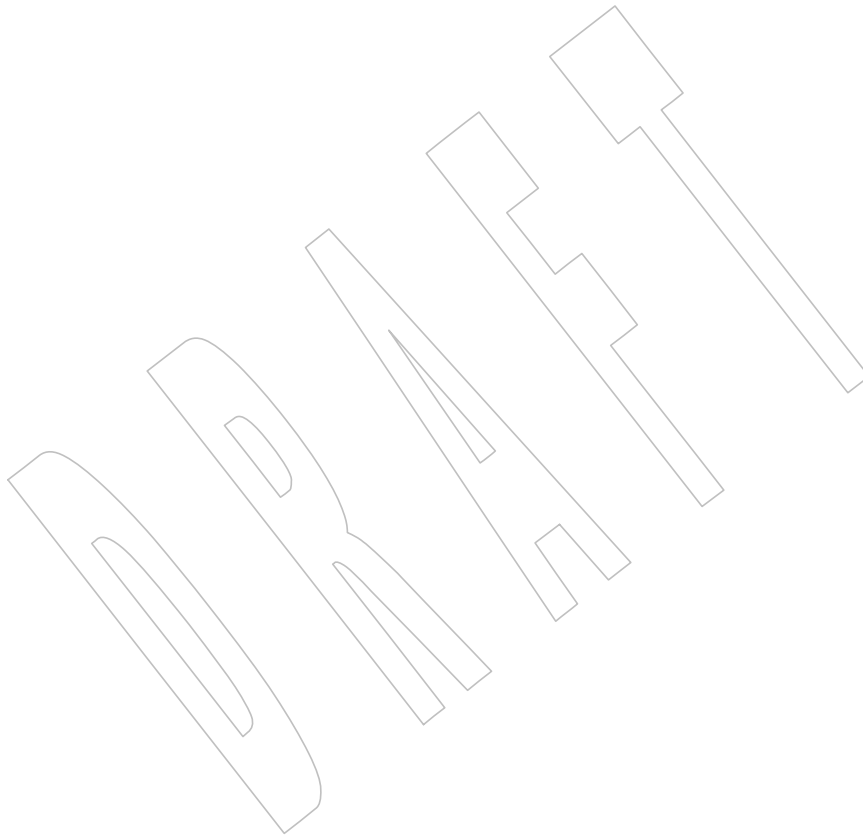
22348 First released in Issue 1. Derived from Issue 1 of the SVID.

22349 **Issue 6**

22350 The normative text is updated to avoid use of the term “must” for application requirements.

22351
22352
22353**Issue 7**

The *isprint_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



22354 **NAME**
 22355 ispunct, ispunct_l — test for a punctuation character

22356 **SYNOPSIS**
 22357 #include <ctype.h>
 22358 int ispunct(int c);
 22359 CX int ispunct_l(int c, locale_t locale);

22360 **DESCRIPTION**
 22361 CX For *ispunct()*: The functionality described on this reference page is aligned with the ISO C
 22362 standard. Any conflict between the requirements described here and the ISO C standard is
 22363 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22364 CX The *ispunct()* and *ispunct_l()* functions shall test whether *c* is a character of class **punct** in the
 22365 current locale of the process, or in the locale represented by *locale*, respectively; see the Base
 22366 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

22367 The *c* argument is an **int**, the value of which the application shall ensure is a character
 22368 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
 22369 any other value, the behavior is undefined.

22370 **RETURN VALUE**
 22371 CX The *ispunct()* and *ispunct_l()* functions shall return non-zero if *c* is a punctuation character;
 22372 otherwise, they shall return 0.

22373 **ERRORS**
 22374 The *ispunct_l()* function may fail if:

22375 CX [EINVAL] *locale* is not a valid locale object handle.

22376 **EXAMPLES**
 22377 None.

22378 **APPLICATION USAGE**
 22379 To ensure applications portability, especially across natural languages, only these functions and
 22380 the functions in the reference pages listed in the SEE ALSO section should be used for character
 22381 classification.

22382 **RATIONALE**
 22383 None.

22384 **FUTURE DIRECTIONS**
 22385 None.

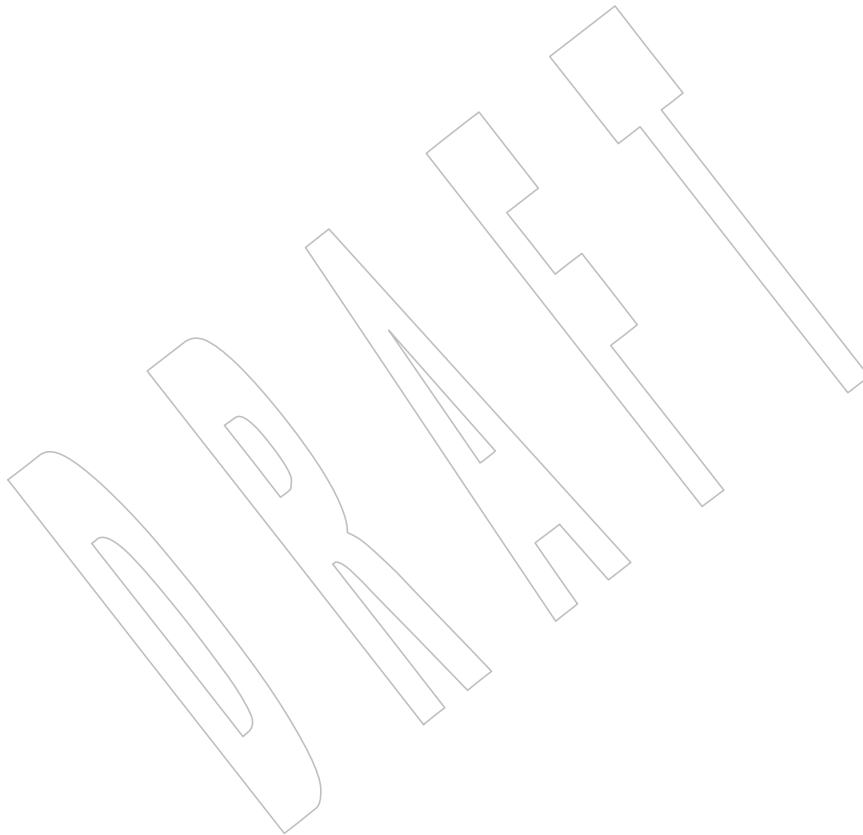
22386 **SEE ALSO**
 22387 *isalnum()*, *isalpha()*, *isblank()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *isspace()*, *isupper()*,
 22388 *isxdigit()*, *setlocale()*, *uselocale()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7,
 22389 Locale, <ctype.h>, <locale.h>

22390 **CHANGE HISTORY**
 22391 First released in Issue 1. Derived from Issue 1 of the SVID.

22392 **Issue 6**
 22393 The normative text is updated to avoid use of the term “must” for application requirements.

22394
22395
22396**Issue 7**

The *ispunct_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



22397 **NAME**
 22398 `isspace, isspace_l` — test for a white-space character

22399 **SYNOPSIS**
 22400 `#include <ctype.h>`
 22401 `int isspace(int c);`
 22402 CX `int isspace_l(int c, locale_t locale);`

22403 **DESCRIPTION**
 22404 CX For `isspace()`: The functionality described on this reference page is aligned with the ISO C
 22405 standard. Any conflict between the requirements described here and the ISO C standard is
 22406 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22407 CX The `isspace()` and `isspace_l()` functions shall test whether `c` is a character of class **space** in the
 22408 current locale of the process, or in the locale represented by `locale`, respectively; see the Base
 22409 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

22410 The `c` argument is an **int**, the value of which the application shall ensure is a character
 22411 representable as an **unsigned char** or equal to the value of the macro `EOF`. If the argument has
 22412 any other value, the behavior is undefined.

22413 **RETURN VALUE**
 22414 CX The `isspace()` and `isspace_l()` functions shall return non-zero if `c` is a white-space character;
 22415 otherwise, they shall return 0.

22416 **ERRORS**
 22417 The `isspace_l()` function may fail if:

22418 CX `[EINVAL]` `locale` is not a valid locale object handle.

22419 **EXAMPLES**
 22420 None.

22421 **APPLICATION USAGE**
 22422 To ensure applications portability, especially across natural languages, only these functions and
 22423 the functions in the reference pages listed in the SEE ALSO section should be used for character
 22424 classification.

22425 **RATIONALE**
 22426 None.

22427 **FUTURE DIRECTIONS**
 22428 None.

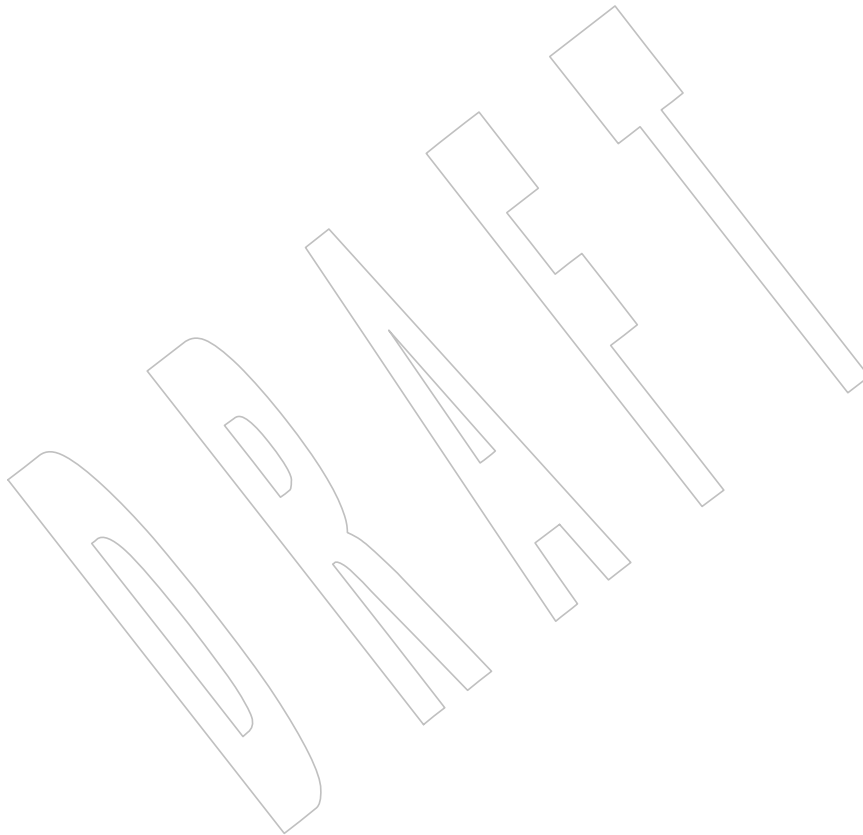
22429 **SEE ALSO**
 22430 `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isupper()`,
 22431 `isxdigit()`, `setlocale()`, `uselocale()`, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7,
 22432 Locale, `<ctype.h>`, `<locale.h>`

22433 **CHANGE HISTORY**
 22434 First released in Issue 1. Derived from Issue 1 of the SVID.

22435 **Issue 6**
 22436 The normative text is updated to avoid use of the term “must” for application requirements.

22437
22438
22439**Issue 7**

The *isspace_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



22440 **NAME**

22441 isunordered — test if arguments are unordered

22442 **SYNOPSIS**

22443 #include <math.h>

22444 int isunordered(real-floating x, real-floating y);

22445 **DESCRIPTION**

22446 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 22447 conflict between the requirements described here and the ISO C standard is unintentional. This
 22448 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22449 The *isunordered()* macro shall determine whether its arguments are unordered.22450 **RETURN VALUE**

22451 Upon successful completion, the *isunordered()* macro shall return 1 if its arguments are
 22452 unordered, and 0 otherwise.

22453 If *x* or *y* is NaN, 1 shall be returned.22454 **ERRORS**

22455 No errors are defined.

22456 **EXAMPLES**

22457 None.

22458 **APPLICATION USAGE**

22459 The relational and equality operators support the usual mathematical relationships between
 22460 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
 22461 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
 22462 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
 22463 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
 22464 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
 22465 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
 22466 indicates that the argument shall be an expression of **real-floating** type.

22467 **RATIONALE**

22468 None.

22469 **FUTURE DIRECTIONS**

22470 None.

22471 **SEE ALSO**

22472 *isgreater()*, *isgreaterequal()*, *isless()*, *islessequal()*, *islessgreater()*, the Base Definitions volume of
 22473 IEEE Std 1003.1-200x, <math.h>

22474 **CHANGE HISTORY**

22475 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

22476 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/50 is applied, correcting the RETURN
 22477 VALUE section when *x* or *y* is NaN.

22478 **NAME**
 22479 `isupper, isupper_l` — test for an uppercase letter

22480 **SYNOPSIS**
 22481 `#include <ctype.h>`
 22482 `int isupper(int c);`
 22483 CX `int isupper_l(int c, locale_t locale);`

22484 **DESCRIPTION**
 22485 CX For `isupper()`: The functionality described on this reference page is aligned with the ISO C
 22486 standard. Any conflict between the requirements described here and the ISO C standard is
 22487 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22488 CX The `isupper()` and `isupper_l()` functions shall test whether `c` is a character of class **upper** in the
 22489 current locale of the process, or in the locale represented by `locale`, respectively; see the Base
 22490 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

22491 The `c` argument is an **int**, the value of which the application shall ensure is a character
 22492 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
 22493 any other value, the behavior is undefined.

22494 **RETURN VALUE**
 22495 CX The `isupper()` and `isupper_l()` functions shall return non-zero if `c` is an uppercase letter;
 22496 otherwise, they shall return 0.

22497 **ERRORS**
 22498 The `isupper_l()` function may fail if:

22499 CX `[EINVAL]` `locale` is not a valid locale object handle.

22500 **EXAMPLES**
 22501 None.

22502 **APPLICATION USAGE**
 22503 To ensure applications portability, especially across natural languages, only these functions and
 22504 the functions in the reference pages listed in the SEE ALSO section should be used for character
 22505 classification.

22506 **RATIONALE**
 22507 None.

22508 **FUTURE DIRECTIONS**
 22509 None.

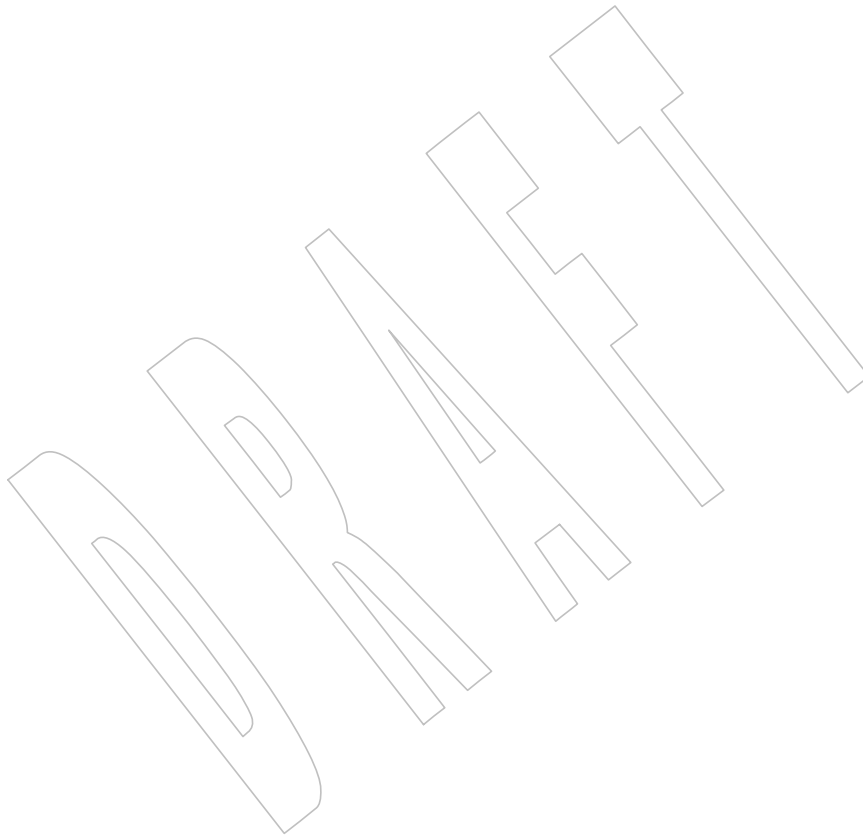
22510 **SEE ALSO**
 22511 `isalnum()`, `isalpha()`, `isblank()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`,
 22512 `isxdigit()`, `setlocale()`, `uselocale()`, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7,
 22513 Locale, `<ctype.h>`, `<locale.h>`

22514 **CHANGE HISTORY**
 22515 First released in Issue 1. Derived from Issue 1 of the SVID.

22516 **Issue 6**
 22517 The normative text is updated to avoid use of the term “must” for application requirements.

22518
22519
22520**Issue 7**

The *isupper_1()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



22521 **NAME**22522 `iswalnum, iswalnum_l` — test for an alphanumeric wide-character code22523 **SYNOPSIS**22524 `#include <wctype.h>`22525 `int iswalnum(wint_t wc);`22526 CX `int iswalnum_l(wint_t wc, locale_t locale);`22527 **DESCRIPTION**22528 CX For `iswalnum()`: The functionality described on this reference page is aligned with the ISO C
22529 standard. Any conflict between the requirements described here and the ISO C standard is
22530 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.22531 CX The `iswalnum()` and `iswalnum_l()` functions shall test whether `wc` is a wide-character code
22532 representing a character of class **alpha** or **digit** in the current locale of the process, or in the
22533 locale represented by `locale`, respectively; see the Base Definitions volume of
22534 IEEE Std 1003.1-200x, Chapter 7, Locale.22535 The `wc` argument is a **wint_t**, the value of which the application shall ensure is a wide-character
22536 code corresponding to a valid character in the current locale, or equal to the value of the macro
22537 WEOF. If the argument has any other value, the behavior is undefined.22538 **RETURN VALUE**22539 CX The `iswalnum()` and `iswalnum_l()` functions shall return non-zero if `wc` is an alphanumeric
22540 wide-character code; otherwise, they shall return 0.22541 **ERRORS**22542 The `iswalnum_l()` function may fail if:22543 CX **[EINVAL]** `locale` is not a valid locale object handle.22544 **EXAMPLES**

22545 None.

22546 **APPLICATION USAGE**22547 To ensure applications portability, especially across natural languages, only these functions and
22548 the functions in the reference pages listed in the SEE ALSO section should be used for character
22549 classification.22550 **RATIONALE**

22551 None.

22552 **FUTURE DIRECTIONS**

22553 None.

22554 **SEE ALSO**22555 `iswalpha()`, `iswcntrl()`, `iswctype()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`,
22556 `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `uselocale()`, the Base Definitions volume of
22557 IEEE Std 1003.1-200x, Chapter 7, Locale, `<locale.h>`, `<stdio.h>`, `<wctype.h>`22558 **CHANGE HISTORY**

22559 First released as a World-wide Portability Interface in Issue 4.

22560

Issue 5

22561

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

22562

22563

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the `<wctype.h>` header rather than `<wchar.h>`.

22564

22565

Issue 6

22566

The normative text is updated to avoid use of the term “must” for application requirements.

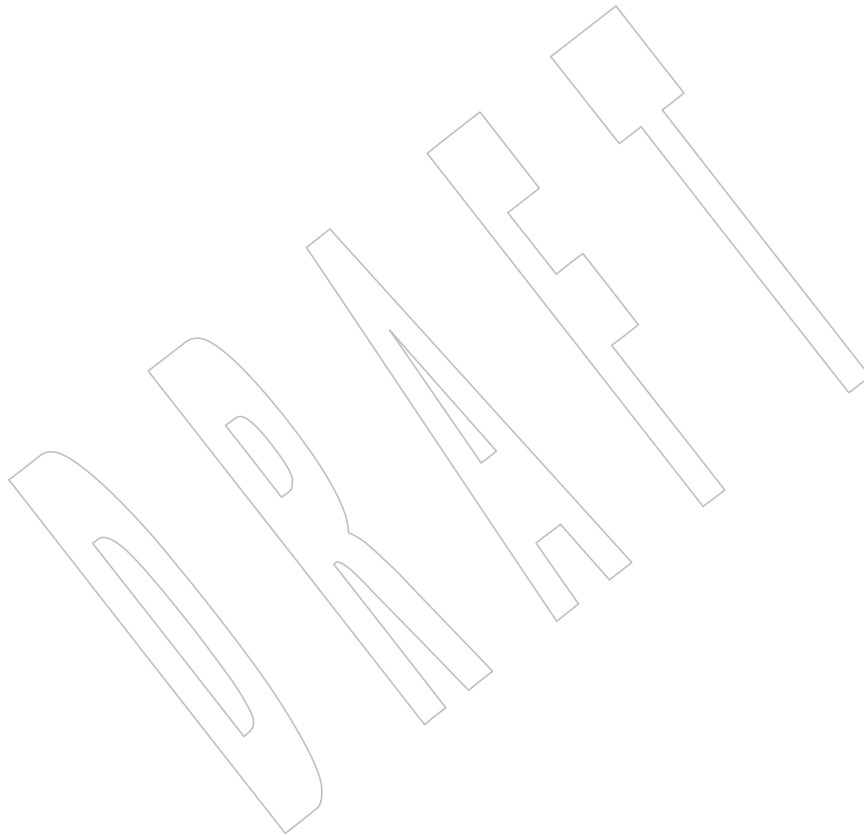
22567

Issue 7

22568

The `iswalnum_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

22569



22570 **NAME**22571 `iswalpha, iswalpha_l` — test for an alphabetic wide-character code22572 **SYNOPSIS**22573 `#include <wctype.h>`22574 `int iswalpha(wint_t wc);`22575 CX `int iswalpha_l(wint_t wc, locale_t locale);`22576 **DESCRIPTION**22577 CX For `iswalpha()`: The functionality described on this reference page is aligned with the ISO C
22578 standard. Any conflict between the requirements described here and the ISO C standard is
22579 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.22580 CX The `iswalpha()` and `iswalpha_l()` functions shall test whether `wc` is a wide-character code
22581 CX representing a character of class **alpha** in the current locale of the process, or in the locale
22582 represented by `locale`, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
22583 Chapter 7, Locale.22584 The `wc` argument is a **wint_t**, the value of which the application shall ensure is a wide-character
22585 code corresponding to a valid character in the current locale, or equal to the value of the macro
22586 WEOF. If the argument has any other value, the behavior is undefined.22587 **RETURN VALUE**22588 CX The `iswalpha()` and `iswalpha_l()` functions shall return non-zero if `wc` is an alphabetic wide-
22589 character code; otherwise, they shall return 0.22590 **ERRORS**22591 The `iswalpha_l()` function may fail if:22592 CX **[EINVAL]** `locale` is not a valid locale object handle.22593 **EXAMPLES**

22594 None.

22595 **APPLICATION USAGE**22596 To ensure applications portability, especially across natural languages, only these functions and
22597 the functions in the reference pages listed in the SEE ALSO section should be used for character
22598 classification.22599 **RATIONALE**

22600 None.

22601 **FUTURE DIRECTIONS**

22602 None.

22603 **SEE ALSO**22604 `iswalnum()`, `iswcntrl()`, `iswctype()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`,
22605 `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `uselocale()`, the Base Definitions volume of
22606 IEEE Std 1003.1-200x, Chapter 7, Locale, `<locale.h>`, `<stdio.h>`, `<wctype.h>`22607 **CHANGE HISTORY**

22608 First released in Issue 4.

22609

Issue 5

22610

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

22611

22612

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the `<wctype.h>` header rather than `<wchar.h>`.

22613

22614

Issue 6

22615

The normative text is updated to avoid use of the term “must” for application requirements.

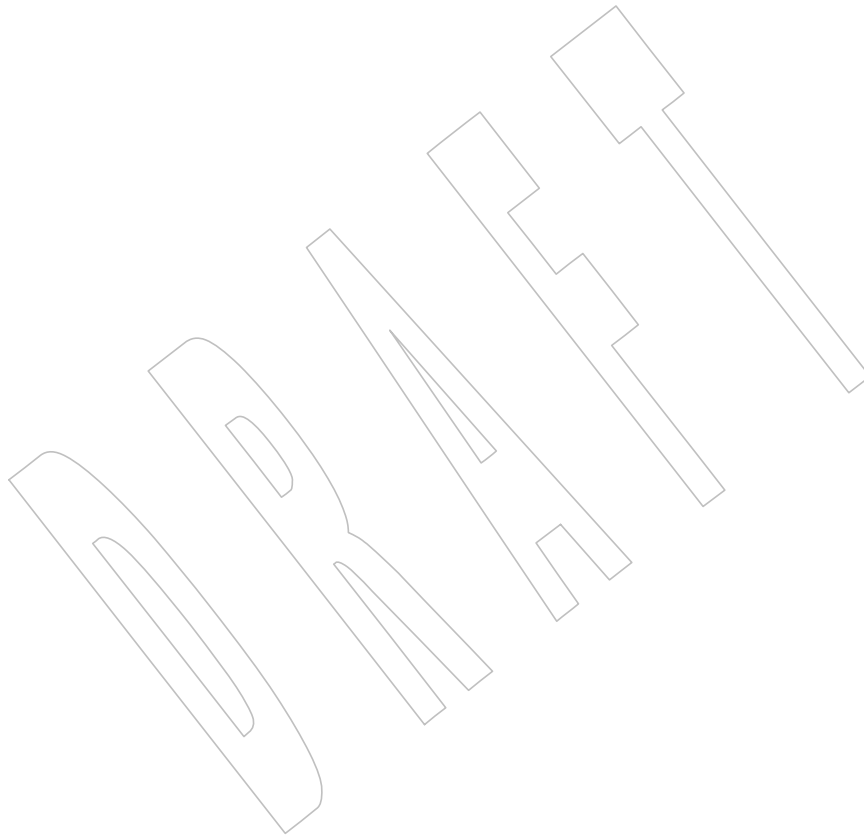
22616

Issue 7

22617

The `iswalpha_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

22618



22619 **NAME**22620 `iswblank`, `iswblank_l` — test for a blank wide-character code22621 **SYNOPSIS**22622 `#include <wctype.h>`22623 `int iswblank(wint_t wc);`22624 CX `int iswblank_l(wint_t wc, locale_t locale);`22625 **DESCRIPTION**22626 CX For `iswblank()`: The functionality described on this reference page is aligned with the ISO C
22627 standard. Any conflict between the requirements described here and the ISO C standard is
22628 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.22629 CX The `iswblank()` and `iswblank_l()` functions shall test whether `wc` is a wide-character code
22630 representing a character of class **blank** in the current locale of the process, or in the locale
22631 represented by `locale`, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
22632 Chapter 7, Locale.22633 The `wc` argument is a **wint_t**, the value of which the application shall ensure is a wide-character
22634 code corresponding to a valid character in the current locale, or equal to the value of the macro
22635 WEOF. If the argument has any other value, the behavior is undefined.22636 **RETURN VALUE**22637 CX The `iswblank()` and `iswblank_l()` functions shall return non-zero if `wc` is a blank wide-character
22638 code; otherwise, they shall return 0.22639 **ERRORS**22640 The `iswblank_l()` function may fail if:22641 CX **[EINVAL]** `locale` is not a valid locale object handle.22642 **EXAMPLES**

22643 None.

22644 **APPLICATION USAGE**22645 To ensure applications portability, especially across natural languages, only these functions and
22646 the functions in the reference pages listed in the SEE ALSO section should be used for character
22647 classification.22648 **RATIONALE**

22649 None.

22650 **FUTURE DIRECTIONS**

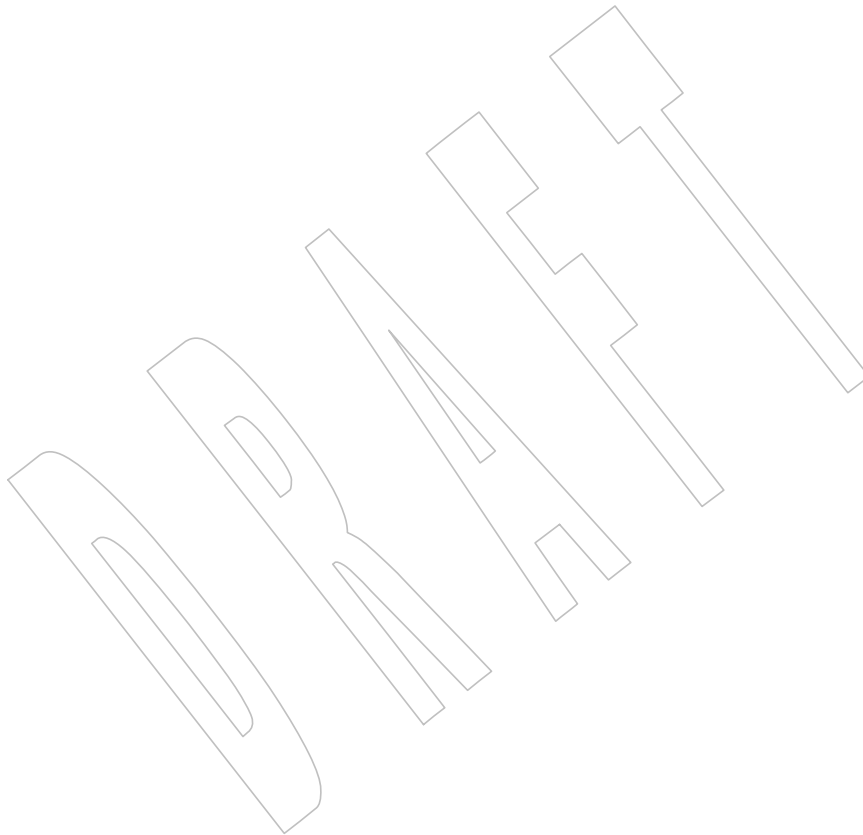
22651 None.

22652 **SEE ALSO**22653 `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswctype()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`,
22654 `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `uselocale()`, the Base Definitions volume
22655 of IEEE Std 1003.1-200x, Chapter 7, Locale, `<locale.h>`, `<stdio.h>`, `<wctype.h>`22656 **CHANGE HISTORY**

22657 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

22658
22659
22660**Issue 7**

The *iswblank_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



22661 **NAME**22662 `iswcntrl, iswcntrl_l` — test for a control wide-character code22663 **SYNOPSIS**22664 `#include <wctype.h>`22665 `int iswcntrl(wint_t wc);`22666 CX `int iswcntrl_l(wint_t wc, locale_t locale);`22667 **DESCRIPTION**22668 CX For `iswcntrl()`: The functionality described on this reference page is aligned with the ISO C
22669 standard. Any conflict between the requirements described here and the ISO C standard is
22670 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.22671 CX The `iswcntrl()` and `iswcntrl_l()` functions shall test whether `wc` is a wide-character code
22672 representing a character of class **cntrl** in the current locale of the process, or in the locale
22673 represented by `locale`, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
22674 Chapter 7, Locale.22675 The `wc` argument is a **wint_t**, the value of which the application shall ensure is a wide-character
22676 code corresponding to a valid character in the current locale, or equal to the value of the macro
22677 WEOF. If the argument has any other value, the behavior is undefined.22678 **RETURN VALUE**22679 CX The `iswcntrl()` and `iswcntrl_l()` functions shall return non-zero if `wc` is a control wide-character
22680 code; otherwise, they shall return 0.22681 **ERRORS**22682 The `iswcntrl_l()` function may fail if:22683 CX **[EINVAL]** `locale` is not a valid locale object handle.22684 **EXAMPLES**

22685 None.

22686 **APPLICATION USAGE**22687 To ensure applications portability, especially across natural languages, only these functions and
22688 the functions in the reference pages listed in the SEE ALSO section should be used for character
22689 classification.22690 **RATIONALE**

22691 None.

22692 **FUTURE DIRECTIONS**

22693 None.

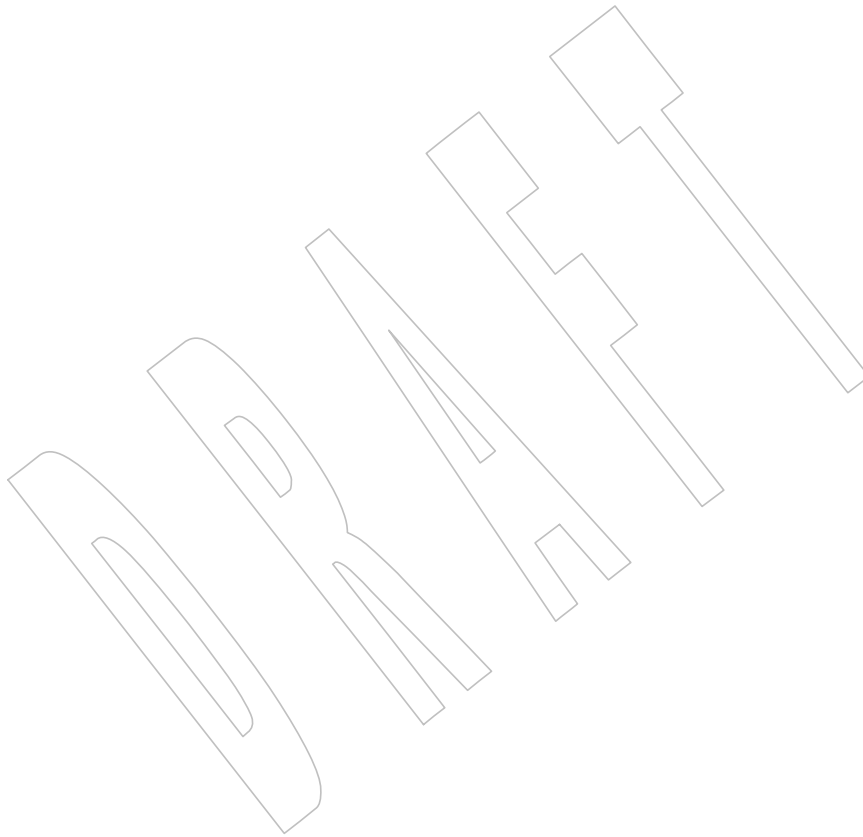
22694 **SEE ALSO**22695 `iswalnum()`, `iswalpha()`, `iswctype()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`,
22696 `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `uselocale()`, the Base Definitions volume of
22697 IEEE Std 1003.1-200x, Chapter 7, Locale, `<locale.h>`, `<wctype.h>`22698 **CHANGE HISTORY**

22699 First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are n



22710 **NAME**
 22711 `iswctype`, `iswctype_l` — test character for a specified class

SYNOPSIS

```
22712 #include <wctype.h>
22713
22714 int iswctype(wint_t wc, wctype_t charclass);
22715 CX int iswctype_l(wint_t wc, wctype_t charclass,
22716 locale_t locale);
```

DESCRIPTION

22717 **DESCRIPTION**
 22718 CX For `iswctype()`: The functionality described on this reference page is aligned with the ISO C
 22719 standard. Any conflict between the requirements described here and the ISO C standard is
 22720 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22721 CX The `iswctype()` and `iswctype_l()` functions shall determine whether the wide-character code `wc`
 22722 has the character class `charclass`, returning true or false. The `iswctype()` and `iswctype_l()`
 22723 functions are defined on WEOF and wide-character codes corresponding to the valid character
 22724 encodings in the current locale, or in the locale represented by `locale`, respectively. If the `wc`
 22725 argument is not in the domain of the function, the result is undefined. If the value of `charclass` is
 22726 invalid (that is, not obtained by a call to `wctype()` or `charclass` is invalidated by a subsequent call
 22727 to `setlocale()` that has affected category `LC_CTYPE`) the result is unspecified.

RETURN VALUE

22728 **RETURN VALUE**
 22729 CX The `iswctype()` and `iswctype_l()` functions shall return non-zero (true) if and only if `wc` has the
 22730 property described by `charclass`. If `charclass` is 0, these functions shall return 0.

ERRORS

22731 **ERRORS**
 22732 The `iswctype_l()` function may fail if:

22733 CX [EINVAL] `locale` is not a valid locale object handle.

EXAMPLES**Testing for a Valid Character**

```
22735 #include <wctype.h>
22736 ...
22737 int yes_or_no;
22738 wint_t wc;
22739 wctype_t valid_class;
22740 ...
22741 if ((valid_class=wctype("vowel")) == (wctype_t)0)
22742     /* Invalid character class. */
22743     yes_or_no=iswctype(wc,valid_class);
```

APPLICATION USAGE

22745 **APPLICATION USAGE**
 22746 The twelve strings "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower",
 22747 "print", "punct", "space", "upper", and "xdigit" are reserved for the standard
 22748 character classes. In the table below, the functions in the left column are equivalent to the
 22749 functions in the right column.

22750	<code>iswalnum(wc)</code>	<code>iswctype(wc, wctype("alnum"))</code>
22751	<code>iswalnum_l(wc, locale)</code>	<code>iswctype_l(wc, wctype("alnum"), locale)</code>
22752	<code>iswalpha(wc)</code>	<code>iswctype(wc, wctype("alpha"))</code>
22753	<code>iswalpha_l(wc, locale)</code>	<code>iswctype_l(wc, wctype("alpha"), locale)</code>

```

22754      iswblank(wc)           iswctype(wc, wctype("blank"))
22755      iswblank_l(wc, locale) iswctype_l(wc, wctype("blank"), locale)
22756      iswcntrl(wc)          iswctype(wc, wctype("cntrl"))
22757      iswcntrl_l(wc, locale) iswctype_l(wc, wctype("cntrl"), locale)
22758      iswdigit(wc)          iswctype(wc, wctype("digit"))
22759      iswdigit_l(wc, locale) iswctype_l(wc, wctype("digit"), locale)
22760      iswgraph(wc)         iswctype(wc, wctype("graph"))
22761      iswgraph_l(wc, locale) iswctype_l(wc, wctype("graph"), locale)
22762      iswlower(wc)         iswctype(wc, wctype("lower"))
22763      iswlower_l(wc, locale) iswctype_l(wc, wctype("lower"), locale)
22764      iswprint(wc)         iswctype(wc, wctype("print"))
22765      iswprint_l(wc, locale) iswctype_l(wc, wctype("print"), locale)
22766      iswpunct(wc)         iswctype(wc, wctype("punct"))
22767      iswpunct_l(wc, locale) iswctype_l(wc, wctype("punct"), locale)
22768      iswspace(wc)         iswctype(wc, wctype("space"))
22769      iswspace_l(wc, locale) iswctype_l(wc, wctype("space"), locale)
22770      iswupper(wc)         iswctype(wc, wctype("upper"))
22771      iswupper_l(wc, locale) iswctype_l(wc, wctype("upper"), locale)
22772      iswxdigit(wc)        iswctype(wc, wctype("xdigit"))
22773      iswxdigit_l(wc, locale) iswctype_l(wc, wctype("xdigit"), locale)

```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[iswalnum\(\)](#), [iswalpha\(\)](#), [iswcntrl\(\)](#), [iswdigit\(\)](#), [iswgraph\(\)](#), [iswlower\(\)](#), [iswprint\(\)](#), [iswpunct\(\)](#), [iswspace\(\)](#), [iswupper\(\)](#), [iswxdigit\(\)](#), [setlocale\(\)](#), [uselocale\(\)](#), [wctype\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, [<locale.h>](#), [<wctype.h>](#)

CHANGE HISTORY

First released as World-wide Portability Interfaces in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the [<wctype.h>](#) header rather than [<wchar.h>](#).

Issue 6

The behavior of $n=0$ is now described.

An example is added.

A new function, [iswblank\(\)](#), is added to the list in the APPLICATION USAGE.

Issue 7

The [iswctype_l\(\)](#) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

22796 **NAME**
 22797 `iswdigit, iswdigit_l` — test for a decimal digit wide-character code

22798 **SYNOPSIS**
 22799 `#include <wctype.h>`
 22800 `int iswdigit(wint_t wc);`
 22801 CX `int iswdigit_l(wint_t wc, locale_t locale);`

22802 **DESCRIPTION**
 22803 CX For `iswdigit()`: The functionality described on this reference page is aligned with the ISO C
 22804 standard. Any conflict between the requirements described here and the ISO C standard is
 22805 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22806 CX The `iswdigit()` and `iswdigit_l()` functions shall test whether `wc` is a wide-character code
 22807 CX representing a character of class **digit** in the current locale of the process, or in the locale
 22808 represented by `locale`, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
 22809 Chapter 7, Locale.

22810 The `wc` argument is a **wint_t**, the value of which the application shall ensure is a wide-character
 22811 code corresponding to a valid character in the current locale, or equal to the value of the macro
 22812 WEOF. If the argument has any other value, the behavior is undefined.

22813 **RETURN VALUE**
 22814 CX The `iswdigit()` and `iswdigit_l()` functions shall return non-zero if `wc` is a decimal digit wide-
 22815 character code; otherwise, they shall return 0.

22816 **ERRORS**
 22817 The `iswdigit_l()` function may fail if:

22818 CX **[EINVAL]** `locale` is not a valid locale object handle.

22819 **EXAMPLES**
 22820 None.

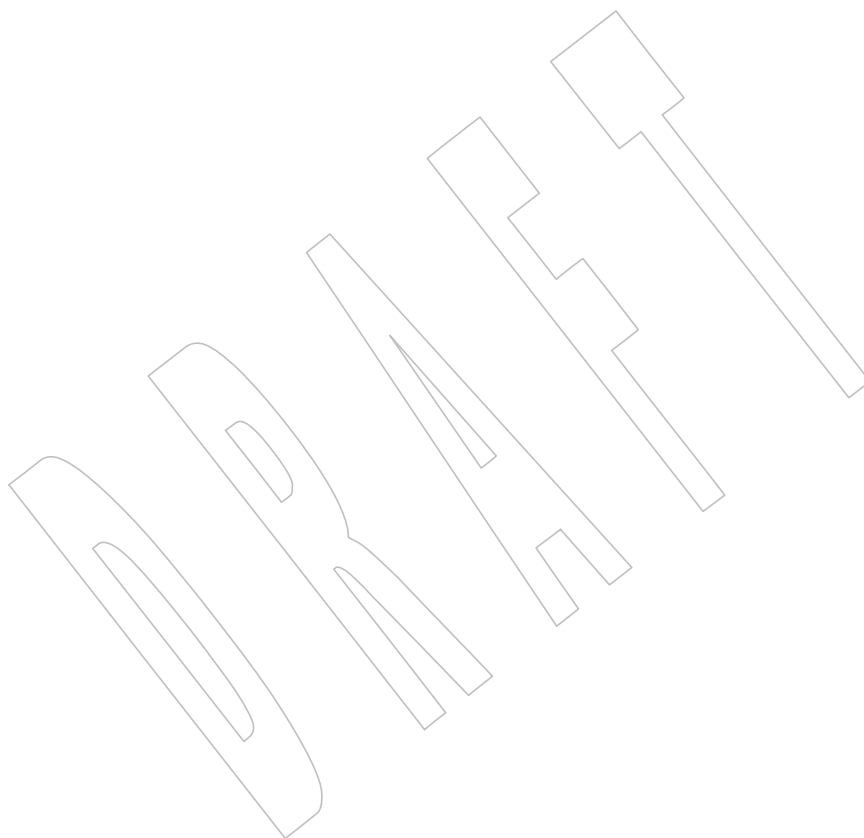
22821 **APPLICATION USAGE**
 22822 To ensure applications portability, especially across natural languages, only these functions and
 22823 the functions in the reference pages listed in the SEE ALSO section should be used for character
 22824 classification.

22825 **RATIONALE**
 22826 None.

22827 **FUTURE DIRECTIONS**
 22828 None.

22829 **SEE ALSO**
 22830 `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswctype()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`,
 22831 `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `uselocale()`, the Base Definitions volume of
 22832 IEEE Std 1003.1-200x, Chapter 7, Locale, `<locale.h>`, `<wctype.h>`

22833 **CHANGE HISTORY**
 22834 First released in Issue 4.



22845 **NAME**22846 `iswgraph, iswgraph_l` — test for a visible wide-character code22847 **SYNOPSIS**22848 `#include <wctype.h>`22849 `int iswgraph(wint_t wc);`22850 CX `int iswgraph_l(wint_t wc, locale_t locale);`22851 **DESCRIPTION**22852 CX For `iswgraph()`: The functionality described on this reference page is aligned with the ISO C
22853 standard. Any conflict between the requirements described here and the ISO C standard is
22854 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.22855 CX The `iswgraph()` and `iswgraph_l()` functions shall test whether `wc` is a wide-character code
22856 representing a character of class **graph** in the current locale of the process, or in the locale
22857 represented by `locale`, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
22858 Chapter 7, Locale.22859 The `wc` argument is a **wint_t**, the value of which the application shall ensure is a wide-character
22860 code corresponding to a valid character in the current locale, or equal to the value of the macro
22861 WEOF. If the argument has any other value, the behavior is undefined.22862 **RETURN VALUE**22863 CX The `iswgraph()` and `iswgraph_l()` functions shall return non-zero if `wc` is a wide-character code
22864 with a visible representation; otherwise, they shall return 0.22865 **ERRORS**22866 The `iswgraph_l()` function may fail if:22867 CX **[EINVAL]** `locale` is not a valid locale object handle.22868 **EXAMPLES**

22869 None.

22870 **APPLICATION USAGE**22871 To ensure applications portability, especially across natural languages, only these functions and
22872 the functions in the reference pages listed in the SEE ALSO section should be used for character
22873 classification.22874 **RATIONALE**

22875 None.

22876 **FUTURE DIRECTIONS**

22877 None.

22878 **SEE ALSO**22879 `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswctype()`, `iswdigit()`, `iswlower()`, `iswprint()`, `iswpunct()`,
22880 `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `uselocale()`, the Base Definitions volume of
22881 IEEE Std 1003.1-200x, Chapter 7, Locale, `<locale.h>`, `<wctype.h>`22882 **CHANGE HISTORY**

22883 First released in Issue 4.

22884

Issue 5

22885

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

22886

22887

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the `<wctype.h>` header rather than `<wchar.h>`.

22888

22889

Issue 6

22890

The normative text is updated to avoid use of the term “must” for application requirements.

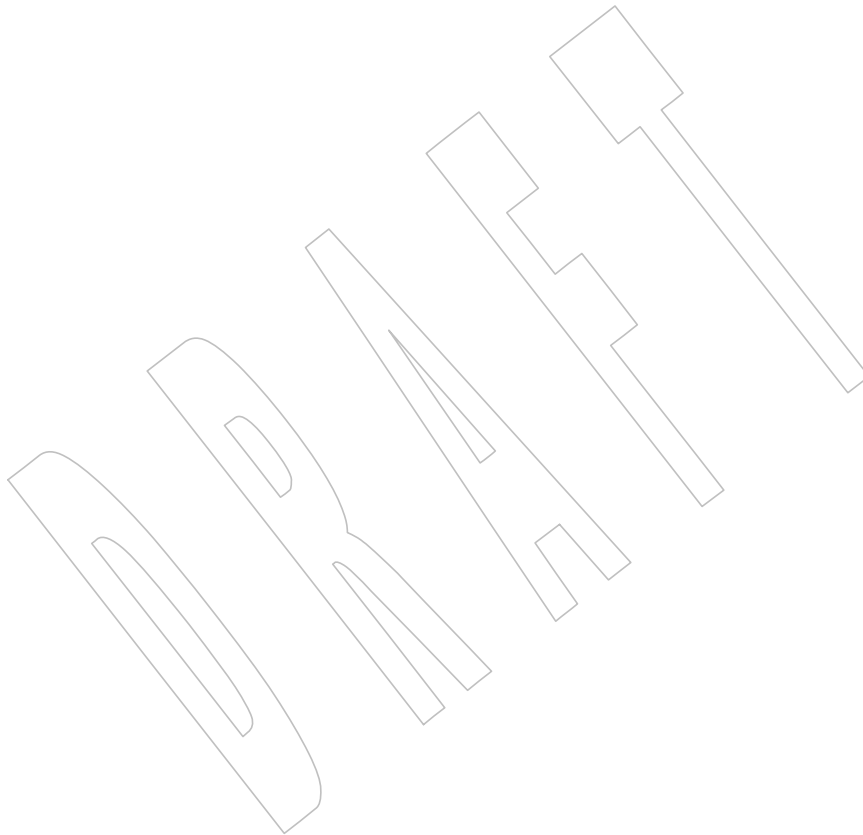
22891

Issue 7

22892

The `iswgraph_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

22893



22894 **NAME**
 22895 `iswlower, iswlower_l` — test for a lowercase letter wide-character code

22896 **SYNOPSIS**
 22897 `#include <wctype.h>`
 22898 `int iswlower(wint_t wc);`
 22899 CX `int iswlower_l(wint_t wc, locale_t locale);`

22900 **DESCRIPTION**
 22901 CX For `iswlower()`: The functionality described on this reference page is aligned with the ISO C
 22902 standard. Any conflict between the requirements described here and the ISO C standard is
 22903 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22904 CX The `iswlower()` and `iswlower_l()` functions shall test whether `wc` is a wide-character code
 22905 CX representing a character of class **lower** in the current locale of the process, or in the locale
 22906 represented by `locale`, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
 22907 Chapter 7, Locale.

22908 The `wc` argument is a **wint_t**, the value of which the application shall ensure is a wide-character
 22909 code corresponding to a valid character in the current locale, or equal to the value of the macro
 22910 WEOF. If the argument has any other value, the behavior is undefined.

22911 **RETURN VALUE**
 22912 CX The `iswlower()` and `iswlower_l()` functions shall return non-zero if `wc` is a lowercase letter wide-
 22913 character code; otherwise, they shall return 0.

22914 **ERRORS**
 22915 The `iswlower_l()` function may fail if:

22916 CX **[EINVAL]** `locale` is not a valid locale object handle.

22917 **EXAMPLES**
 22918 None.

22919 **APPLICATION USAGE**
 22920 To ensure applications portability, especially across natural languages, only these functions and
 22921 the functions in the reference pages listed in the SEE ALSO section should be used for character
 22922 classification.

22923 **RATIONALE**
 22924 None.

22925 **FUTURE DIRECTIONS**
 22926 None.

22927 **SEE ALSO**
 22928 `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswctype()`, `iswdigit()`, `iswgraph()`, `iswprint()`, `iswpunct()`,
 22929 `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `uselocale()`, the Base Definitions volume of
 22930 IEEE Std 1003.1-200x, Chapter 7, Locale, `<locale.h>`, `<wctype.h>`

22931 **CHANGE HISTORY**
 22932 First released in Issue 4.

22933

Issue 5

22934

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

22935

22936

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the `<wctype.h>` header rather than `<wchar.h>`.

22937

22938

Issue 6

22939

The normative text is updated to avoid use of the term “must” for application requirements.

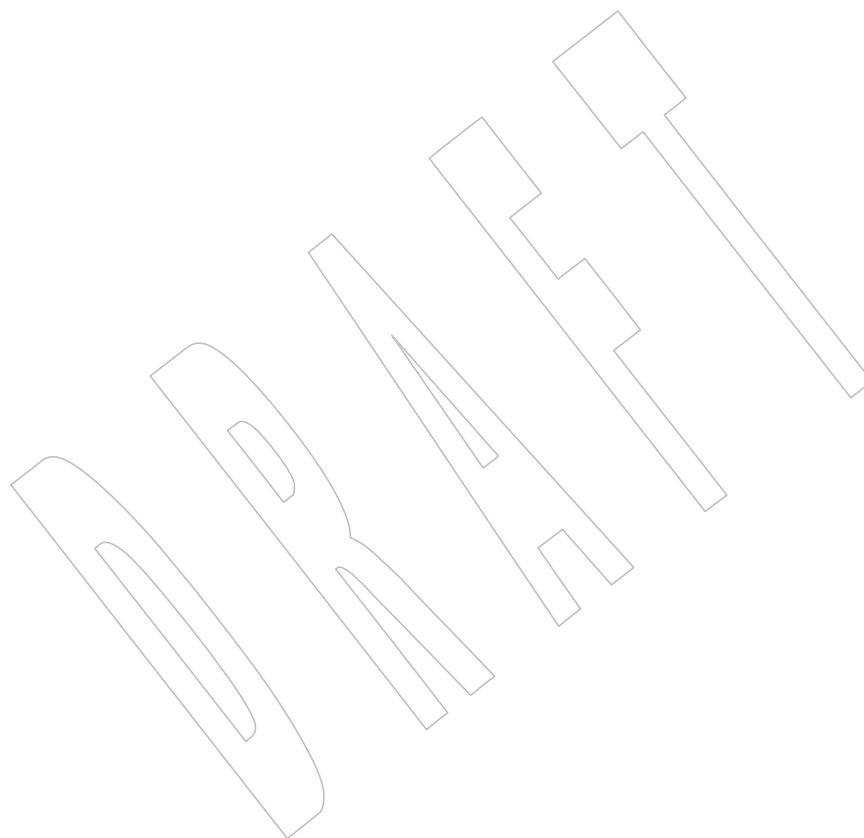
22940

Issue 7

22941

The `iswlower_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

22942



22943 **NAME**
 22944 `iswprint, iswprint_l` — test for a printable wide-character code

22945 **SYNOPSIS**
 22946 `#include <wctype.h>`
 22947 `int iswprint(wint_t wc);`
 22948 CX `int iswprint_l(wint_t wc, locale_t locale);`

22949 **DESCRIPTION**
 22950 CX For `iswprint()`: The functionality described on this reference page is aligned with the ISO C
 22951 standard. Any conflict between the requirements described here and the ISO C standard is
 22952 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22953 CX The `iswprint()` and `iswprint_l()` functions shall test whether `wc` is a wide-character code
 22954 CX representing a character of class **print** in the current locale of the process, or in the locale
 22955 represented by `locale`, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
 22956 Chapter 7, Locale.

22957 The `wc` argument is a **wint_t**, the value of which the application shall ensure is a wide-character
 22958 code corresponding to a valid character in the current locale, or equal to the value of the macro
 22959 WEOF. If the argument has any other value, the behavior is undefined.

22960 **RETURN VALUE**
 22961 CX The `iswprint()` and `iswprint_l()` functions shall return non-zero if `wc` is a printable wide-
 22962 character code; otherwise, they shall return 0.

22963 **ERRORS**
 22964 The `iswprint_l()` function may fail if:

22965 CX **[EINVAL]** `locale` is not a valid locale object handle.

22966 **EXAMPLES**
 22967 None.

22968 **APPLICATION USAGE**
 22969 To ensure applications portability, especially across natural languages, only these functions and
 22970 the functions in the reference pages listed in the SEE ALSO section should be used for character
 22971 classification.

22972 **RATIONALE**
 22973 None.

22974 **FUTURE DIRECTIONS**
 22975 None.

22976 **SEE ALSO**
 22977 `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswctype()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswpunct()`,
 22978 `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `uselocale()`, the Base Definitions volume of
 22979 IEEE Std 1003.1-200x, Chapter 7, Locale, `<locale.h>`, `<wctype.h>`

22980 **CHANGE HISTORY**
 22981 First released in Issue 4.

22982

Issue 5

22983

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

22984

22985

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the `<wctype.h>` header rather than `<wchar.h>`.

22986

22987

Issue 6

22988

The normative text is updated to avoid use of the term “must” for application requirements.

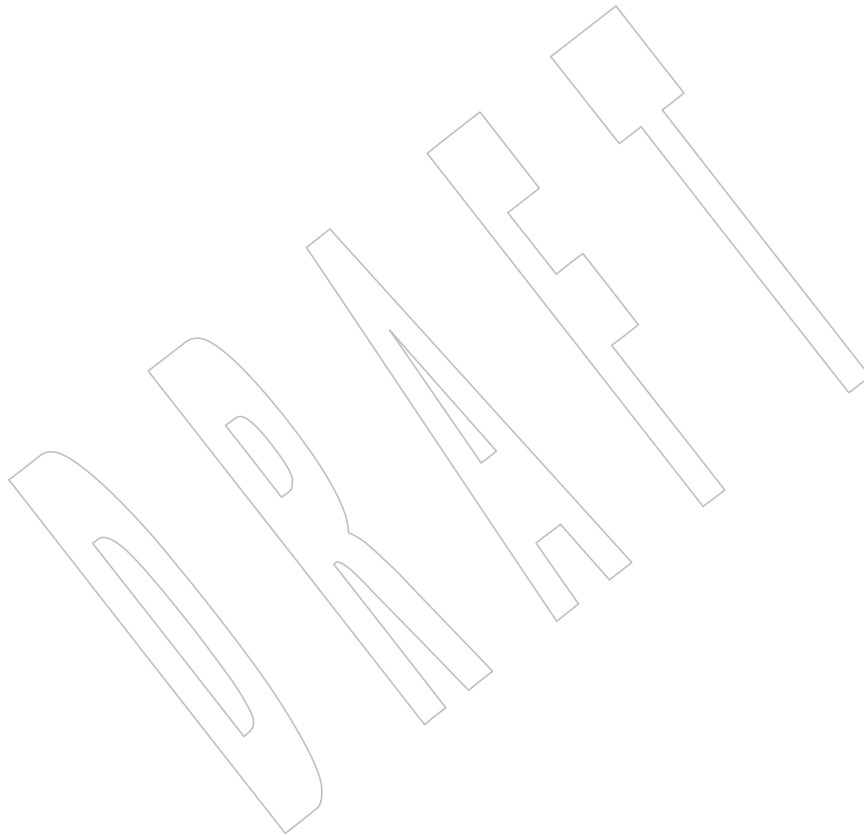
22989

Issue 7

22990

The `iswprint_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

22991



22992 **NAME**22993 `iswpunct`, `iswpunct_l` — test for a punctuation wide-character code22994 **SYNOPSIS**22995 `#include <wctype.h>`22996 `int iswpunct(wint_t wc);`22997 CX `int iswpunct_l(wint_t wc, locale_t locale);`22998 **DESCRIPTION**22999 CX For `iswpunct()`: The functionality described on this reference page is aligned with the ISO C
23000 standard. Any conflict between the requirements described here and the ISO C standard is
23001 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.23002 CX The `iswpunct()` and `iswpunct_l()` functions shall test whether `wc` is a wide-character code
23003 CX representing a character of class **punct** in the current locale of the process, or in the locale
23004 represented by `locale`, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
23005 Chapter 7, Locale.23006 The `wc` argument is a **wint_t**, the value of which the application shall ensure is a wide-character
23007 code corresponding to a valid character in the current locale, or equal to the value of the macro
23008 WEOF. If the argument has any other value, the behavior is undefined.23009 **RETURN VALUE**23010 CX The `iswpunct()` and `iswpunct_l()` functions shall return non-zero if `wc` is a punctuation wide-
23011 character code; otherwise, they shall return 0.23012 **ERRORS**23013 The `iswpunct_l()` function may fail if:23014 CX **[EINVAL]** `locale` is not a valid locale object handle.23015 **EXAMPLES**

23016 None.

23017 **APPLICATION USAGE**23018 To ensure applications portability, especially across natural languages, only these functions and
23019 the functions in the reference pages listed in the SEE ALSO section should be used for character
23020 classification.23021 **RATIONALE**

23022 None.

23023 **FUTURE DIRECTIONS**

23024 None.

23025 **SEE ALSO**23026 `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswctype()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`,
23027 `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `uselocale()`, the Base Definitions volume of
23028 IEEE Std 1003.1-200x, Chapter 7, Locale, `<locale.h>`, `<wctype.h>`23029 **CHANGE HISTORY**

23030 First released in Issue 4.

23031

Issue 5

23032

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

23033

23034

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the `<wctype.h>` header rather than `<wchar.h>`.

23035

23036

Issue 6

23037

The normative text is updated to avoid use of the term “must” for application requirements.

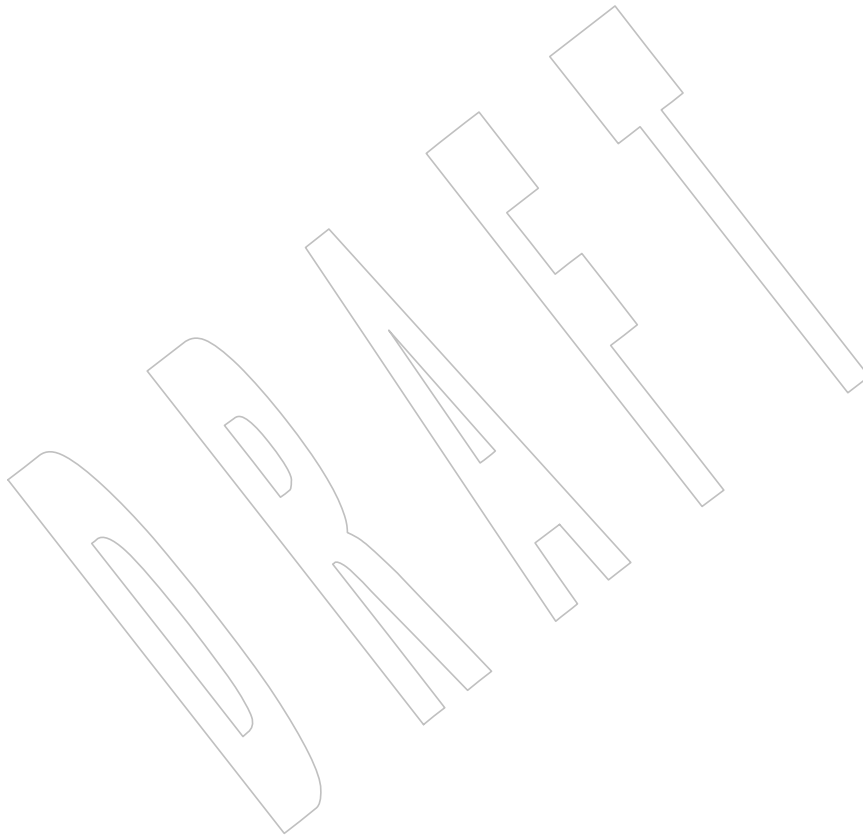
23038

Issue 7

23039

The `iswpunct_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

23040



23041 **NAME**23042 `iswspace`, `iswspace_l` — test for a white-space wide-character code23043 **SYNOPSIS**23044 `#include <wctype.h>`23045 `int iswspace(wint_t wc);`23046 CX `int iswspace_l(wint_t wc, locale_t locale);`23047 **DESCRIPTION**23048 CX For `iswspace()`: The functionality described on this reference page is aligned with the ISO C
23049 standard. Any conflict between the requirements described here and the ISO C standard is
23050 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.23051 CX The `iswspace()` and `iswspace_l()` functions shall test whether `wc` is a wide-character code
23052 representing a character of class **space** in the current locale of the process, or in the locale
23053 represented by `locale`, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
23054 Chapter 7, Locale.23055 The `wc` argument is a **wint_t**, the value of which the application shall ensure is a wide-character
23056 code corresponding to a valid character in the current locale, or equal to the value of the macro
23057 WEOF. If the argument has any other value, the behavior is undefined.23058 **RETURN VALUE**23059 CX The `iswspace()` and `iswspace_l()` functions shall return non-zero if `wc` is a white-space wide-
23060 character code; otherwise, they shall return 0.23061 **ERRORS**23062 The `iswspace_l()` function may fail if:23063 CX **[EINVAL]** `locale` is not a valid locale object handle.23064 **EXAMPLES**

23065 None.

23066 **APPLICATION USAGE**23067 To ensure applications portability, especially across natural languages, only these functions and
23068 the functions in the reference pages listed in the SEE ALSO section should be used for character
23069 classification.23070 **RATIONALE**

23071 None.

23072 **FUTURE DIRECTIONS**

23073 None.

23074 **SEE ALSO**23075 `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswctype()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`,
23076 `iswpunct()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `uselocale()`, the Base Definitions volume of
23077 IEEE Std 1003.1-200x, Chapter 7, Locale, `<locale.h>`, `<wctype.h>`23078 **CHANGE HISTORY**

23079 First released in Issue 4.

23080

Issue 5

23081

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

23082

23083

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the `<wctype.h>` header rather than `<wchar.h>`.

23084

23085

Issue 6

23086

The normative text is updated to avoid use of the term “must” for application requirements.

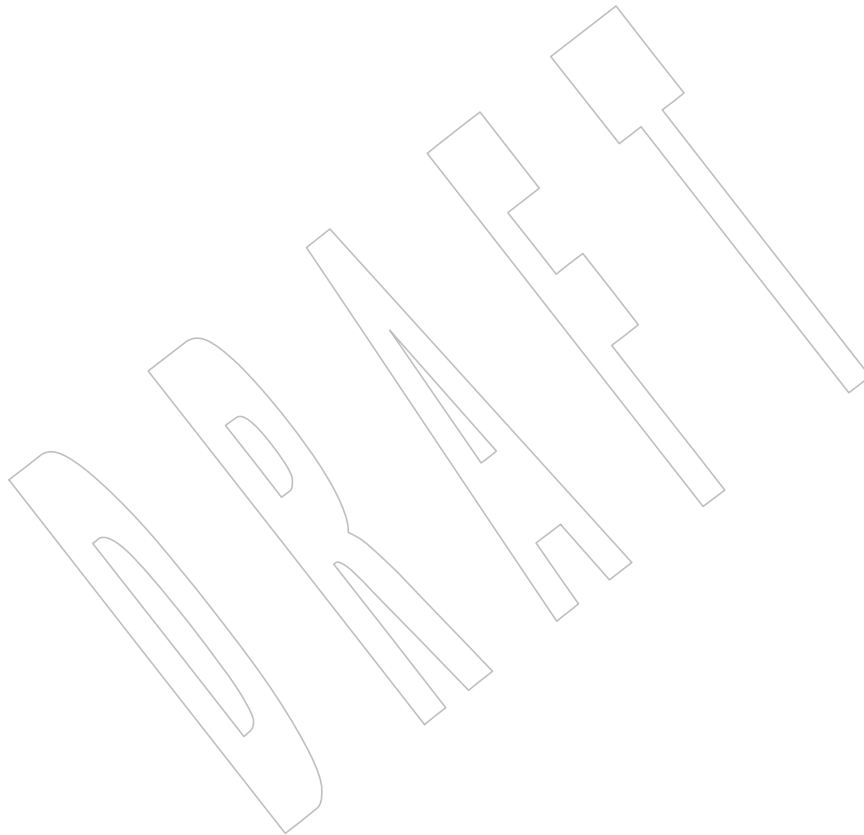
23087

Issue 7

23088

The `iswspace_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

23089



23090 **NAME**
 23091 `iswupper, iswupper_l` — test for an uppercase letter wide-character code

23092 **SYNOPSIS**
 23093 `#include <wctype.h>`
 23094 `int iswupper(wint_t wc);`
 23095 CX `int iswupper_l(wint_t wc, locale_t locale);`

23096 **DESCRIPTION**
 23097 CX For `iswupper()`: The functionality described on this reference page is aligned with the ISO C
 23098 standard. Any conflict between the requirements described here and the ISO C standard is
 23099 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23100 CX The `iswupper()` and `iswupper_l()` functions shall test whether `wc` is a wide-character code
 23101 CX representing a character of class **upper** in the current locale of the process, or in the locale
 23102 represented by `locale`, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
 23103 Chapter 7, Locale.

23104 The `wc` argument is a **wint_t**, the value of which the application shall ensure is a wide-character
 23105 code corresponding to a valid character in the current locale, or equal to the value of the macro
 23106 WEOF. If the argument has any other value, the behavior is undefined.

23107 **RETURN VALUE**
 23108 CX The `iswupper()` and `iswupper_l()` functions shall return non-zero if `wc` is an uppercase letter
 23109 wide-character code; otherwise, they shall return 0.

23110 **ERRORS**
 23111 The `iswupper_l()` function may fail if:

23112 CX **[EINVAL]** `locale` is not a valid locale object handle.

23113 **EXAMPLES**
 23114 None.

23115 **APPLICATION USAGE**
 23116 To ensure applications portability, especially across natural languages, only these functions and
 23117 the functions in the reference pages listed in the SEE ALSO section should be used for character
 23118 classification.

23119 **RATIONALE**
 23120 None.

23121 **FUTURE DIRECTIONS**
 23122 None.

23123 **SEE ALSO**
 23124 `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswctype()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`,
 23125 `iswpunct()`, `iswspace()`, `iswxdigit()`, `setlocale()`, `uselocale()`, the Base Definitions volume of
 23126 IEEE Std 1003.1-200x, Chapter 7, Locale, `<locale.h>`, `<wctype.h>`

23127 **CHANGE HISTORY**
 23128 First released in Issue 4.

23129

Issue 5

23130

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

23131

23132

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the `<wctype.h>` header rather than `<wchar.h>`.

23133

23134

Issue 6

23135

The normative text is updated to avoid use of the term “must” for application requirements.

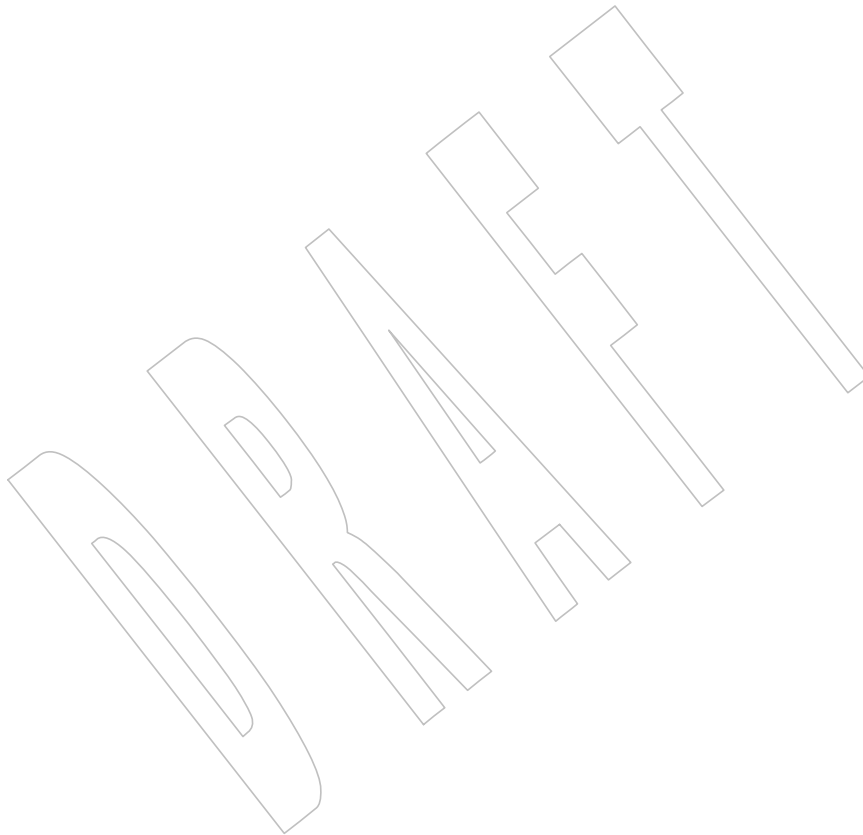
23136

Issue 7

23137

The `iswupper_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

23138



23139 **NAME**

23140 iswxdigit, iswxdigit_l — test for a hexadecimal digit wide-character code

23141 **SYNOPSIS**

23142 #include <wctype.h>

23143 int iswxdigit(wint_t wc);

23144 CX int iswxdigit_l(wint_t wc, locale_t locale);

23145 **DESCRIPTION**23146 CX For *iswxdigit()*: The functionality described on this reference page is aligned with the ISO C
23147 standard. Any conflict between the requirements described here and the ISO C standard is
23148 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.23149 CX The *iswxdigit()* and *iswxdigit_l()* functions shall test whether *wc* is a wide-character code
23150 representing a character of class **xdigit** in the current locale of the process, or in the locale
23151 represented by *locale*, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
23152 Chapter 7, Locale.23153 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
23154 code corresponding to a valid character in the current locale, or equal to the value of the macro
23155 WEOF. If the argument has any other value, the behavior is undefined.23156 **RETURN VALUE**23157 CX The *iswxdigit()* and *iswxdigit_l()* functions shall return non-zero if *wc* is a hexadecimal digit
23158 wide-character code; otherwise, they shall return 0.23159 **ERRORS**23160 The *iswxdigit_l()* function may fail if:23161 CX [EINVAL] *locale* is not a valid locale object handle.23162 **EXAMPLES**

23163 None.

23164 **APPLICATION USAGE**23165 To ensure applications portability, especially across natural languages, only these functions and
23166 the functions in the reference pages listed in the SEE ALSO section should be used for character
23167 classification.23168 **RATIONALE**

23169 None.

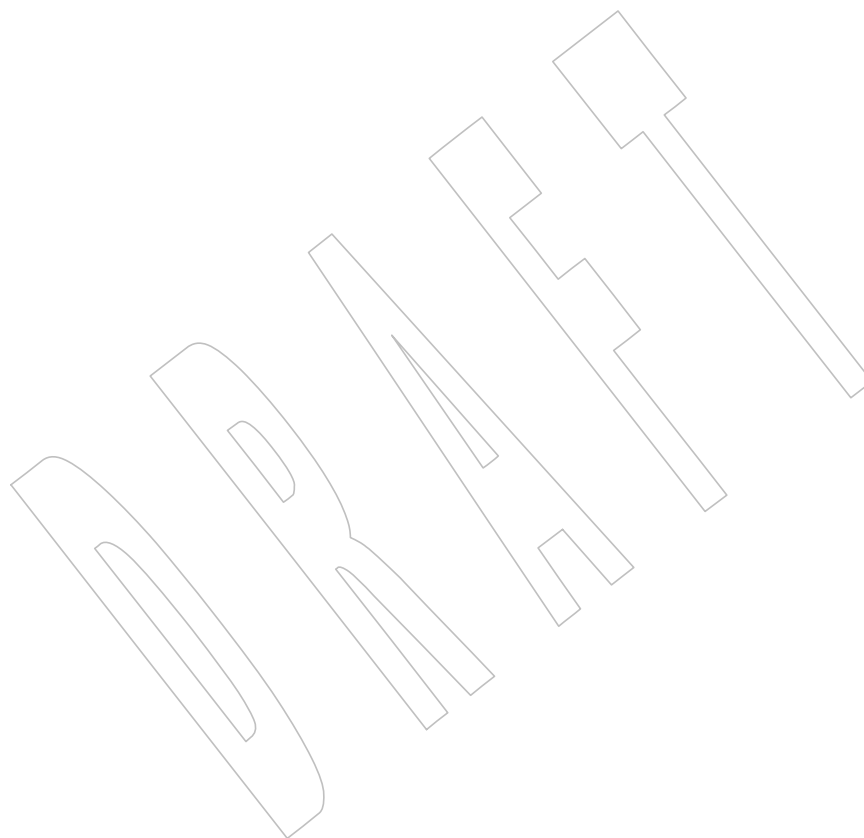
23170 **FUTURE DIRECTIONS**

23171 None.

23172 **SEE ALSO**23173 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*,
23174 *iswpunct()*, *iswspace()*, *iswupper()*, *setlocale()*, *uselocale()*, the Base Definitions volume of
23175 IEEE Std 1003.1-200x, Chapter 7, Locale, <locale.h>, <wctype.h>23176 **CHANGE HISTORY**

23177 First released in Issue 4.

- 23178 **Issue 5**
23179 The following change has been made in this issue for alignment with
23180 ISO/IEC 9899:1990/Amendment 1:1995 (E):
- 23181 • The SYNOPSIS has been changed to indicate that this function and associated data types
23182 are now made visible by inclusion of the `<wctype.h>` header rather than `<wchar.h>`.
- 23183 **Issue 6**
23184 The normative text is updated to avoid use of the term “must” for application requirements.
- 23185 **Issue 7**
23186 The `iswxdigit_l()` function is added from The Open Group Technical Standard, 2006, Extended
23187 API Set Part 4.



23188 **NAME**
 23189 isxdigit, isxdigit_l — test for a hexadecimal digit

23190 **SYNOPSIS**
 23191 #include <ctype.h>
 23192 int isxdigit(int c);
 23193 CX int isxdigit_l(int c, locale_t locale);

23194 **DESCRIPTION**
 23195 CX For *isxdigit()*: The functionality described on this reference page is aligned with the ISO C
 23196 standard. Any conflict between the requirements described here and the ISO C standard is
 23197 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23198 CX The *isxdigit()* and *isxdigit_l()* functions shall test whether *c* is a character of class **xdigit** in the
 23199 current locale of the process, or in the locale represented by *locale*, respectively; see the Base
 23200 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

23201 The *c* argument is an **int**, the value of which the application shall ensure is a character
 23202 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
 23203 any other value, the behavior is undefined.

23204 **RETURN VALUE**
 23205 CX The *isxdigit()* and *isxdigit_l()* functions shall return non-zero if *c* is a hexadecimal digit;
 23206 otherwise, they shall return 0.

23207 **ERRORS**
 23208 The *isxdigit_l()* function may fail if:

23209 CX [EINVAL] *locale* is not a valid locale object handle.

23210 **EXAMPLES**
 23211 None.

23212 **APPLICATION USAGE**
 23213 To ensure applications portability, especially across natural languages, only these functions and
 23214 the functions in the reference pages listed in the SEE ALSO section should be used for character
 23215 classification.

23216 **RATIONALE**
 23217 None.

23218 **FUTURE DIRECTIONS**
 23219 None.

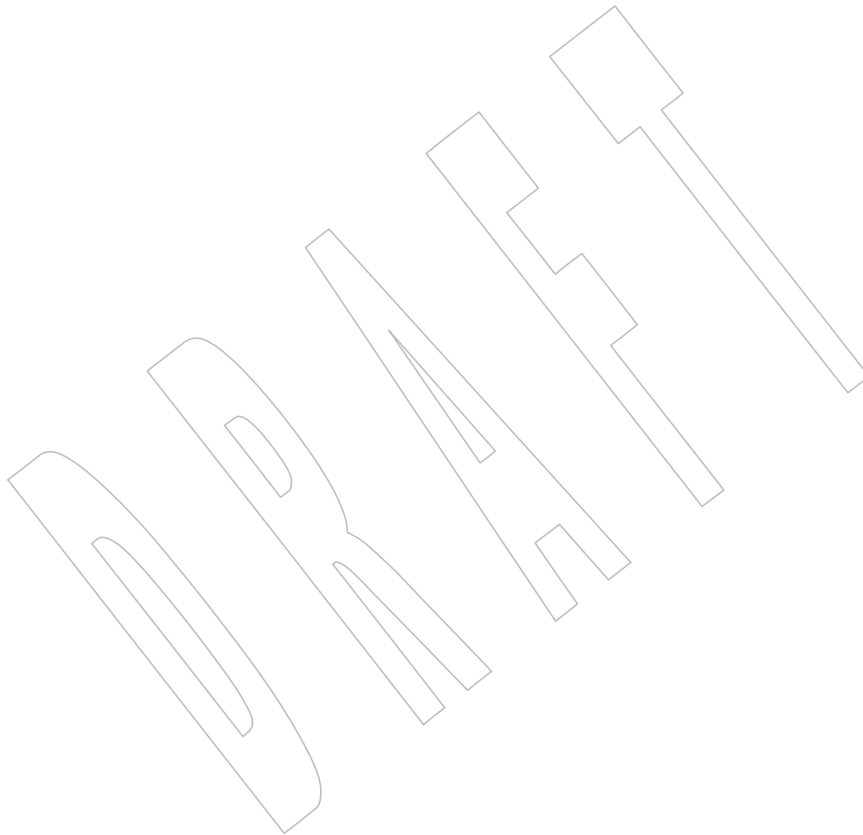
23220 **SEE ALSO**
 23221 *isalnum()*, *isalpha()*, *isblank()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*,
 23222 *isupper()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale, <ctype.h>

23223 **CHANGE HISTORY**
 23224 First released in Issue 1. Derived from Issue 1 of the SVID.

23225 **Issue 6**
 23226 The normative text is updated to avoid use of the term “must” for application requirements.

23227
23228
23229**Issue 7**

The *isxdigit_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



23230 **NAME**23231 `j0, j1, jn` — Bessel functions of the first kind23232 **SYNOPSIS**

```

23233 XSI      #include <math.h>
23234
23234 double j0(double x);
23235 double j1(double x);
23236 double jn(int n, double x);

```

23237 **DESCRIPTION**

23238 The `j0()`, `j1()`, and `jn()` functions shall compute Bessel functions of x of the first kind of orders 0,
 23239 1, and n , respectively.

23240 An application wishing to check for error situations should set `errno` to zero and call
 23241 `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `errno` is non-zero or
 23242 `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-
 23243 zero, an error has occurred.

23244 **RETURN VALUE**

23245 Upon successful completion, these functions shall return the relevant Bessel value of x of the
 23246 first kind.

23247 If the x argument is too large in magnitude, or the correct result would cause underflow, 0 shall
 23248 be returned and a range error may occur.

23249 If x is NaN, a NaN shall be returned.

23250 **ERRORS**

23251 These functions may fail if:

23252 **Range Error** The value of x was too large in magnitude, or an underflow occurred.

23253 If the integer expression `(math_errhandling & MATH_ERRNO)` is non-zero,
 23254 then `errno` shall be set to [ERANGE]. If the integer expression
 23255 `(math_errhandling & MATH_ERREXCEPT)` is non-zero, then the underflow
 23256 floating-point exception shall be raised.

23257 No other errors shall occur.

23258 **EXAMPLES**

23259 None.

23260 **APPLICATION USAGE**

23261 On error, the expressions `(math_errhandling & MATH_ERRNO)` and `(math_errhandling &`
 23262 `MATH_ERREXCEPT)` are independent of each other, but at least one of them must be non-zero.

23263 **RATIONALE**

23264 None.

23265 **FUTURE DIRECTIONS**

23266 None.

23267 **SEE ALSO**

23268 `feclearexcept()`, `fetestexcept()`, `isnan()`, `y0()`, the Base Definitions volume of IEEE Std 1003.1-200x,
 23269 Section 4.18, Treatment of Error Conditions for Mathematical Functions, `<math.h>`

23270

CHANGE HISTORY

23271

First released in Issue 1. Derived from Issue 1 of the SVID.

23272

Issue 5

23273

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

23274

23275

Issue 6

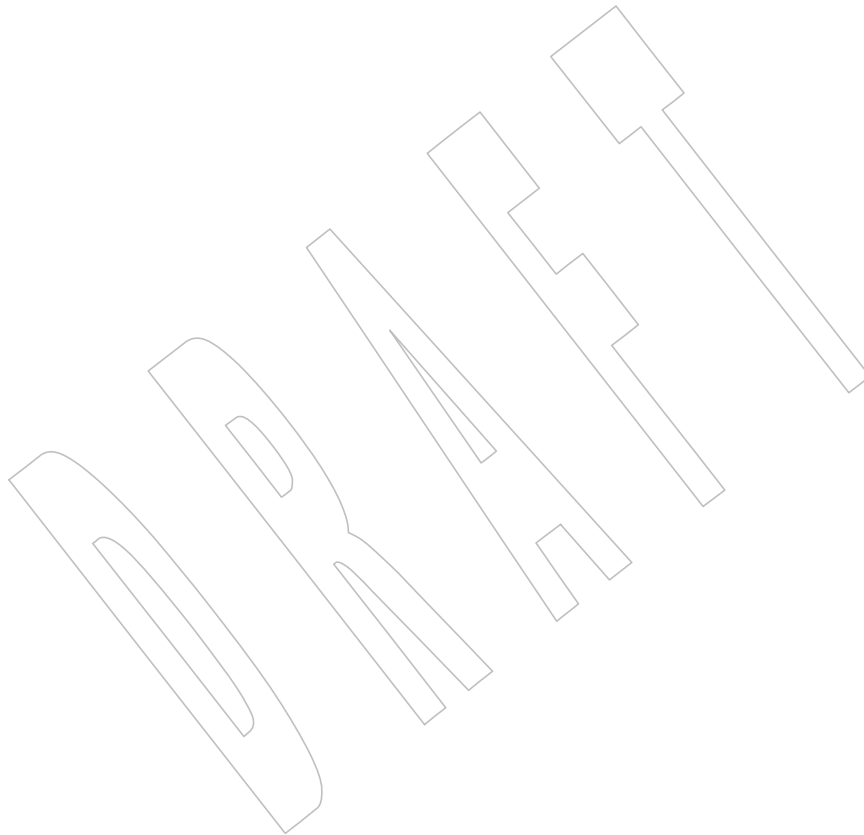
23276

The may fail [EDOM] error is removed for the case for NaN.

23277

The RETURN VALUE and ERRORS sections are reworked for alignment of the error handling with the ISO/IEC 9899:1999 standard.

23278



jrand48()

23279 **NAME**
23280 jrand48 — generate a uniformly distributed pseudo-random long signed integer

SYNOPSIS

23281 XSI #include <stdlib.h>
23282
23283 long jrand48(unsigned short xsubi[3]);

DESCRIPTION

23284 Refer to *drand48()*.
23285

23286 **NAME**
 23287 kill — send a signal to a process or a group of processes

23288 **SYNOPSIS**

```
23289 CX #include <signal.h>
23290 int kill(pid_t pid, int sig);
```

23291 **DESCRIPTION**

23292 The *kill()* function shall send a signal to a process or a group of processes specified by *pid*. The
 23293 signal to be sent is specified by *sig* and is either one from the list given in **<signal.h>** or 0. If *sig* is
 23294 0 (the null signal), error checking is performed but no signal is actually sent. The null signal can
 23295 be used to check the validity of *pid*.

23296 For a process to have permission to send a signal to a process designated by *pid*, unless the
 23297 sending process has appropriate privileges, the real or effective user ID of the sending process
 23298 shall match the real or saved set-user-ID of the receiving process.

23299 If *pid* is greater than 0, *sig* shall be sent to the process whose process ID is equal to *pid*.

23300 If *pid* is 0, *sig* shall be sent to all processes (excluding an unspecified set of system processes)
 23301 whose process group ID is equal to the process group ID of the sender, and for which the process
 23302 has permission to send a signal.

23303 If *pid* is -1 , *sig* shall be sent to all processes (excluding an unspecified set of system processes) for
 23304 which the process has permission to send that signal.

23305 If *pid* is negative, but not -1 , *sig* shall be sent to all processes (excluding an unspecified set of
 23306 system processes) whose process group ID is equal to the absolute value of *pid*, and for which
 23307 the process has permission to send a signal.

23308 If the value of *pid* causes *sig* to be generated for the sending process, and if *sig* is not blocked for
 23309 the calling thread and if no other thread has *sig* unblocked or is waiting in a *sigwait()* function
 23310 for *sig*, either *sig* or at least one pending unblocked signal shall be delivered to the sending
 23311 thread before *kill()* returns.

23312 The user ID tests described above shall not be applied when sending SIGCONT to a process that
 23313 is a member of the same session as the sending process.

23314 An implementation that provides extended security controls may impose further
 23315 implementation-defined restrictions on the sending of signals, including the null signal. In
 23316 particular, the system may deny the existence of some or all of the processes specified by *pid*.

23317 The *kill()* function is successful if the process has permission to send *sig* to any of the processes
 23318 specified by *pid*. If *kill()* fails, no signal shall be sent.

23319 **RETURN VALUE**

23320 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
 23321 indicate the error.

23322 **ERRORS**

23323 The *kill()* function shall fail if:

23324	[EINVAL]	The value of the <i>sig</i> argument is an invalid or unsupported signal number.
23325	[EPERM]	The process does not have permission to send the signal to any receiving process.
23326		

kill()

[ESRCH] No process or process group can be found corresponding to that specified by *pid*.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The semantics for permission checking for *kill()* differed between System V and most other implementations, such as Version 7 or 4.3 BSD. The semantics chosen for this volume of IEEE Std 1003.1-200x agree with System V. Specifically, a set-user-ID process cannot protect itself against signals (or at least not against SIGKILL) unless it changes its real user ID. This choice allows the user who starts an application to send it signals even if it changes its effective user ID. The other semantics give more power to an application that wants to protect itself from the user who ran it.

Some implementations provide semantic extensions to the *kill()* function when the absolute value of *pid* is greater than some maximum, or otherwise special, value. Negative values are a flag to *kill()*. Since most implementations return [ESRCH] in this case, this behavior is not included in this volume of IEEE Std 1003.1-200x, although a conforming implementation could provide such an extension.

The unspecified processes to which a signal cannot be sent may include the scheduler or *init*.

There was initially strong sentiment to specify that, if *pid* specifies that a signal be sent to the calling process and that signal is not blocked, that signal would be delivered before *kill()* returns. This would permit a process to call *kill()* and be guaranteed that the call never return. However, historical implementations that provide only the *signal()* function make only the weaker guarantee in this volume of IEEE Std 1003.1-200x, because they only deliver one signal each time a process enters the kernel. Modifications to such implementations to support the *sigaction()* function generally require entry to the kernel following return from a signal-catching function, in order to restore the signal mask. Such modifications have the effect of satisfying the stronger requirement, at least when *sigaction()* is used, but not necessarily when *signal()* is used. The developers of this volume of IEEE Std 1003.1-200x considered making the stronger requirement except when *signal()* is used, but felt this would be unnecessarily complex. Implementors are encouraged to meet the stronger requirement whenever possible. In practice, the weaker requirement is the same, except in the rare case when two signals arrive during a very short window. This reasoning also applies to a similar requirement for *sigprocmask()*.

In 4.2 BSD, the SIGCONT signal can be sent to any descendant process regardless of user-ID security checks. This allows a job control shell to continue a job even if processes in the job have altered their user IDs (as in the *su* command). In keeping with the addition of the concept of sessions, similar functionality is provided by allowing the SIGCONT signal to be sent to any process in the same session regardless of user ID security checks. This is less restrictive than BSD in the sense that ancestor processes (in the same session) can now be the recipient. It is more restrictive than BSD in the sense that descendant processes that form new sessions are now subject to the user ID checks. A similar relaxation of security is not necessary for the other job control signals since those signals are typically sent by the terminal driver in recognition of special characters being typed; the terminal driver bypasses all security checks.

In secure implementations, a process may be restricted from sending a signal to a process having a different security label. In order to prevent the existence of the signal to a process having a different security label.

23377 call (subject to permission checking), while others give an error of [ESRCH]. Since the definition
 23378 of process lifetime in this volume of IEEE Std 1003.1-200x covers inactive processes, the [ESRCH]
 23379 error as described is inappropriate in this case. In particular, this means that an application
 23380 cannot have a parent process check for termination of a particular child with *kill()*. (Usually this
 23381 is done with the null signal; this can be done reliably with *waitpid()*.)

23382 There is some belief that the name *kill()* is misleading, since the function is not always intended
 23383 to cause process termination. However, the name is common to all historical implementations,
 23384 and any change would be in conflict with the goal of minimal changes to existing application
 23385 code.

23386 FUTURE DIRECTIONS

23387 None.

23388 SEE ALSO

23389 *getpid()*, *raise()*, *setsid()*, *sigaction()*, *sigqueue()*, the Base Definitions volume of
 23390 IEEE Std 1003.1-200x, **<signal.h>**, **<sys/types.h>**

23391 CHANGE HISTORY

23392 First released in Issue 1. Derived from Issue 1 of the SVID.

23393 Issue 5

23394 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

23395 Issue 6

23396 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

23397 The following new requirements on POSIX implementations derive from alignment with the
 23398 Single UNIX Specification:

- 23399 • In the DESCRIPTION, the second paragraph is reworded to indicate that the saved set-
 23400 user-ID of the calling process is checked in place of its effective user ID. This is a FIPS
 23401 requirement.
- 23402 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
 23403 required for conforming implementations of previous POSIX specifications, it was not
 23404 required for UNIX applications.
- 23405 • The behavior when *pid* is -1 is now specified. It was previously explicitly unspecified in
 23406 the POSIX.1-1988 standard.

23407 The normative text is updated to avoid use of the term “must” for application requirements.

23408 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/51 is applied, correcting the RATIONALE
 23409 section.

23410 **NAME**
 23411 `killpg` — send a signal to a process group

23412 **SYNOPSIS**

23413 XSI `#include <signal.h>`
 23414 `int killpg(pid_t pgrp, int sig);`

23415 **DESCRIPTION**

23416 The `killpg()` function shall send the signal specified by `sig` to the process group specified by `pgrp`.
 23417 If `pgrp` is greater than 1, `killpg(pgrp, sig)` shall be equivalent to `kill(-pgrp, sig)`. If `pgrp` is less than or
 23418 equal to 1, the behavior of `killpg()` is undefined.

23419 **RETURN VALUE**

23420 Refer to `kill()`.

23421 **ERRORS**

23422 Refer to `kill()`.

23423 **EXAMPLES**

23424 **Sending a Signal to All Other Members of a Process Group**

23425 The following example shows how the calling process could send a signal to all other members
 23426 of its process group. To prevent itself from receiving the signal it first makes itself immune to the
 23427 signal by ignoring it.

```
23428 #include <signal.h>
23429 #include <unistd.h>
23430 ...
23431     if (signal(SIGUSR1, SIG_IGN) == SIG_ERR)
23432         /* Handle error */;
23433
23434     if (killpg(getpgrp(), SIGUSR1) == -1)
23435         /* Handle error */;
```

23435 **APPLICATION USAGE**

23436 None.

23437 **RATIONALE**

23438 None.

23439 **FUTURE DIRECTIONS**

23440 None.

23441 **SEE ALSO**

23442 `getpgid()`, `getpid()`, `kill()`, `raise()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<signal.h>`

23443 **CHANGE HISTORY**

23444 First released in Issue 4, Version 2.

23445 **Issue 5**

23446 Moved from X/OPEN UNIX extension to BASE.

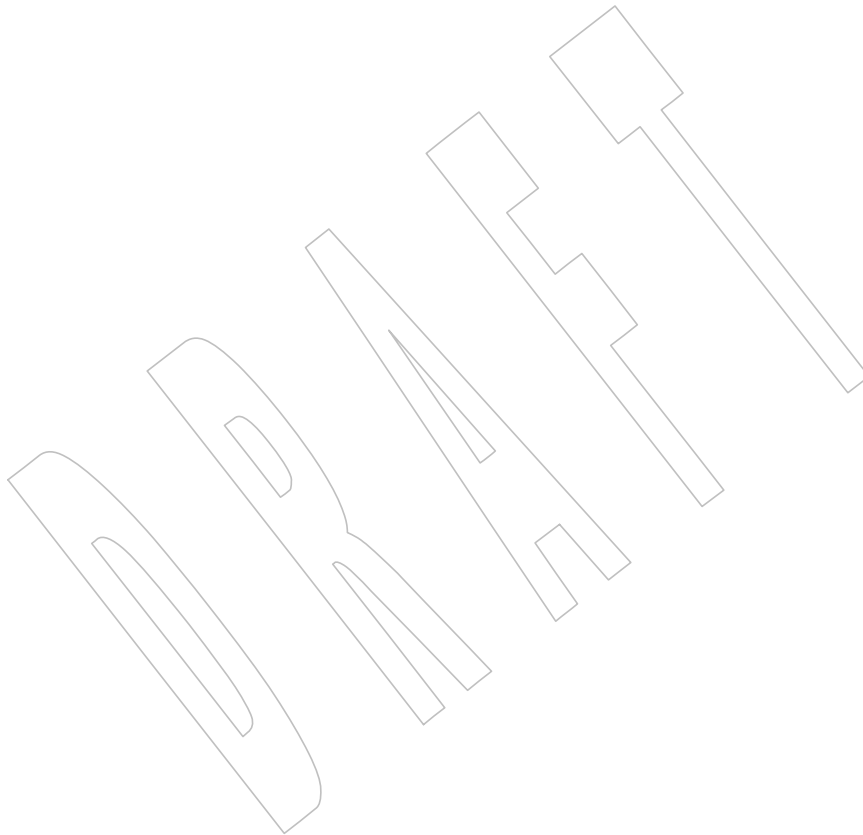
23447

Issue 6

23448

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/52 is applied, adding the example to the EXAMPLES section.

23449



l64a()

23450 **NAME**
23451 l64a — convert a 32-bit integer to a radix-64 ASCII string

SYNOPSIS

23452 XSI #include <stdlib.h>
23453
23454 char *l64a(long value);

DESCRIPTION

23455 Refer to [a64l\(\)](#).
23456

23457 **NAME**
 23458 labs, llabs — return a long integer absolute value

23459 **SYNOPSIS**
 23460 #include <stdlib.h>
 23461 long labs(long *i*);
 23462 long long llabs(long long *i*);

23463 **DESCRIPTION**
 23464 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 23465 conflict between the requirements described here and the ISO C standard is unintentional. This
 23466 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23467 The *labs()* function shall compute the absolute value of the **long** integer operand *i*. The *llabs()*
 23468 function shall compute the absolute value of the **long long** integer operand *i*. If the result
 23469 cannot be represented, the behavior is undefined.

23470 **RETURN VALUE**
 23471 The *labs()* function shall return the absolute value of the **long** integer operand. The *labs()*
 23472 function shall return the absolute value of the **long long** integer operand.

23473 **ERRORS**
 23474 No errors are defined.

23475 **EXAMPLES**
 23476 None.

23477 **APPLICATION USAGE**
 23478 None.

23479 **RATIONALE**
 23480 None.

23481 **FUTURE DIRECTIONS**
 23482 None.

23483 **SEE ALSO**
 23484 *abs()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdlib.h>

23485 **CHANGE HISTORY**
 23486 First released in Issue 4. Derived from the ISO C standard.

23487 **Issue 6**
 23488 The *llabs()* function is added for alignment with the ISO/IEC 9899:1999 standard.

23489 **NAME**

23490 lchown — change the owner and group of a symbolic link

23491 **SYNOPSIS**

23492 #include <unistd.h>

23493 int lchown(const char *path, uid_t owner, gid_t group);

23494 **DESCRIPTION**

23495 The *lchown()* function shall be equivalent to *chown()*, except in the case where the named file is a
 23496 symbolic link. In this case, *lchown()* shall change the ownership of the symbolic link file itself,
 23497 while *chown()* changes the ownership of the file or directory to which the symbolic link refers.

23498 **RETURN VALUE**

23499 Upon successful completion, *lchown()* shall return 0. Otherwise, it shall return -1 and set *errno* to
 23500 indicate an error.

23501 **ERRORS**23502 The *lchown()* function shall fail if:

- 23503 [EACCES] Search permission is denied on a component of the path prefix of *path*.
- 23504 [EINVAL] The owner or group ID is not a value supported by the implementation.
- 23505 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 23506 argument.
- 23507 [ENAMETOOLONG]
 23508 The length of a pathname exceeds {PATH_MAX} or a pathname component is
 23509 longer than {NAME_MAX}.
- 23510 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.
- 23511 [ENOTDIR] A component of the path prefix of *path* is not a directory.
- 23512 [EOPNOTSUPP] The *path* argument names a symbolic link and the implementation does not
 23513 support setting the owner or group of a symbolic link.
- 23514 [EPERM] The effective user ID does not match the owner of the file and the process does
 23515 not have appropriate privileges.
- 23516 [EROFS] The file resides on a read-only file system.
- 23517 The *lchown()* function may fail if:
- 23518 [EIO] An I/O error occurred while reading or writing to the file system.
- 23519 [EINTR] A signal was caught during execution of the function.
- 23520 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 23521 resolution of the *path* argument.
- 23522 [ENAMETOOLONG]
 23523 Pathname resolution of a symbolic link produced an intermediate result
 23524 whose length exceeds {PATH_MAX}.

EXAMPLES**Changing the Current Owner of a File**

The following example shows how to change the ownership of the symbolic link named `/modules/pass1` to the user ID associated with "jones" and the group ID associated with "cnd".

The numeric value for the user ID is obtained by using the `getpwnam()` function. The numeric value for the group ID is obtained by using the `getgrnam()` function.

```
#include <sys/types.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>

struct passwd *pwd;
struct group *grp;
char          *path = "/modules/pass1";
...
pwd = getpwnam("jones");
grp = getgrnam("cnd");
lchown(path, pwd->pw_uid, grp->gr_gid);
```

APPLICATION USAGE

On implementations which support symbolic links as directory entries rather than files, `lchown()` may fail.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[*chown\(\)*](#), [*symlink\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<unistd.h>`

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

The Open Group Base Resolution bwg2001-013 is applied, adding wording to the APPLICATION USAGE.

Issue 7

The `lchown()` function is moved from the XSI option to the Base.

lcong48()

23562

NAME

23563

lcong48 — seed a uniformly distributed pseudo-random signed long integer generator

23564

SYNOPSIS

23565

XSI `#include <stdlib.h>`

23566

`void lcong48(unsigned short param[7]);`

23567

DESCRIPTION

23568

Refer to *drand48()*.

23569 **NAME**

23570 ldexp, ldexpf, ldexpl — load exponent of a floating-point number

23571 **SYNOPSIS**

23572 #include <math.h>

23573 double ldexp(double x, int exp);

23574 float ldexpf(float x, int exp);

23575 long double ldexpl(long double x, int exp);

23576 **DESCRIPTION**23577 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23578 conflict between the requirements described here and the ISO C standard is unintentional. This
23579 volume of IEEE Std 1003.1-200x defers to the ISO C standard.23580 These functions shall compute the quantity $x * 2^{exp}$.23581 An application wishing to check for error situations should set *errno* to zero and call
23582 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
23583 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
23584 zero, an error has occurred.23585 **RETURN VALUE**23586 Upon successful completion, these functions shall return *x* multiplied by 2, raised to the power
23587 *exp*.23588 If these functions would cause overflow, a range error shall occur and *ldexp()*, *ldexpf()*, and
23589 *ldexpl()* shall return \pm HUGE_VAL, \pm HUGE_VALF, and \pm HUGE_VALL (according to the sign of
23590 *x*), respectively.23591 If the correct value would cause underflow, and is not representable, a range error may occur,
23592 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.23593 MX If *x* is NaN, a NaN shall be returned.23594 If *x* is ± 0 or \pm Inf, *x* shall be returned.23595 If *exp* is 0, *x* shall be returned.23596 If the correct value would cause underflow, and is representable, a range error may occur and
23597 the correct value shall be returned.23598 **ERRORS**

23599 These functions shall fail if:

23600 Range Error The result overflows.

23601 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
23602 then *errno* shall be set to [ERANGE]. If the integer expression
23603 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
23604 floating-point exception shall be raised.

23605 These functions may fail if:

23606 Range Error The result underflows.

23607 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
23608 then *errno* shall be set to [ERANGE]. If the integer expression
23609 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
23610 floating-point exception shall be raised.

23611 **EXAMPLES**
 23612 None.

23613 **APPLICATION USAGE**

23614 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
 23615 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23616 **RATIONALE**

23617 None.

23618 **FUTURE DIRECTIONS**

23619 None.

23620 **SEE ALSO**

23621 *feclearexcept()*, *fetestexcept()*, *frexp()*, *isnan()*, the Base Definitions volume of
 23622 IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
 23623 <math.h>

23624 **CHANGE HISTORY**

23625 First released in Issue 1. Derived from Issue 1 of the SVID.

23626 **Issue 5**

23627 The DESCRIPTION is updated to indicate how an application should check for an error. This
 23628 text was previously published in the APPLICATION USAGE section.

23629 **Issue 6**

23630 The *ldexpf()* and *ldexpl()* functions are added for alignment with the ISO/IEC 9899:1999
 23631 standard.

23632 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
 23633 revised to align with the ISO/IEC 9899:1999 standard.

23634 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
 23635 marked.

23636 **NAME**
 23637 ldiv, lldiv — compute quotient and remainder of a long division

23638 **SYNOPSIS**
 23639 #include <stdlib.h>
 23640 ldiv_t ldiv(long numer, long denom);
 23641 lldiv_t lldiv(long long numer, long long denom);

23642 **DESCRIPTION**
 23643 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 23644 conflict between the requirements described here and the ISO C standard is unintentional. This
 23645 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23646 These functions shall compute the quotient and remainder of the division of the numerator
 23647 *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the **long**
 23648 integer (for the *ldiv()* function) or **long long** integer (for the *lldiv()* function) of lesser magnitude
 23649 that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is
 23650 undefined; otherwise, *quot * denom + rem* shall equal *numer*.

23651 **RETURN VALUE**
 23652 The *ldiv()* function shall return a structure of type **ldiv_t**, comprising both the quotient and the
 23653 remainder. The structure shall include the following members, in any order:

23654 long quot; /* Quotient */
 23655 long rem; /* Remainder */

23656 The *lldiv()* function shall return a structure of type **lldiv_t**, comprising both the quotient and the
 23657 remainder. The structure shall include the following members, in any order:

23658 long long quot; /* Quotient */
 23659 long long rem; /* Remainder */

23660 **ERRORS**
 23661 No errors are defined.

23662 **EXAMPLES**
 23663 None.

23664 **APPLICATION USAGE**
 23665 None.

23666 **RATIONALE**
 23667 None.

23668 **FUTURE DIRECTIONS**
 23669 None.

23670 **SEE ALSO**
 23671 *div()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdlib.h>

23672 **CHANGE HISTORY**
 23673 First released in Issue 4. Derived from the ISO C standard.

23674 **Issue 6**
 23675 The *lldiv()* function is added for alignment with the ISO/IEC 9899:1999 standard.

lfind()

23676 **NAME**
23677 `lfind` — find entry in a linear search table

23678 **SYNOPSIS**

23679 XSI `#include <search.h>`
23680 `void *lfind(const void *key, const void *base, size_t *nelp,`
23681 `size_t width, int (*compar)(const void *, const void *));`

23682 **DESCRIPTION**

23683 Refer to [lsearch\(\)](#).

23684 **NAME**
 23685 lgamma, lgammaf, lgammal — log gamma function

23686 **SYNOPSIS**
 23687 #include <math.h>
 23688 double lgamma(double x);
 23689 float lgammaf(float x);
 23690 long double lgammal(long double x);
 23691 XSI extern int signgam;

23692 DESCRIPTION

23693 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 23694 conflict between the requirements described here and the ISO C standard is unintentional. This
 23695 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23696 These functions shall compute $\log_e |\Gamma(x)|$ where $\Gamma(x)$ is defined as $\int_0^{\infty} e^{-t} t^{x-1} dt$. The argument x
 23697 need not be a non-positive integer ($\Gamma(x)$ is defined over the reals, except the non-positive
 23698 integers).

23699 XSI The sign of $\Gamma(x)$ is returned in the external integer *signgam*.

23700 CX These functions need not be thread-safe. A function that is not required to be thread-safe is not
 23701 required to be reentrant.

23702 An application wishing to check for error situations should set *errno* to zero and call
 23703 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 23704 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 23705 zero, an error has occurred.

23706 RETURN VALUE

23707 Upon successful completion, these functions shall return the logarithmic gamma of x .

23708 If x is a non-positive integer, a pole error shall occur and *lgamma()*, *lgammaf()*, and *lgammal()*
 23709 shall return +HUGE_VAL, +HUGE_VALF, and +HUGE_VALL, respectively.

23710 If the correct value would cause overflow, a range error shall occur and *lgamma()*, *lgammaf()*,
 23711 and *lgammal()* shall return \pm HUGE_VAL, \pm HUGE_VALF, and \pm HUGE_VALL (having the same
 23712 sign as the correct value), respectively.

23713 MX If x is NaN, a NaN shall be returned.

23714 If x is 1 or 2, +0 shall be returned.

23715 If x is \pm Inf, +Inf shall be returned.

23716 ERRORS

23717 These functions shall fail if:

23718 Pole Error The x argument is a negative integer or zero.

23719 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 23720 then *errno* shall be set to [ERANGE]. If the integer expression
 23721 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by-zero
 23722 floating-point exception shall be raised.

23723 Range Error The result overflows.

23724 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,

23725 then *errno* shall be set to [ERANGE]. If the integer expression

23726 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow

23727 floating-point exception shall be raised.

EXAMPLES

23728 None.

23729

APPLICATION USAGE

23730 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &

23731 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23732

RATIONALE

23733 None.

23734

FUTURE DIRECTIONS

23735 None.

23736

SEE ALSO

23737 *exp()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-200x,

23738 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

23739

CHANGE HISTORY

23740 First released in Issue 3.

23741

Issue 5

23742 The DESCRIPTION is updated to indicate how an application should check for an error. This

23743 text was previously published in the APPLICATION USAGE section.

23744

23745 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

23746 The *lgamma()* function is no longer marked as an extension.

23747

23748 The *lgammaf()* and *lgammal()* functions are added for alignment with the ISO/IEC 9899:1999

23749 standard.

23750 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are

23751 revised to align with the ISO/IEC 9899:1999 standard.

23752 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are

23753 marked.

23754 Functionality relating to the XSI option is marked.

23755 **NAME**
 23756 link, linkat — link one file to another file relative to two directory file descriptors

23757 **SYNOPSIS**
 23758 #include <unistd.h>
 23759 int link(const char *path1, const char *path2);
 23760 int linkat(int fd1, const char *path1, int fd2, const char *path2,
 23761 int flag);

23762 **DESCRIPTION**
 23763 The *link()* function shall create a new link (directory entry) for the existing file, *path1*.
 23764 The *path1* argument points to a pathname naming an existing file. The *path2* argument points to
 23765 a pathname naming the new directory entry to be created. The *link()* function shall atomically
 23766 create a new link for the existing file and the link count of the file shall be incremented by one.

23767 If *path1* names a directory, *link()* shall fail unless the process has appropriate privileges and the
 23768 implementation supports using *link()* on directories.

23769 Upon successful completion, *link()* shall mark for update the *st_ctime* field of the file. Also, the
 23770 *st_ctime* and *st_mtime* fields of the directory that contains the new entry shall be marked for
 23771 update.

23772 If *link()* fails, no link shall be created and the link count of the file shall remain unchanged.

23773 The implementation may require that the calling process has permission to access the existing
 23774 file.

23775 The *linkat()* function shall be equivalent to the *link()* function except in the case where either
 23776 *path1* or *path2* or both are relative paths. In this case a relative path *path1* is interpreted relative to
 23777 the directory associated with the file descriptor *fd1* instead of the current working directory and
 23778 similarly for *path2* and the file descriptor *fd2*. It is unspecified whether directory searches are
 23779 permitted based on whether the file was opened with search permission or on the current
 23780 permissions of the directory underlying the file descriptor.

23781 Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined
 23782 in <fcntl.h>:

23783 AT_SYMLINK_FOLLOW
 23784 If *path1* names a symbolic link, a new link for the target of the symbolic link is
 23785 created.

23786 If *linkat()* is passed the special value AT_FDCWD in the *fd1* or *fd2* parameter, the current
 23787 working directory is used for the respective *path* argument. If both *fd1* and *fd2* have value
 23788 AT_FDCWD, the behavior shall be identical to a call to *link()*.

23789 Unless *flag* contains the AT_SYMLINK_FOLLOW flag, if *path1* names a symbolic link, a new link
 23790 is created for the symbolic link *path1* and not its target.

23791 **RETURN VALUE**
 23792 Upon successful completion, these functions shall return 0. Otherwise, these functions shall
 23793 return -1 and set *errno* to indicate the error.

23794 **ERRORS**
 23795 These functions shall fail if:
 23796 [EACCES] A component of either path prefix denies search permission, or the requested
 23797 link requires writing in a directory that denies write permission, or the calling
 23798 process does not have permission to access the existing file and this is required

link()

23799		by the implementation.
23800	[EEXIST]	The <i>path2</i> argument resolves to an existing file or refers to a symbolic link.
23801	[ELOOP]	A loop exists in symbolic links encountered during resolution of the <i>path1</i> or <i>path2</i> argument.
23802		
23803	[EMLINK]	The number of links to the file named by <i>path1</i> would exceed {LINK_MAX}.
23804	[ENAMETOOLONG]	
23805		The length of the <i>path1</i> or <i>path2</i> argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
23806		
23807	[ENOENT]	A component of either path prefix does not exist; the file named by <i>path1</i> does not exist; or <i>path1</i> or <i>path2</i> points to an empty string.
23808		
23809	[ENOSPC]	The directory to contain the link cannot be extended.
23810	[ENOTDIR]	A component of either path prefix is not a directory.
23811	[EPERM]	The file named by <i>path1</i> is a directory and either the calling process does not have appropriate privileges or the implementation prohibits using <i>link()</i> on directories.
23812		
23813		
23814	[EROFS]	The requested link requires writing in a directory on a read-only file system.
23815	[EXDEV]	The link named by <i>path2</i> and the file named by <i>path1</i> are on different file systems and the implementation does not support links between file systems.
23816		
23817	OB XSR [EXDEV]	<i>path1</i> refers to a named STREAM.
23818		The <i>linkat()</i> function shall fail if:
23819	[EBADF]	The <i>path1</i> or <i>path2</i> argument does not specify an absolute path and the <i>fd1</i> or <i>fd2</i> argument, respectively, is neither AT_FDCWD nor a valid file descriptor open for searching.
23820		
23821		
23822		These functions may fail if:
23823	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the <i>path1</i> or <i>path2</i> argument.
23824		
23825	[ENAMETOOLONG]	
23826		As a result of encountering a symbolic link in resolution of the <i>path1</i> or <i>path2</i> argument, the length of the substituted pathname string exceeded {PATH_MAX}.
23827		
23828		
23829		The <i>linkat()</i> function may fail if:
23830	[EINVAL]	The value of the <i>flag</i> argument is not valid.
23831	[ENOTDIR]	The <i>path1</i> or <i>path2</i> argument is not an absolute path and <i>fd1</i> or <i>fd2</i> , respectively, is neither AT_FDCWD nor a file descriptor associated with a directory.
23832		
23833		

EXAMPLES

Creating a Link to a File

The following example shows how to create a link to a file named `/home/cnd/mod1` by creating a new directory entry named `/modules/pass1`.

```
#include <unistd.h>

char *path1 = "/home/cnd/mod1";
char *path2 = "/modules/pass1";
int  status;
...
status = link (path1, path2);
```

Creating a Link to a File Within a Program

In the following program example, the `link()` function links the `/etc/passwd` file (defined as `PASSWDFILE`) to a file named `/etc/opasswd` (defined as `SAVEFILE`), which is used to save the current password file. Then, after removing the current password file (defined as `PASSWDFILE`), the new password file is saved as the current password file using the `link()` function again.

```
#include <unistd.h>

#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"
#define SAVEFILE "/etc/opasswd"
...
/* Save current password file */
link (PASSWDFILE, SAVEFILE);

/* Remove current password file. */
unlink (PASSWDFILE);

/* Save new password file as current password file. */
link (LOCKFILE, PASSWDFILE);
```

APPLICATION USAGE

Some implementations do allow links between file systems.

RATIONALE

Linking to a directory is restricted to the superuser in most historical implementations because this capability may produce loops in the file hierarchy or otherwise corrupt the file system. This volume of IEEE Std 1003.1-200x continues that philosophy by prohibiting `link()` and `unlink()` from doing this. Other functions could do it if the implementor designed such an extension.

Some historical implementations allow linking of files on different file systems. Wording was added to explicitly allow this optional behavior.

The exception for cross-file system links is intended to apply only to links that are programmatically indistinguishable from “hard” links.

The purpose of the `linkat()` function is to link files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to `link()`, resulting in unspecified behavior. By opening a file descriptor for the directory of both the existing file and the target location and using the `linkat()` function it can be guaranteed that the both filenames are in the desired directories.

The `AT_SYMLINK_FOLLOW` flag allows for implementing both common behaviors of the `link()` function. The POSIX specification requires that if `path1` is a symbolic link, a new link for

23879 the target of the symbolic link is created. Many systems by default or as an alternative provide a
 23880 mechanism to avoid the implicit symlink lookup and create a new link for the symbolic link
 23881 itself.

23882 Earlier versions of this standard specified only the *link()* function, and required it to behave like
 23883 *linkat()* with the `AT_SYMLINK_FOLLOW` flag. However, historical practice from SVR4 and
 23884 Linux kernels had *link()* behaving like *linkat()* with no flags, and many systems that attempted
 23885 to provide a conforming *link()* function did so in a way that was rarely used, and when it was
 23886 used did not conform to the standard (e.g., by not being atomic, or by dereferencing the
 23887 symbolic link incorrectly). Since applications could not rely on *link()* following links in practice,
 23888 the *linkat()* function was added taking a flag to specify the desired behavior for the application.

23889 **FUTURE DIRECTIONS**

23890 None.

23891 **SEE ALSO**

23892 *rename()*, *symlink()*, *unlink()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<fcntl.h>`,
 23893 `<unistd.h>`

23894 **CHANGE HISTORY**

23895 First released in Issue 1. Derived from Issue 1 of the SVID.

23896 **Issue 6**

23897 The following new requirements on POSIX implementations derive from alignment with the
 23898 Single UNIX Specification:

- 23899 • The [ELOOP] mandatory error condition is added.
- 23900 • A second [ENAMETOOLONG] is added as an optional error condition.

23901 The following changes were made to align with the IEEE P1003.1a draft standard:

- 23902 • An explanation is added of the action when *path2* refers to a symbolic link.
- 23903 • The [ELOOP] optional error condition is added.

23904 **Issue 7**

23905 SD5-XSH-ERN-93 is applied, adding RATIONALE.

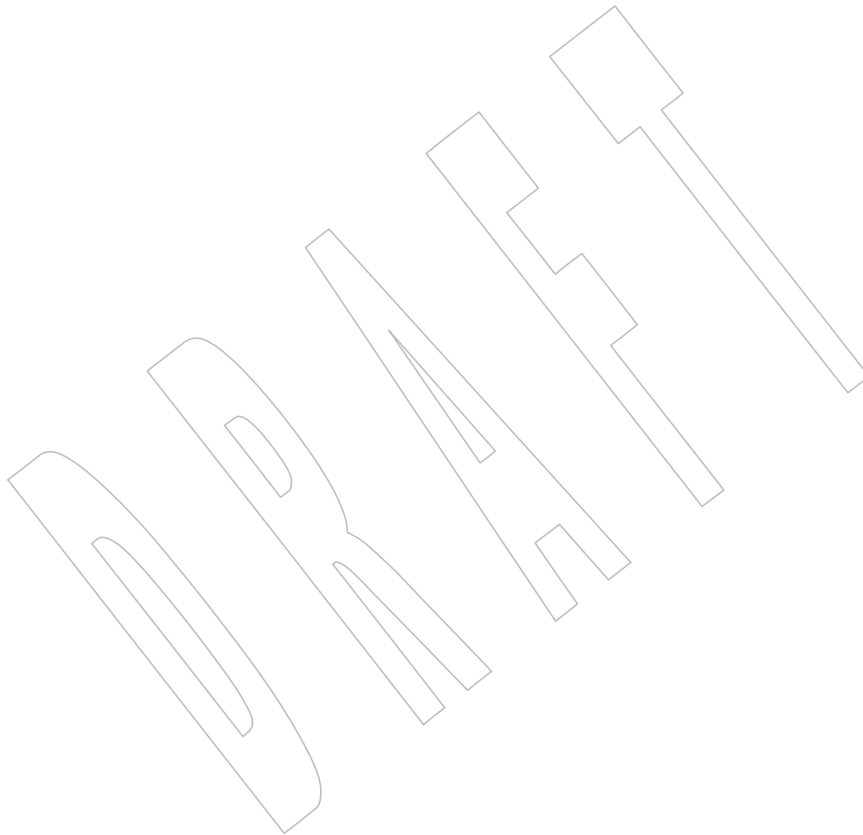
23906 The *linkat()* function is added from The Open Group Technical Standard, 2006, Extended API Set
 23907 Part 2.

23908 Functionality relating to XSI STREAMS is marked obsolescent.

23909 **NAME**
23910 linkat — link one file to another file relative to two directory file descriptors

23911 **SYNOPSIS**
23912 #include <unistd.h>
23913 int linkat(int *fd1*, const char **path1*, int *fd2*, const char **path2*,
23914 int *flag*);

23915 **DESCRIPTION**
23916 Refer to [link\(\)](#).



23917 **NAME**

23918 lio_listio — list directed I/O

23919 **SYNOPSIS**

23920 #include <aio.h>

23921 int lio_listio(int mode, struct aiocb *restrict const list[restrict],
23922 int nent, struct sigevent *restrict sig);23923 **DESCRIPTION**23924 The *lio_listio()* function shall initiate a list of I/O requests with a single function call.23925 The *mode* argument takes one of the values LIO_WAIT or LIO_NOWAIT declared in <aio.h> and
23926 determines whether the function returns when the I/O operations have been completed, or as
23927 soon as the operations have been queued. If the *mode* argument is LIO_WAIT, the function shall
23928 wait until all I/O is complete and the *sig* argument shall be ignored.23929 If the *mode* argument is LIO_NOWAIT, the function shall return immediately, and asynchronous
23930 notification shall occur, according to the *sig* argument, when all the I/O operations complete. If
23931 *sig* is NULL, then no asynchronous notification shall occur. If *sig* is not NULL, asynchronous
23932 notification occurs as specified in Section 2.4.1 when all the requests in *list* have completed.23933 The I/O requests enumerated by *list* are submitted in an unspecified order.23934 The *list* argument is an array of pointers to **aiocb** structures. The array contains *nent* elements.
23935 The array may contain NULL elements, which shall be ignored.23936 If the buffer pointed to by *list* or the **aiocb** structures pointed to by the elements of the array *list*
23937 become illegal addresses before all asynchronous I/O completed and, if necessary, the
23938 notification is sent, then the behavior is undefined. If the buffers pointed to by the *aio_buf*
23939 member of the **aiocb** structure pointed to by the elements of the array *list* become illegal
23940 addresses prior to the asynchronous I/O associated with that **aiocb** structure being completed,
23941 the behavior is undefined.23942 The *aio_lio_opcode* field of each **aiocb** structure specifies the operation to be performed. The
23943 supported operations are LIO_READ, LIO_WRITE, and LIO_NOP; these symbols are defined in
23944 <aio.h>. The LIO_NOP operation causes the list entry to be ignored. If the *aio_lio_opcode*
23945 element is equal to LIO_READ, then an I/O operation is submitted as if by a call to *aio_read()*
23946 with the *aio_cbp* equal to the address of the **aiocb** structure. If the *aio_lio_opcode* element is equal to
23947 LIO_WRITE, then an I/O operation is submitted as if by a call to *aio_write()* with the *aio_cbp*
23948 equal to the address of the **aiocb** structure.23949 The *aio_fildes* member specifies the file descriptor on which the operation is to be performed.23950 The *aio_buf* member specifies the address of the buffer to or from which the data is transferred.23951 The *aio_nbytes* member specifies the number of bytes of data to be transferred.23952 The members of the **aiocb** structure further describe the I/O operation to be performed, in a
23953 manner identical to that of the corresponding **aiocb** structure when used by the *aio_read()* and
23954 *aio_write()* functions.23955 The *nent* argument specifies how many elements are members of the list; that is, the length of the
23956 array.23957 The behavior of this function is altered according to the definitions of synchronized I/O data
23958 integrity completion and synchronized I/O file integrity completion if synchronized I/O is
23959 enabled on the file associated with *aio_fildes*.

23960 For regular files, no data transfer shall occur past the offset maximum established in the open

23961 file description associated with *aio*cbp->*aio_fildes*.

23962 If *sig*->*sige*v_notify is SIGEV_THREAD and *sig*->*sige*v_notify_attributes is a non-NULL pointer
23963 and the block pointed to by this pointer becomes an illegal address prior to all asynchronous
23964 I/O being completed, then the behavior is undefined.

23965 RETURN VALUE

23966 If the *mode* argument has the value LIO_NOWAIT, the *lio_listio*() function shall return the value
23967 zero if the I/O operations are successfully queued; otherwise, the function shall return the value
23968 -1 and set *errno* to indicate the error.

23969 If the *mode* argument has the value LIO_WAIT, the *lio_listio*() function shall return the value zero
23970 when all the indicated I/O has completed successfully. Otherwise, *lio_listio*() shall return a value
23971 of -1 and set *errno* to indicate the error.

23972 In either case, the return value only indicates the success or failure of the *lio_listio*() call itself,
23973 not the status of the individual I/O requests. In some cases one or more of the I/O requests
23974 contained in the list may fail. Failure of an individual request does not prevent completion of
23975 any other individual request. To determine the outcome of each I/O request, the application
23976 shall examine the error status associated with each **aio**cb control block. The error statuses so
23977 returned are identical to those returned as the result of an *aio_read*() or *aio_write*() function.

23978 ERRORS

23979 The *lio_listio*() function shall fail if:

23980 [EAGAIN] The resources necessary to queue all the I/O requests were not available. The
23981 application may check the error status for each **aio**cb to determine the
23982 individual request(s) that failed.

23983 [EAGAIN] The number of entries indicated by *nent* would cause the system-wide limit
23984 {AIO_MAX} to be exceeded.

23985 [EINVAL] The *mode* argument is not a proper value, or the value of *nent* was greater than
23986 {AIO_LISTIO_MAX}.

23987 [EINTR] A signal was delivered while waiting for all I/O requests to complete during
23988 an LIO_WAIT operation. Note that, since each I/O operation invoked by
23989 *lio_listio*() may possibly provoke a signal when it completes, this error return
23990 may be caused by the completion of one (or more) of the very I/O operations
23991 being awaited. Outstanding I/O requests are not canceled, and the application
23992 shall examine each list element to determine whether the request was
23993 initiated, canceled, or completed.

23994 [EIO] One or more of the individual I/O operations failed. The application may
23995 check the error status for each **aio**cb structure to determine the individual
23996 request(s) that failed.

23997 In addition to the errors returned by the *lio_listio*() function, if the *lio_listio*() function succeeds
23998 or fails with errors of [EAGAIN], [EINTR], or [EIO], then some of the I/O specified by the list
23999 may have been initiated. If the *lio_listio*() function fails with an error code other than [EAGAIN],
24000 [EINTR], or [EIO], no operations from the list shall have been initiated. The I/O operation
24001 indicated by each list element can encounter errors specific to the individual read or write
24002 function being performed. In this event, the error status for each **aio**cb control block contains the
24003 associated error code. The error codes that can be set are the same as would be set by a *read*() or
24004 *write*() function, with the following additional error codes possible:

24005 [EAGAIN] The requested I/O operation was not queued due to resource limitations.

24006 [ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit
24007 *aio_cancel*() request.

lio_listio()

- 24008 [EFBIG] The *aiocbp->aio_lio_opcode* is LIO_WRITE, the file is a regular file,
24009 *aiocbp->aio_nbytes* is greater than 0, and the *aiocbp->aio_offset* is greater than or
24010 equal to the offset maximum in the open file description associated with
24011 *aiocbp->aio_fildes*.
- 24012 [EINPROGRESS] The requested I/O is in progress.
- 24013 [EOVERFLOW] The *aiocbp->aio_lio_opcode* is LIO_READ, the file is a regular file,
24014 *aiocbp->aio_nbytes* is greater than 0, and the *aiocbp->aio_offset* is before the
24015 end-of-file and is greater than or equal to the offset maximum in the open file
24016 description associated with *aiocbp->aio_fildes*.

EXAMPLES

24017 None.
24018

APPLICATION USAGE

24019 None.
24020

RATIONALE

24021 Although it may appear that there are inconsistencies in the specified circumstances for error
24022 codes, the [EIO] error condition applies when any circumstance relating to an individual
24023 operation makes that operation fail. This might be due to a badly formulated request (for
24024 example, the *aio_lio_opcode* field is invalid, and *aio_error()* returns [EINVAL]) or might arise from
24025 application behavior (for example, the file descriptor is closed before the operation is initiated,
24026 and *aio_error()* returns [EBADF]).
24027

24028 The limitation on the set of error codes returned when operations from the list shall have been
24029 initiated enables applications to know when operations have been started and whether
24030 *aio_error()* is valid for a specific operation.

FUTURE DIRECTIONS

24031 None.
24032

SEE ALSO

24033 *aio_read()*, *aio_write()*, *aio_error()*, *aio_return()*, *aio_cancel()*, *close()*, *exec*, *exit()*, *fork()*, *lseek()*,
24034 *read()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**aio.h**>
24035

CHANGE HISTORY

24036 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.
24037

24038 Large File Summit extensions are added.

Issue 6

24039 The [ENOSYS] error condition has been removed as stubs need not be provided if an
24040 implementation does not support the Asynchronous Input and Output option.
24041

24042 The *lio_listio()* function is marked as part of the Asynchronous Input and Output option.

24043 The following new requirements on POSIX implementations derive from alignment with the
24044 Single UNIX Specification:

- 24045 • In the DESCRIPTION, text is added to indicate that for regular files no data transfer occurs
24046 past the offset maximum established in the open file description associated with
24047 *aiocbp->aio_fildes*. This change is to support large files.
- 24048 • The [EBIG] and [EOVERFLOW] error conditions are defined. This change is to support
24049 large files.

24050 The normative text is updated to avoid use of the term “must” for application requirements.

24051 The **restrict** keyword is added to the *lio_listio()* prototype for alignment with the
24052 ISO/IEC 9899:1999 standard.

24053

Issue 6

24054

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/53 is applied, adding new text for symmetry with the *aio_read()* and *aio_write()* functions to the DESCRIPTION.

24055

24056

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/54 is applied, adding text to the DESCRIPTION making it explicit that the user is required to keep the structure pointed to by *sig->sigev_notify_attributes* valid until the last asynchronous operation finished and the notification has been sent.

24057

24058

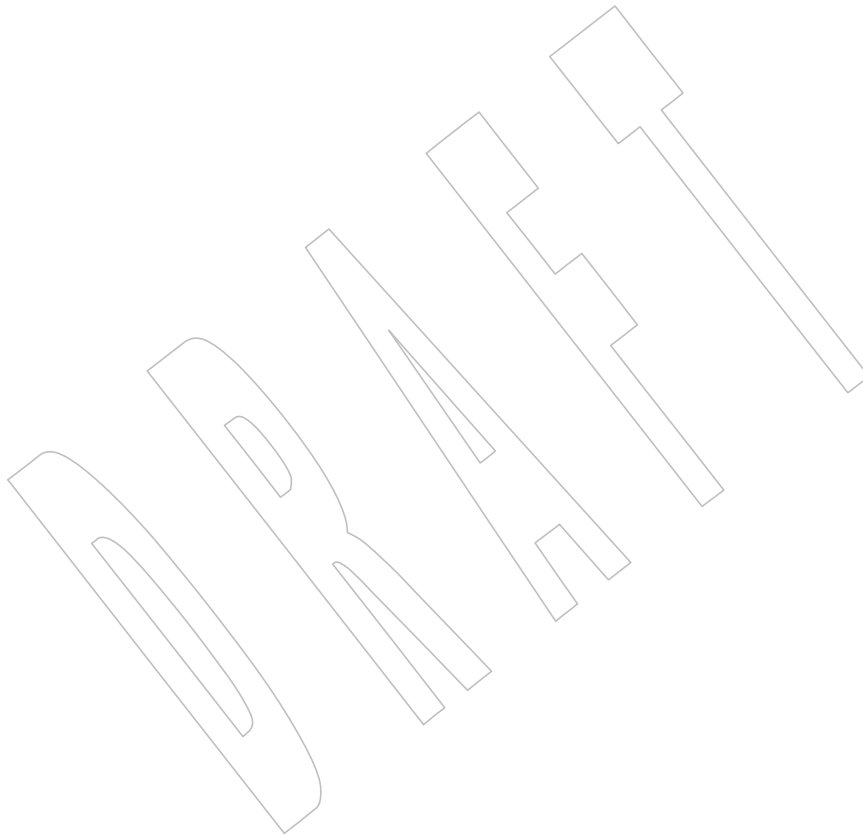
24059

24060

Issue 7

24061

The *lio_listio()* function is moved from the Asynchronous Input and Output option to the Base.



24062 **NAME**24063 `listen` — listen for socket connections and limit the queue of incoming connections24064 **SYNOPSIS**24065 `#include <sys/socket.h>`24066 `int listen(int socket, int backlog);`24067 **DESCRIPTION**24068 The `listen()` function shall mark a connection-mode socket, specified by the `socket` argument, as
24069 accepting connections.24070 The `backlog` argument provides a hint to the implementation which the implementation shall use
24071 to limit the number of outstanding connections in the socket's listen queue. Implementations
24072 may impose a limit on `backlog` and silently reduce the specified value. Normally, a larger `backlog`
24073 argument value shall result in a larger or equal length of the listen queue. Implementations shall
24074 support values of `backlog` up to `SOMAXCONN`, defined in `<sys/socket.h>`.24075 The implementation may include incomplete connections in its listen queue. The limits on the
24076 number of incomplete connections and completed connections queued may be different.24077 The implementation may have an upper limit on the length of the listen queue—either global or
24078 per accepting socket. If `backlog` exceeds this limit, the length of the listen queue is set to the limit.24079 If `listen()` is called with a `backlog` argument value that is less than 0, the function behaves as if it
24080 had been called with a `backlog` argument value of 0.24081 A `backlog` argument of 0 may allow the socket to accept connections, in which case the length of
24082 the listen queue may be set to an implementation-defined minimum value.24083 The socket in use may require the process to have appropriate privileges to use the `listen()`
24084 function.24085 **RETURN VALUE**24086 Upon successful completions, `listen()` shall return 0; otherwise, `-1` shall be returned and `errno` set
24087 to indicate the error.24088 **ERRORS**24089 The `listen()` function shall fail if:24090 [EBADF] The `socket` argument is not a valid file descriptor.

24091 [EDESTADDRREQ]

24092 The socket is not bound to a local address, and the protocol does not support
24093 listening on an unbound socket.24094 [EINVAL] The `socket` is already connected.24095 [ENOTSOCK] The `socket` argument does not refer to a socket.24096 [EOPNOTSUPP] The socket protocol does not support `listen()`.24097 The `listen()` function may fail if:

24098 [EACCES] The calling process does not have the appropriate privileges.

24099 [EINVAL] The `socket` has been shut down.

24100 [ENOBUFS] Insufficient resources are available in the system to complete the call.

24101

EXAMPLES

24102

None.

24103

APPLICATION USAGE

24104

None.

24105

RATIONALE

24106

None.

24107

FUTURE DIRECTIONS

24108

None.

24109

SEE ALSO

24110

accept(), *connect()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-200x, <sys/socket.h>

24111

CHANGE HISTORY

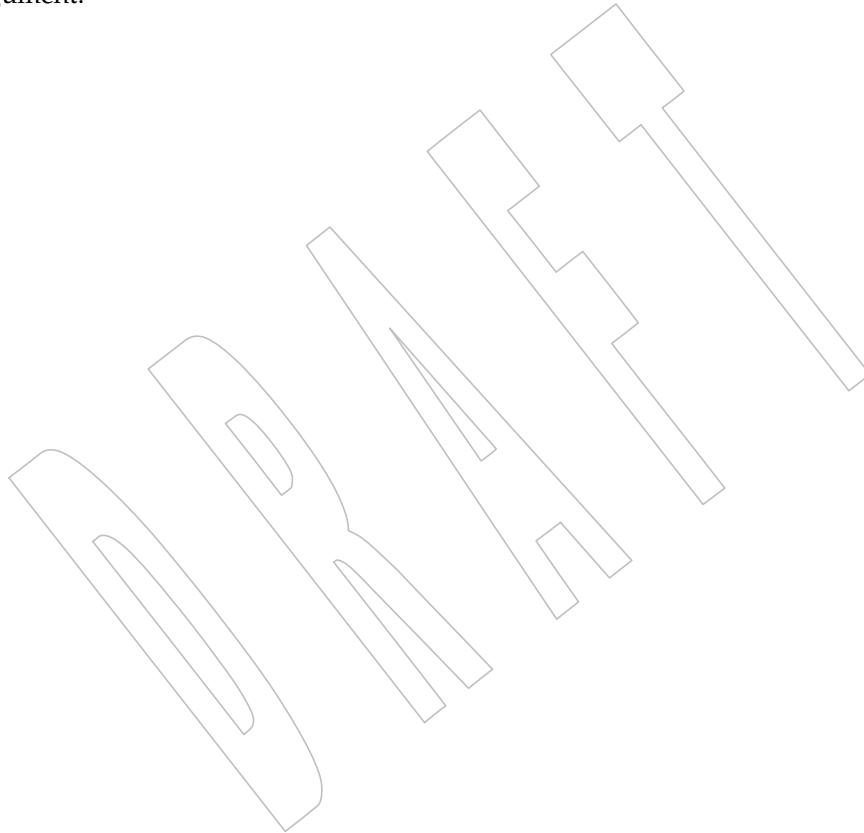
24112

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

24113

The DESCRIPTION is updated to describe the relationship of SOMAXCONN and the *backlog* argument.

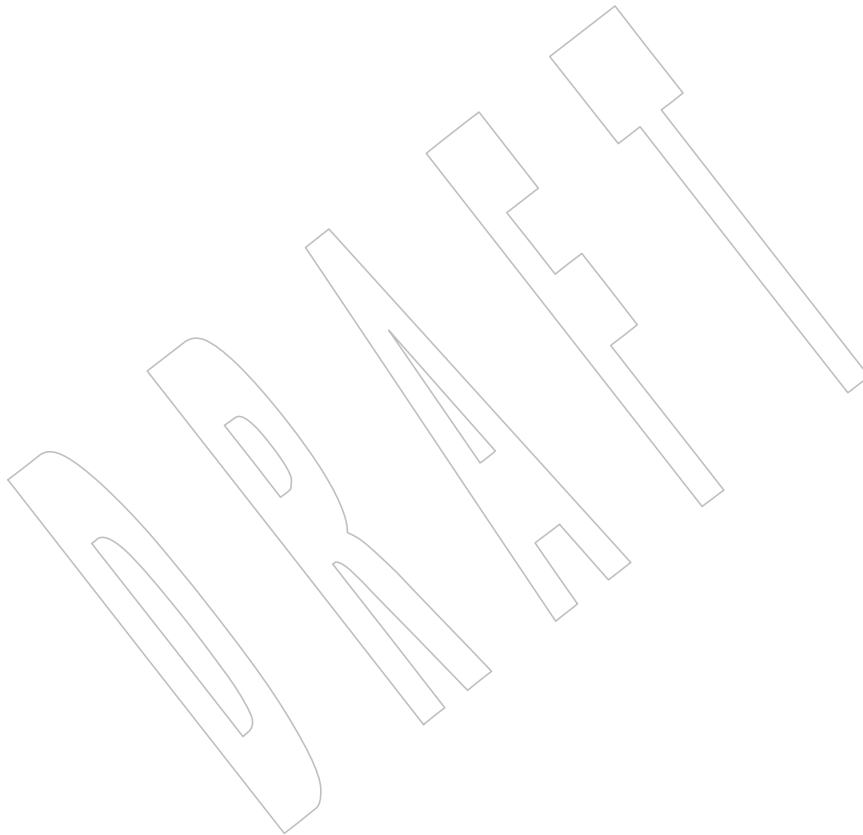
24114



24115 **NAME**
24116 llabs — return a long integer absolute value

24117 **SYNOPSIS**
24118 #include <stdlib.h>
24119 long long llabs(long long i);

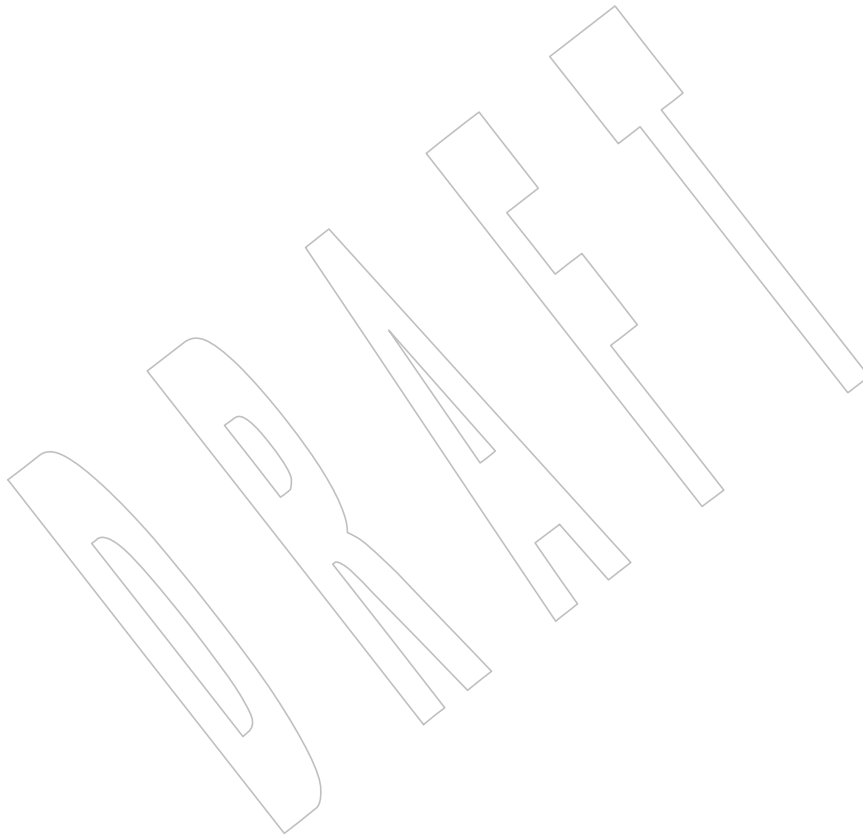
24120 **DESCRIPTION**
24121 Refer to *labs()*.



24122 **NAME**
24123 `lldiv` — compute quotient and remainder of a long division

24124 **SYNOPSIS**
24125 `#include <stdlib.h>`
24126 `lldiv_t lldiv(long long numer, long long denom);`

24127 **DESCRIPTION**
24128 Refer to *ldiv()*.



24129 **NAME**

24130 llrint, llrintf, llrintl — round to the nearest integer value using current rounding direction

24131 **SYNOPSIS**

```
24132 #include <math.h>
24133
24133 long long llrint(double x);
24134 long long llrintf(float x);
24135 long long llrintl(long double x);
```

24136 **DESCRIPTION**

24137 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 24138 conflict between the requirements described here and the ISO C standard is unintentional. This
 24139 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

24140 These functions shall round their argument to the nearest integer value, rounding according to
 24141 the current rounding direction.

24142 An application wishing to check for error situations should set *errno* to zero and call
 24143 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 24144 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 24145 zero, an error has occurred.

24146 **RETURN VALUE**

24147 Upon successful completion, these functions shall return the rounded integer value.

24148 MX If *x* is NaN, a domain error shall occur, and an unspecified value is returned.24149 If *x* is +Inf, a domain error shall occur and an unspecified value is returned.24150 If *x* is -Inf, a domain error shall occur and an unspecified value is returned.

24151 If the correct value is positive and too large to represent as a **long long**, an unspecified value
 24152 MX shall be returned. On systems that support the IEC 60559 Floating-Point option, a domain error
 24153 shall occur;

24154 CX otherwise, a domain error may occur.

24155 If the correct value is negative and too large to represent as a **long long**, an unspecified value
 24156 MX shall be returned. On systems that support the IEC 60559 Floating-Point option, a domain error
 24157 shall occur;

24158 CX otherwise, a domain error may occur.

24159 **ERRORS**

24160 These functions shall fail if:

24161 MX **Domain Error** The *x* argument is NaN or ±Inf, or the correct value is not representable as an
 24162 integer.

24163 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 24164 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 24165 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 24166 shall be raised.

24167 These functions may fail if:

24168 **Domain Error** The correct value is not representable as an integer.

24169 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 24170 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 24171 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception

24172 shall be raised.

24173 **EXAMPLES**

24174 None.

24175 **APPLICATION USAGE**

24176 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
24177 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

24178 **RATIONALE**

24179 These functions provide floating-to-integer conversions. They round according to the current
24180 rounding direction. If the rounded value is outside the range of the return type, the numeric
24181 result is unspecified and the invalid floating-point exception is raised. When they raise no other
24182 floating-point exception and the result differs from the argument, they raise the inexact floating-
24183 point exception.

24184 **FUTURE DIRECTIONS**

24185 None.

24186 **SEE ALSO**

24187 *feclearexcept()*, *fetestexcept()*, *lrint()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section
24188 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

24189 **CHANGE HISTORY**

24190 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

24191 **Issue 7**

24192 ISO C TC2 #53 is applied.

24193 **NAME**

24194 llround, llroundf, llroundl — round to nearest integer value

24195 **SYNOPSIS**

```
24196 #include <math.h>
24197 long long llround(double x);
24198 long long llroundf(float x);
24199 long long llroundl(long double x);
```

24200 **DESCRIPTION**

24201 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 24202 conflict between the requirements described here and the ISO C standard is unintentional. This
 24203 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

24204 These functions shall round their argument to the nearest integer value, rounding halfway cases
 24205 away from zero, regardless of the current rounding direction.

24206 An application wishing to check for error situations should set *errno* to zero and call
 24207 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 24208 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 24209 zero, an error has occurred.

24210 **RETURN VALUE**

24211 Upon successful completion, these functions shall return the rounded integer value.

24212 MX If *x* is NaN, a domain error shall occur, and an unspecified value is returned.24213 If *x* is +Inf, a domain error shall occur and an unspecified value is returned.24214 If *x* is -Inf, a domain error shall occur and an unspecified value is returned.

24215 If the correct value is positive and too large to represent as a **long long**, an unspecified value
 24216 MX shall be returned. On systems that support the IEC 60559 Floating-Point option, a domain error
 24217 shall occur;

24218 CX otherwise, a domain error may occur.

24219 If the correct value is negative and too large to represent as a **long long**, an unspecified value
 24220 MX shall be returned. On systems that support the IEC 60559 Floating-Point option, a domain error
 24221 shall occur;

24222 CX otherwise, a domain error may occur.

24223 **ERRORS**

24224 These functions shall fail if:

24225 MX **Domain Error** The *x* argument is NaN or ±Inf, or the correct value is not representable as an
 24226 integer.

24227 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 24228 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 24229 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 24230 shall be raised.

24231 These functions may fail if:

24232 **Domain Error** The correct value is not representable as an integer.

24233 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 24234 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 24235 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception

24236 shall be raised.

24237 **EXAMPLES**

24238 None.

24239 **APPLICATION USAGE**

24240 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
24241 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

24242 **RATIONALE**

24243 These functions differ from the *llrint()* functions in that the default rounding direction for the
24244 *llround()* functions round halfway cases away from zero and need not raise the inexact floating-
24245 point exception for non-integer arguments that round to within the range of the return type.

24246 **FUTURE DIRECTIONS**

24247 None.

24248 **SEE ALSO**

24249 *feclearexcept()*, *fetestexcept()*, *lround()*, the Base Definitions volume of IEEE Std 1003.1-200x,
24250 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

24251 **CHANGE HISTORY**

24252 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

24253 **Issue 7**

24254 ISO/IEC 9899:1999 standard, Technical Corrigendum 2 #54 (SD5-XSH-ERN-75) is applied.

24255 **NAME**
 24256 localeconv — return locale-specific information

24257 **SYNOPSIS**
 24258 #include <locale.h>

24259 struct lconv *localeconv(void);

24260 **DESCRIPTION**

24261 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 24262 conflict between the requirements described here and the ISO C standard is unintentional. This
 24263 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

24264 The *localeconv()* function shall set the components of an object with the type **struct lconv** with
 24265 the values appropriate for the formatting of numeric quantities (monetary and otherwise)
 24266 according to the rules of the current locale.

24267 The members of the structure with type **char *** are pointers to strings, any of which (except
 24268 **decimal_point**) can point to " ", to indicate that the value is not available in the current locale or
 24269 is of zero length. The members with type **char** are non-negative numbers, any of which can be
 24270 {CHAR_MAX} to indicate that the value is not available in the current locale.

24271 The members include the following:

24272 **char *decimal_point**
 24273 The radix character used to format non-monetary quantities.

24274 **char *thousands_sep**
 24275 The character used to separate groups of digits before the decimal-point character in
 24276 formatted non-monetary quantities.

24277 **char *grouping**
 24278 A string whose elements taken as one-byte integer values indicate the size of each group of
 24279 digits in formatted non-monetary quantities.

24280 **char *int_curr_symbol**
 24281 The international currency symbol applicable to the current locale. The first three characters
 24282 contain the alphabetic international currency symbol in accordance with those specified in
 24283 the ISO 4217:2001 standard. The fourth character (immediately preceding the null byte) is
 24284 the character used to separate the international currency symbol from the monetary
 24285 quantity.

24286 **char *currency_symbol**
 24287 The local currency symbol applicable to the current locale.

24288 **char *mon_decimal_point**
 24289 The radix character used to format monetary quantities.

24290 **char *mon_thousands_sep**
 24291 The separator for groups of digits before the decimal-point in formatted monetary
 24292 quantities.

24293 **char *mon_grouping**
 24294 A string whose elements taken as one-byte integer values indicate the size of each group of
 24295 digits in formatted monetary quantities.

24296 **char *positive_sign**
 24297 The string used to indicate a non-negative valued formatted monetary quantity.

- 24298 **char *negative_sign**
 24299 The string used to indicate a negative valued formatted monetary quantity.
- 24300 **char int_frac_digits**
 24301 The number of fractional digits (those after the decimal-point) to be displayed in an
 24302 internationally formatted monetary quantity.
- 24303 **char frac_digits**
 24304 The number of fractional digits (those after the decimal-point) to be displayed in a
 24305 formatted monetary quantity.
- 24306 **char p_cs_precedes**
 24307 Set to 1 if the **currency_symbol** precedes the value for a non-negative formatted monetary
 24308 quantity. Set to 0 if the symbol succeeds the value.
- 24309 **char p_sep_by_space**
 24310 Set to a value indicating the separation of the **currency_symbol**, the sign string, and the
 24311 value for a non-negative formatted monetary quantity.
- 24312 **char n_cs_precedes**
 24313 Set to 1 if the **currency_symbol** precedes the value for a negative formatted monetary
 24314 quantity. Set to 0 if the symbol succeeds the value.
- 24315 **char n_sep_by_space**
 24316 Set to a value indicating the separation of the **currency_symbol**, the sign string, and the
 24317 value for a negative formatted monetary quantity.
- 24318 **char p_sign_posn**
 24319 Set to a value indicating the positioning of the **positive_sign** for a non-negative formatted
 24320 monetary quantity.
- 24321 **char n_sign_posn**
 24322 Set to a value indicating the positioning of the **negative_sign** for a negative formatted
 24323 monetary quantity.
- 24324 **char int_p_cs_precedes**
 24325 Set to 1 or 0 if the **int_curr_symbol** respectively precedes or succeeds the value for a non-
 24326 negative internationally formatted monetary quantity.
- 24327 **char int_n_cs_precedes**
 24328 Set to 1 or 0 if the **int_curr_symbol** respectively precedes or succeeds the value for a
 24329 negative internationally formatted monetary quantity.
- 24330 **char int_p_sep_by_space**
 24331 Set to a value indicating the separation of the **int_curr_symbol**, the sign string, and the
 24332 value for a non-negative internationally formatted monetary quantity.
- 24333 **char int_n_sep_by_space**
 24334 Set to a value indicating the separation of the **int_curr_symbol**, the sign string, and the
 24335 value for a negative internationally formatted monetary quantity.
- 24336 **char int_p_sign_posn**
 24337 Set to a value indicating the positioning of the **positive_sign** for a non-negative
 24338 internationally formatted monetary quantity.
- 24339 **char int_n_sign_posn**
 24340 Set to a value indicating the positioning of the **negative_sign** for a negative internationally
 24341 formatted monetary quantity.
- 24342 The elements of **grouping** and **mon_grouping** are interpreted according to the following:

localeconv()

- 24343 {CHAR_MAX} No further grouping is to be performed.
- 24344 0 The previous element is to be repeatedly used for the remainder of the digits.
- 24345 *other* The integer value is the number of digits that comprise the current group. The
- 24346 next element is examined to determine the size of the next group of digits
- 24347 before the current group.

24348 The values of **p_sep_by_space**, **n_sep_by_space**, **int_p_sep_by_space**, and **int_n_sep_by_space**

24349 are interpreted according to the following:

- 24350 0 No space separates the currency symbol and value.
- 24351 1 If the currency symbol and sign string are adjacent, a space separates them from the value;
- 24352 otherwise, a space separates the currency symbol from the value.
- 24353 2 If the currency symbol and sign string are adjacent, a space separates them; otherwise, a
- 24354 space separates the sign string from the value.

24355 For **int_p_sep_by_space** and **int_n_sep_by_space**, the fourth character of **int_curr_symbol** is

24356 used instead of a space.

24357 The values of **p_sign_posn**, **n_sign_posn**, **int_p_sign_posn**, and **int_n_sign_posn** are

24358 interpreted according to the following:

- 24359 0 Parentheses surround the quantity and **currency_symbol** or **int_curr_symbol**.
- 24360 1 The sign string precedes the quantity and **currency_symbol** or **int_curr_symbol**.
- 24361 2 The sign string succeeds the quantity and **currency_symbol** or **int_curr_symbol**.
- 24362 3 The sign string immediately precedes the **currency_symbol** or **int_curr_symbol**.
- 24363 4 The sign string immediately succeeds the **currency_symbol** or **int_curr_symbol**.

24364 The implementation shall behave as if no function in this volume of IEEE Std 1003.1-200x calls

24365 *localeconv()*.

24366 CX The *localeconv()* function need not be thread-safe. A function that is not required to be thread-

24367 safe is not required to be reentrant.

RETURN VALUE

24368 The *localeconv()* function shall return a pointer to the filled-in object. The application shall not

24369 modify the structure pointed to by the return value which may be overwritten by a subsequent

24370 call to *localeconv()*. In addition, calls to *setlocale()* with the categories *LC_ALL*, *LC_MONETARY*,

24371 or *LC_NUMERIC* or calls to *uselocale()* which change the categories *LC_MONETARY* or

24372 *LC_NUMERIC* may overwrite the contents of the structure.

ERRORS

24374 No errors are defined.

EXAMPLES

24376 None.

APPLICATION USAGE

24378 The following table illustrates the rules which may be used by four countries to format

24379 monetary quantities.

Country	Positive Format	Negative Format	International Format
Italy	L.1.230	-L.1.230	ITL.1.230
Netherlands	F 1.234,56	F -1.234,56	NLG 1.234,56
Norway	kr1.234,56	kr1.234,56-	NOK 1.234,56
Switzerland	SFrs.1,234.56	SFrs.1,234.56C	CHF 1,234.56

24386

For these four countries, the respective values for the monetary members of the structure returned by *localeconv()* are:

24387

24388

	Italy	Netherlands	Norway	Switzerland
<code>int_curr_symbol</code>	"ITL. "	"NLG "	"NOK "	"CHF "
<code>currency_symbol</code>	"L. "	"F"	"kr"	"SFrS."
<code>mon_decimal_point</code>	" "	","	","	."
<code>mon_thousands_sep</code>	."	."	."	","
<code>mon_grouping</code>	"\3"	"\3"	"\3"	"\3"
<code>positive_sign</code>	" "	" "	" "	" "
<code>negative_sign</code>	"-"	"-"	"-"	"C"
<code>int_frac_digits</code>	0	2	2	2
<code>frac_digits</code>	0	2	2	2
<code>p_cs_precedes</code>	1	1	1	1
<code>p_sep_by_space</code>	0	1	0	0
<code>n_cs_precedes</code>	1	1	1	1
<code>n_sep_by_space</code>	0	1	0	0
<code>p_sign_posn</code>	1	1	1	1
<code>n_sign_posn</code>	1	4	2	2
<code>int_p_cs_precedes</code>	1	1	1	1
<code>int_n_cs_precedes</code>	1	1	1	1
<code>int_p_sep_by_space</code>	0	0	0	0
<code>int_n_sep_by_space</code>	0	0	0	0
<code>int_p_sign_posn</code>	1	1	1	1
<code>int_n_sign_posn</code>	1	4	4	2

24410

RATIONALE

24411

None.

24412

FUTURE DIRECTIONS

24413

None.

24414

SEE ALSO

24415

isalpha(), *isascii()*, *nl_langinfo()*, *printf()*, *scanf()*, *setlocale()*, *strcat()*, *strchr()*, *strcmp()*, *strcoll()*, *strcpy()*, *strftime()*, *strlen()*, *strpbrk()*, *strspn()*, *strtok()*, *strxfrm()*, *strtod()*, *uselocale()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<langinfo.h>`, `<locale.h>`

24417

24418

CHANGE HISTORY

24419

First released in Issue 4. Derived from the ANSI C standard.

24420

Issue 6

24421

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

24422

The RETURN VALUE section is rewritten to avoid use of the term "must".

24423

This reference page is updated for alignment with the ISO/IEC 9899:1999 standard.

24424

ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

24425

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/31 is applied, removing references to `int_curr_symbol` and updating the descriptions of `p_sep_by_space` and `n_sep_by_space`. These changes are for alignment with the ISO C standard.

24426

24427

NAME

`localtime`, `localtime_r` — convert a time value to a broken-down local time

SYNOPSIS

```
#include <time.h>

struct tm *localtime(const time_t *timer);
CX struct tm *localtime_r(const time_t *restrict timer,
    struct tm *restrict result);
```

DESCRIPTION

CX For `localtime()`: The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The `localtime()` function shall convert the time in seconds since the Epoch pointed to by `timer` into a broken-down time, expressed as a local time. The function corrects for the timezone and any seasonal time adjustments. CX Local timezone information is used as though `localtime()` calls `tzset()`.

The relationship between a time in seconds since the Epoch used as an argument to `localtime()` and the **tm** structure (defined in the `<time.h>` header) is that the result shall be as specified in the expression given in the definition of seconds since the Epoch (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.14, Seconds Since the Epoch) corrected for timezone and any seasonal time adjustments, where the names in the structure and in the expression correspond.

The same relationship shall apply for `localtime_r()`.

The `localtime()` function need not be thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

The `asctime()`, `ctime()`, `gmtime()`, and `localtime()` functions shall return values in one of two static objects: a broken-down time structure and an array of type **char**. Execution of any of the functions may overwrite the information returned in either of these objects by any of the other functions.

The `localtime_r()` function shall convert the time in seconds since the Epoch pointed to by `timer` into a broken-down time stored in the structure to which `result` points. The `localtime_r()` function shall also return a pointer to that same structure.

Unlike `localtime()`, the reentrant version is not required to set `tzname`.

If the reentrant version does not set `tzname`, it shall not set `daylight` and shall not set `timezone`.

RETURN VALUE

CX Upon successful completion, the `localtime()` function shall return a pointer to the broken-down time structure. If an error is detected, `localtime()` shall return a null pointer and set `errno` to indicate the error.

Upon successful completion, `localtime_r()` shall return a pointer to the structure pointed to by the argument `result`. If an error is detected, `localtime_r()` s

24468 **ERRORS**24469 CX The `localtime()` and `localtime_r()` functions shall fail if:24470 CX **[EOVERFLOW]** The result cannot be represented.24471 **EXAMPLES**24472 **Getting the Local Date and Time**

24473 The following example uses the `time()` function to calculate the time elapsed, in seconds, since
 24474 January 1, 1970 0:00 UTC (the Epoch), `localtime()` to convert that value to a broken-down time,
 24475 and `asctime()` to convert the broken-down time values into a printable string.

```
24476 #include <stdio.h>
24477 #include <time.h>
24478
24479 int main(void)
24480 {
24481     time_t result;
24482     result = time(NULL);
24483     printf("%s%ju secs since the Epoch\n",
24484           asctime(localtime(&result)),
24485           (uintmax_t)result);
24486     return(0);
24487 }
```

24487 This example writes the current time to `stdout` in a form like this:

```
24488 Wed Jun 26 10:32:15 1996
24489 835810335 secs since the Epoch
```

24490 **Getting the Modification Time for a File**

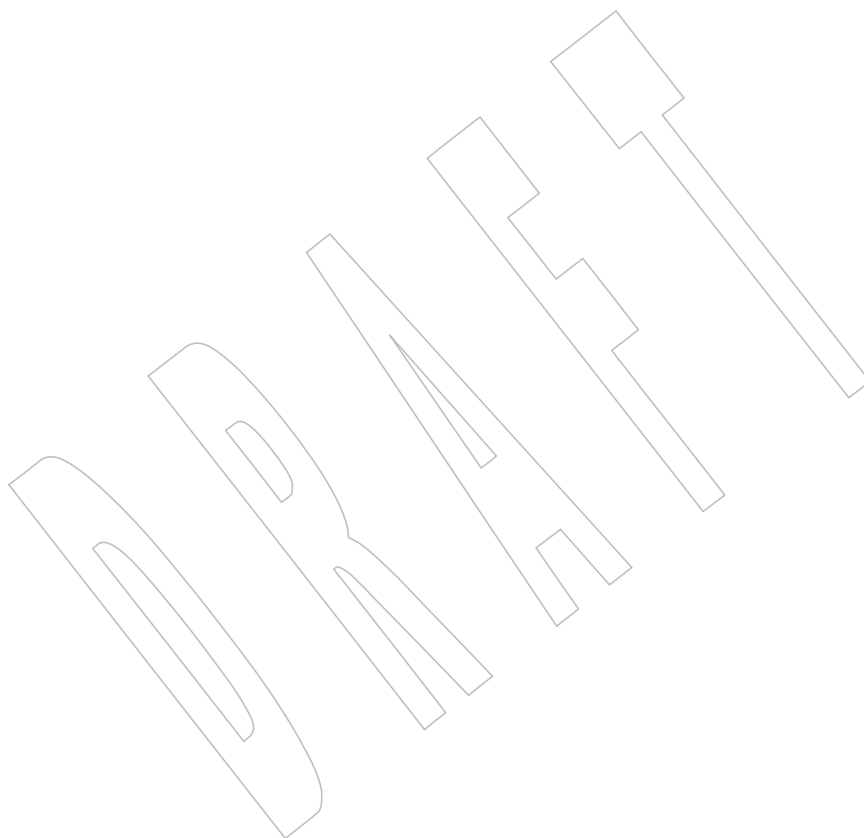
24491 The following example gets the modification time for a file. The `localtime()` function converts the
 24492 **time_t** value of the last modification date, obtained by a previous call to `stat()`, into a **tm**
 24493 structure that contains the year, month, day, and so on.

```
24494 #include <time.h>
24495 ...
24496 struct stat statbuf;
24497 ...
24498 tm = localtime(&statbuf.st_mtime);
24499 ...
```

24500 **Timing an Event**

24501 The following example gets the current time, converts it to a string using `localtime()` and
 24502 `asctime()`, and prints it to standard output using `fputs()`. It then prints the number of minutes to
 24503 an event being timed.

```
24504 #include <time.h>
24505 #include <stdio.h>
24506 ...
24507 time_t now;
24508 int minutes_to_event;
24509 ...
24510 time(&now);
24511 printf("The time is ");
```

24551 **NAME**
 24552 lockf — record locking on files

24553 **SYNOPSIS**

```
24554 XSI #include <unistd.h>
24555 int lockf(int fildes, int function, off_t size);
```

24556 **DESCRIPTION**

24557 The *lockf()* function shall lock sections of a file with advisory-mode locks. Calls to *lockf()* from
 24558 threads in other processes which attempt to lock the locked file section shall either return an
 24559 error value or block until the section becomes unlocked. All the locks for a process are removed
 24560 when the process terminates. Record locking with *lockf()* shall be supported for regular files and
 24561 may be supported for other files.

24562 The *fildes* argument is an open file descriptor. To establish a lock with this function, the file
 24563 descriptor shall be opened with write-only permission (O_WRONLY) or with read/write
 24564 permission (O_RDWR).

24565 The *function* argument is a control value which specifies the action to be taken. The permissible
 24566 values for *function* are defined in **<unistd.h>** as follows:

Function	Description
F_ULOCK	Unlock locked sections.
F_LOCK	Lock a section for exclusive use.
F_TLOCK	Test and lock a section for exclusive use.
F_TEST	Test a section for locks by other processes.

24572 F_TEST shall detect if a lock by another process is present on the specified section.

24573 F_LOCK and F_TLOCK shall both lock a section of a file if the section is available.

24574 F_ULOCK shall remove locks from a section of the file.

24575 The *size* argument is the number of contiguous bytes to be locked or unlocked. The section to be
 24576 locked or unlocked starts at the current offset in the file and extends forward for a positive size
 24577 or backward for a negative size (the preceding bytes up to but not including the current offset).
 24578 If *size* is 0, the section from the current offset through the largest possible file offset shall be
 24579 locked (that is, from the current offset through the present or any future end-of-file). An area
 24580 need not be allocated to the file to be locked because locks may exist past the end-of-file.

24581 The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained
 24582 by a previously locked section for the same process. When this occurs, or if adjacent locked
 24583 sections would occur, the sections shall be combined into a single locked section. If the request
 24584 would cause the number of locks to exceed a system-imposed limit, the request shall fail.

24585 F_LOCK and F_TLOCK requests differ only by the action taken if the section is not available.
 24586 F_LOCK shall block the calling thread until the section is available. F_TLOCK shall cause the
 24587 function to fail if the section is already locked by another process.

24588 File locks shall be released on first close by the locking process of any file descriptor for the file.

24589 F_ULOCK requests may release (wholly or in part) one or more locked sections controlled by the
 24590 process. Locked sections shall be unlocked starting at the current file offset through *size* bytes or
 24591 to the end-of-file if *size* is (off_t)0. When all of a locked section is not released (that is, when the
 24592 beginning or end of the area to be unlocked falls within a locked section), the remaining portions
 24593 of that section shall remain locked by the process. Releasing the center portion of a locked

24594 section shall cause the remaining locked beginning and end portions to become two separate
 24595 locked sections. If the request would cause the number of locks in the system to exceed a system-
 24596 imposed limit, the request shall fail.

24597 A potential for deadlock occurs if the threads of a process controlling a locked section are
 24598 blocked by accessing a locked section of another process. If the system detects that deadlock
 24599 would occur, *lockf()* shall fail with an [EDEADLK] error.

24600 The interaction between *fcntl()* and *lockf()* locks is unspecified.

24601 Blocking on a section shall be interrupted by any signal.

24602 An F_ULOCK request in which *size* is non-zero and the offset of the last byte of the requested
 24603 section is the maximum value for an object of type **off_t**, when the process has an existing lock
 24604 in which *size* is 0 and which includes the last byte of the requested section, shall be treated as a
 24605 request to unlock from the start of the requested section with a size equal to 0. Otherwise, an
 24606 F_ULOCK request shall attempt to unlock only the requested section.

24607 Attempting to lock a section of a file that is associated with a buffered stream produces
 24608 unspecified results.

RETURN VALUE

24609 Upon successful completion, *lockf()* shall return 0. Otherwise, it shall return -1, set *errno* to
 24610 indicate an error, and existing locks shall not be changed.

ERRORS

24612 The *lockf()* function shall fail if:

24613 [EBADF] The *fildev* argument is not a valid open file descriptor; or *function* is F_LOCK
 24614 or F_TLOCK and *fildev* is not a valid file descriptor open for writing.

24615 [EACCES] or [EAGAIN] The *function* argument is F_TLOCK or F_TEST and the section is already
 24616 locked by another process.

24617 [EDEADLK] The *function* argument is F_LOCK and a deadlock is detected.

24618 [EINTR] A signal was caught during execution of the function.

24619 [EINVAL] The *function* argument is not one of F_LOCK, F_TLOCK, F_TEST, or
 24620 F_ULOCK; or *size* plus the current file offset is less than 0.

24621 [EOVERFLOW] The offset of the first, or if *size* is not 0 then the last, byte in the requested
 24622 section cannot be represented correctly in an object of type **off_t**.

24623 The *lockf()* function may fail if:

24624 [EAGAIN] The *function* argument is F_LOCK or F_TLOCK and the file is mapped with
 24625 *mmap()*.

24626 [EDEADLK] or [ENOLCK] The *function* argument is F_LOCK, F_TLOCK, or F_ULOCK, and the request
 24627 would cause the number of locks to exceed a system-imposed limit.

24628 [EOPNOTSUPP] or [EINVAL] The implementation does not support the locking of files of the type indicated
 24629 by the *fildev* argument.

24634

EXAMPLES

24635

Locking a Portion of a File

24636

In the following example, a file named `/home/cnd/mod1` is being modified. Other processes that use locking are prevented from changing it during this process. Only the first 10 000 bytes are locked, and the lock call fails if another process has any part of this area locked already.

24637

24638

```
#include <fcntl.h>
#include <unistd.h>

int fildes;
int status;
...
fildes = open("/home/cnd/mod1", O_RDWR);
status = lockf(fildes, F_TLOCK, (off_t)10000);
```

24641

24642

24643

24644

24645

24646

APPLICATION USAGE

24647

Record-locking should not be used in combination with the `fopen()`, `fread()`, `fwrite()`, and other `stdio` functions. Instead, the more primitive, non-buffered functions (such as `open()`) should be used. Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The `stdio` functions are the most common source of unexpected buffering.

24648

24649

24650

24651

24652

The `alarm()` function may be used to provide a timeout facility in applications requiring it.

24653

RATIONALE

24654

None.

24655

FUTURE DIRECTIONS

24656

None.

24657

SEE ALSO

24658

`alarm()`, `chmod()`, `close()`, `creat()`, `fcntl()`, `fopen()`, `mmap()`, `open()`, `read()`, `write()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<unistd.h>`

24659

24660

CHANGE HISTORY

24661

First released in Issue 4, Version 2.

24662

Issue 5

24663

Moved from X/OPEN UNIX extension to BASE.

24664

Large File Summit extensions are added. In particular, the description of [EINVAL] is clarified and moved from optional to mandatory status.

24665

24666

A note is added to the DESCRIPTION indicating the effects of attempting to lock a section of a file that is associated with a buffered stream.

24667

24668

Issue 6

24669

The normative text is updated to avoid use of the term “must” for application requirements.

24670

Issue 7

24671

Austin Group Interpretation 1003.1-2001 #054 is applied, updating the DESCRIPTION.

24672 **NAME**
 24673 `log`, `logf`, `logl` — natural logarithm function

24674 **SYNOPSIS**
 24675 `#include <math.h>`
 24676 `double log(double x);`
 24677 `float logf(float x);`
 24678 `long double logl(long double x);`

DESCRIPTION

24680 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 24681 conflict between the requirements described here and the ISO C standard is unintentional. This
 24682 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

24683 These functions shall compute the natural logarithm of their argument x , $\log_e(x)$.

24684 An application wishing to check for error situations should set *errno* to zero and call
 24685 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 24686 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 24687 zero, an error has occurred.

RETURN VALUE

24688 Upon successful completion, these functions shall return the natural logarithm of x .

24690 If x is ± 0 , a pole error shall occur and *log*(x), *logf*(x), and *logl*(x) shall return `-HUGE_VAL`,
 24691 `-HUGE_VALF`, and `-HUGE_VALL`, respectively.

24692 MX For finite values of x that are less than 0, or if x is `-Inf`, a domain error shall occur, and either a
 24693 NaN (if supported), or an implementation-defined value shall be returned.

24694 MX If x is NaN, a NaN shall be returned.

24695 If x is 1, `+0` shall be returned.

24696 If x is `+Inf`, x shall be returned.

ERRORS

24698 These functions shall fail if:

24699 MX Domain Error The finite value of x is negative, or x is `-Inf`.

24700 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 24701 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 24702 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 24703 shall be raised.

24704 Pole Error The value of x is zero.

24705 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 24706 then *errno* shall be set to [ERANGE]. If the integer expression
 24707 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by-zero
 24708 floating-point exception shall be raised.

24709
24710
24711
24712
24713
24714
24715
24716
24717
24718
24719
24720
24721
24722
24723
24724
24725
24726
24727
24728
24729
24730
24731
24732
24733

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exp(), *feclearexcept()*, *fetestexcept()*, *isnan()*, *log10()*, *log1p()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The normative text is updated to avoid use of the term “must” for application requirements.

The *logf()* and *logl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

24734 **NAME**
 24735 `log10, log10f, log10l` — base 10 logarithm function

24736 **SYNOPSIS**
 24737 `#include <math.h>`
 24738 `double log10(double x);`
 24739 `float log10f(float x);`
 24740 `long double log10l(long double x);`

24741 **DESCRIPTION**

24742 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 24743 conflict between the requirements described here and the ISO C standard is unintentional. This
 24744 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

24745 These functions shall compute the base 10 logarithm of their argument x , $\log_{10}(x)$.

24746 An application wishing to check for error situations should set *errno* to zero and call
 24747 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 24748 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 24749 zero, an error has occurred.

24750 **RETURN VALUE**

24751 Upon successful completion, these functions shall return the base 10 logarithm of x .

24752 If x is ± 0 , a pole error shall occur and *log10()*, *log10f()*, and *log10l()* shall return `-HUGE_VAL`,
 24753 `-HUGE_VALF`, and `-HUGE_VALL`, respectively.

24754 MX For finite values of x that are less than 0, or if x is `-Inf`, a domain error shall occur, and either a
 24755 NaN (if supported), or an implementation-defined value shall be returned.

24756 MX If x is NaN, a NaN shall be returned.

24757 If x is 1, `+0` shall be returned.

24758 If x is `+Inf`, `+Inf` shall be returned.

24759 **ERRORS**

24760 These functions shall fail if:

24761 MX Domain Error The finite value of x is negative, or x is `-Inf`.
 24762 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 24763 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 24764 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 24765 shall be raised.

24766 Pole Error The value of x is zero.

24767 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 24768 then *errno* shall be set to [ERANGE]. If the integer expression
 24769 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by-zero
 24770 floating-point exception shall be raised.

24771
24772
24773
24774
24775
24776
24777
24778
24779
24780
24781
24782
24783
24784
24785
24786
24787
24788
24789
24790
24791
24792
24793
24794
24795
24796

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept(), *fetestexcept()*, *isnan()*, *log()*, *pow()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The normative text is updated to avoid use of the term “must” for application requirements.

The *log10f()* and *log10l()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

24797 **NAME**
 24798 `log1p`, `log1pf`, `log1pl` — compute a natural logarithm

24799 **SYNOPSIS**
 24800 `#include <math.h>`
 24801 `double log1p(double x);`
 24802 `float log1pf(float x);`
 24803 `long double log1pl(long double x);`

24804 **DESCRIPTION**
 24805 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 24806 conflict between the requirements described here and the ISO C standard is unintentional. This
 24807 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

24808 These functions shall compute $\log_e(1.0 + x)$.
 24809 An application wishing to check for error situations should set *errno* to zero and call
 24810 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 24811 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 24812 zero, an error has occurred.

24813 **RETURN VALUE**
 24814 Upon successful completion, these functions shall return the natural logarithm of $1.0 + x$.
 24815 If x is -1 , a pole error shall occur and *log1p()*, *log1pf()*, and *log1pl()* shall return `-HUGE_VAL`,
 24816 `-HUGE_VALF`, and `-HUGE_VALL`, respectively.

24817 MX For finite values of x that are less than -1 , or if x is $-\text{Inf}$, a domain error shall occur, and either a
 24818 NaN (if supported), or an implementation-defined value shall be returned.

24819 MX If x is NaN, a NaN shall be returned.

24820 If x is ± 0 , or $+\text{Inf}$, x shall be returned.

24821 If x is subnormal, a range error may occur and x should be returned.

24822 **ERRORS**
 24823 These functions shall fail if:

24824 MX **Domain Error** The finite value of x is less than -1 , or x is $-\text{Inf}$.
 24825 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 24826 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 24827 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 24828 shall be raised.

24829 **Pole Error** The value of x is -1 .
 24830 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 24831 then *errno* shall be set to [ERANGE]. If the integer expression
 24832 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by-zero
 24833 floating-point exception shall be raised.

24834 These functions may fail if:

24835 MX **Range Error** The value of x is subnormal.
 24836 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 24837 then *errno* shall be set to [ERANGE]. If the integer expression
 24838 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow

24839 floating-point exception shall be raised.

24840 **EXAMPLES**

24841 None.

24842 **APPLICATION USAGE**

24843 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
24844 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

24845 **RATIONALE**

24846 None.

24847 **FUTURE DIRECTIONS**

24848 None.

24849 **SEE ALSO**

24850 *feclearexcept()*, *fetestexcept()*, *log()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section
24851 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

24852 **CHANGE HISTORY**

24853 First released in Issue 4, Version 2.

24854 **Issue 5**

24855 Moved from X/OPEN UNIX extension to BASE.

24856 **Issue 6**

24857 The normative text is updated to avoid use of the term “must” for application requirements.

24858 The *log1p()* function is no longer marked as an extension.

24859 The *log1pf()* and *log1pl()* functions are added for alignment with the ISO/IEC 9899:1999
24860 standard.

24861 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
24862 revised to align with the ISO/IEC 9899:1999 standard.

24863 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
24864 marked.

24865 **NAME**
 24866 `log2, log2f, log2l` — compute base 2 logarithm functions

24867 **SYNOPSIS**
 24868 `#include <math.h>`
 24869 `double log2(double x);`
 24870 `float log2f(float x);`
 24871 `long double log2l(long double x);`

24872 **DESCRIPTION**
 24873 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 24874 conflict between the requirements described here and the ISO C standard is unintentional. This
 24875 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

24876 These functions shall compute the base 2 logarithm of their argument x , $\log_2(x)$.
 24877 An application wishing to check for error situations should set *errno* to zero and call
 24878 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 24879 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 24880 zero, an error has occurred.

24881 **RETURN VALUE**
 24882 Upon successful completion, these functions shall return the base 2 logarithm of x .
 24883 If x is ± 0 , a pole error shall occur and *log2()*, *log2f()*, and *log2l()* shall return `-HUGE_VAL`,
 24884 `-HUGE_VALF`, and `-HUGE_VALL`, respectively.

24885 MX For finite values of x that are less than 0, or if x is `-Inf`, a domain error shall occur, and either a
 24886 NaN (if supported), or an implementation-defined value shall be returned.

24887 MX If x is NaN, a NaN shall be returned.

24888 If x is 1, `+0` shall be returned.

24889 If x is `+Inf`, x shall be returned.

24890 **ERRORS**
 24891 These functions shall fail if:

24892 MX Domain Error The finite value of x is less than zero, or x is `-Inf`.
 24893 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 24894 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 24895 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 24896 shall be raised.

24897 Pole Error The value of x is zero.
 24898 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 24899 then *errno* shall be set to [ERANGE]. If the integer expression
 24900 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by-zero
 24901 floating-point exception shall be raised.

24902

EXAMPLES

24903

None.

24904

APPLICATION USAGE

24905

On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

24906

24907

RATIONALE

24908

None.

24909

FUTURE DIRECTIONS

24910

None.

24911

SEE ALSO

24912

feclearexcept(), *fetestexcept()*, *log()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

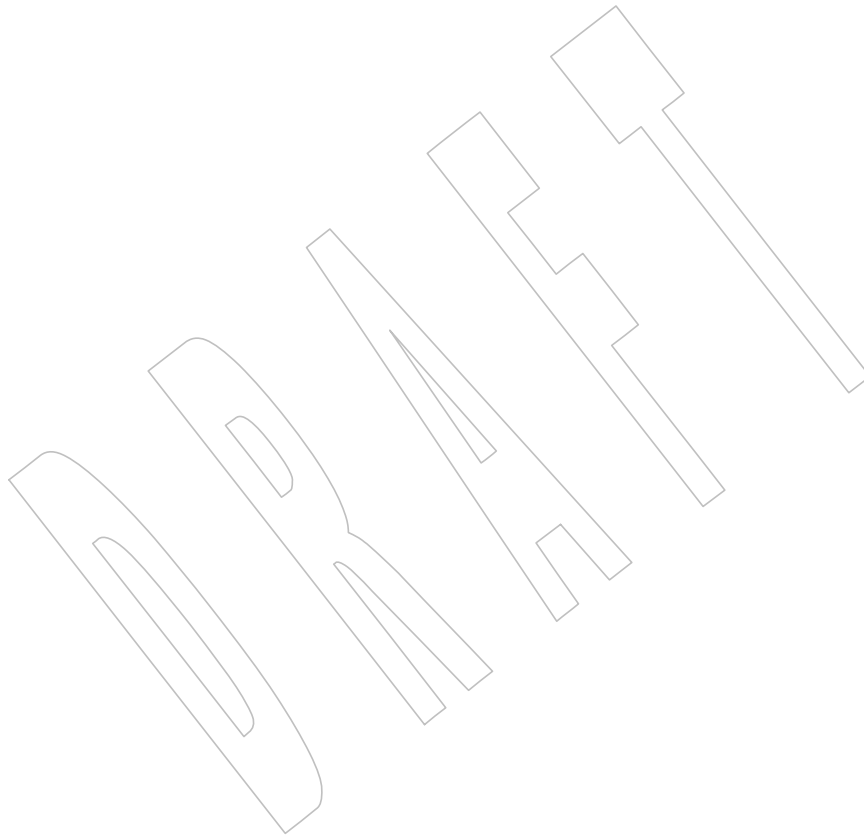
24913

24914

CHANGE HISTORY

24915

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.



24916 **NAME**
 24917 logb, logbf, logbl — radix-independent exponent

24918 **SYNOPSIS**
 24919 #include <math.h>
 24920 double logb(double x);
 24921 float logbf(float x);
 24922 long double logbl(long double x);

24923 DESCRIPTION

24924 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 24925 conflict between the requirements described here and the ISO C standard is unintentional. This
 24926 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

24927 These functions shall compute the exponent of x , which is the integral part of $\log_r |x|$, as a
 24928 signed floating-point value, for non-zero x , where r is the radix of the machine's floating-point
 24929 arithmetic, which is the value of FLT_RADIX defined in the <float.h> header.

24930 If x is subnormal it is treated as though it were normalized; thus for finite positive x :

24931
$$1 \leq x * FLT_RADIX^{-\log b(x)} < FLT_RADIX$$

24932 An application wishing to check for error situations should set *errno* to zero and call
 24933 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 24934 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 24935 zero, an error has occurred.

24936 RETURN VALUE

24937 Upon successful completion, these functions shall return the exponent of x .

24938 If x is ± 0 , *logb*(x), *logbf*(x), and *logbl*(x) shall return `-HUGE_VAL`, `-HUGE_VALF`, and
 24939 MX `-HUGE_VALL`, respectively. On systems that support the IEC 60559 Floating-Point option, a
 24940 pole error shall occur;

24941 CX otherwise, a pole error may occur.

24942 MX If x is NaN, a NaN shall be returned.

24943 MX If x is $\pm\text{Inf}$, $+\text{Inf}$ shall be returned.

24944 ERRORS

24945 These functions shall fail if:

24946 MX **Pole Error** The value of x is ± 0 .

24947 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 24948 then *errno* shall be set to [ERANGE]. If the integer expression
 24949 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by-zero
 24950 floating-point exception shall be raised.

24951 These functions may fail if:

24952 **Pole Error** The value of x is 0.

24953 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 24954 then *errno* shall be set to [ERANGE]. If the integer expression
 24955 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by-zero
 24956 floating-point exception shall be raised.

24957

EXAMPLES

24958

None.

24959

APPLICATION USAGE

24960

On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

24961

24962

RATIONALE

24963

None.

24964

FUTURE DIRECTIONS

24965

None.

24966

SEE ALSO

24967

feclearexcept(), *fetestexcept()*, *ilogb()*, *scalbln()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, `<float.h>`, `<math.h>`

24968

24969

24970

CHANGE HISTORY

24971

First released in Issue 4, Version 2.

24972

Issue 5

24973

Moved from X/OPEN UNIX extension to BASE.

24974

Issue 6

24975

The *logb()* function is no longer marked as an extension.

24976

The *logbf()* and *logbl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

24977

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

24978

24979

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

24980

24981

Issue 7

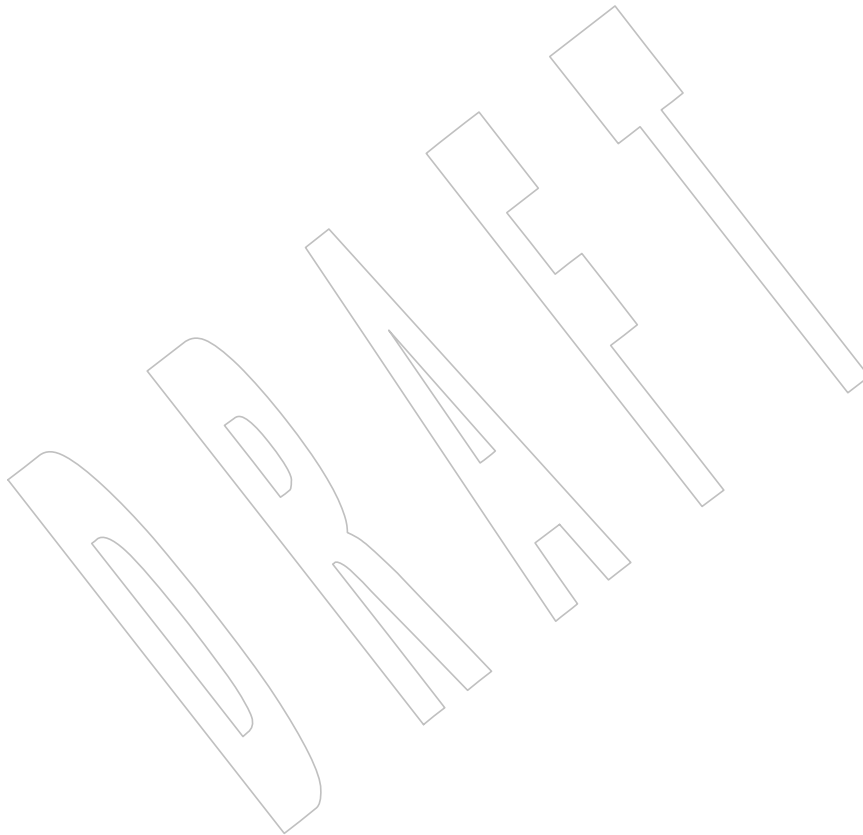
24982

ISO/IEC 9899:1999 standard, Technical Corrigendum 2 #50 (SD5-XSH-ERN-76) is applied.

24983 **NAME**
24984 `logf, logl` — natural logarithm function

24985 **SYNOPSIS**
24986 `#include <math.h>`
24987 `float logf(float x);`
24988 `long double logl(long double x);`

24989 **DESCRIPTION**
24990 Refer to *log()*.



24991 **NAME**
 24992 `longjmp` — non-local goto

24993 **SYNOPSIS**
 24994 `#include <setjmp.h>`

24995 `void longjmp(jmp_buf env, int val);`

24996 DESCRIPTION

24997 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 24998 conflict between the requirements described here and the ISO C standard is unintentional. This
 24999 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25000 The `longjmp()` function shall restore the environment saved by the most recent invocation of
 25001 `setjmp()` in the same thread, with the corresponding `jmp_buf` argument. If there is no such
 25002 invocation, or if the function containing the invocation of `setjmp()` has terminated execution in
 25003 the interim, or if the invocation of `setjmp()` was within the scope of an identifier with variably
 25004 CX modified type and execution has left that scope in the interim, the behavior is undefined. It is
 25005 unspecified whether `longjmp()` restores the signal mask, leaves the signal mask unchanged, or
 25006 restores it to its value at the time `setjmp()` was called.

25007 All accessible objects have values, and all other components of the abstract machine have state
 25008 (for example, floating-point status flags and open files), as of the time `longjmp()` was called,
 25009 except that the values of objects of automatic storage duration are unspecified if they meet all
 25010 the following conditions:

- 25011 • They are local to the function containing the corresponding `setjmp()` invocation.
- 25012 • They do not have volatile-qualified type.
- 25013 • They are changed between the `setjmp()` invocation and `longjmp()` call.

25014 CX As it bypasses the usual function call and return mechanisms, `longjmp()` shall execute correctly
 25015 in contexts of interrupts, signals, and any of their associated functions. However, if `longjmp()` is
 25016 invoked from a nested signal handler (that is, from a function invoked as a result of a signal
 25017 raised during the handling of another signal), the behavior is undefined.

25018 The effect of a call to `longjmp()` where initialization of the `jmp_buf` structure was not performed
 25019 in the calling thread is undefined.

25020 RETURN VALUE

25021 After `longjmp()` is completed, program execution continues as if the corresponding invocation of
 25022 `setjmp()` had just returned the value specified by `val`. The `longjmp()` function shall not cause
 25023 `setjmp()` to return 0; if `val` is 0, `setjmp()` shall return 1.

25024 ERRORS

25025 No errors are defined.

25026 EXAMPLES

25027 None.

25028 APPLICATION USAGE

25029 Applications whose behavior depends on the value of the signal mask should not use `longjmp()`
 25030 and `setjmp()`, since their effect on the signal mask is unspecified, but should instead use the
 25031 `siglongjmp()` and `sigsetjmp()` functions (which can save and restore the signal mask under
 25032 application control).

longjmp()

25033

RATIONALE

25034

None.

25035

FUTURE DIRECTIONS

25036

None.

25037

SEE ALSO

25038

setjmp(), *sigaction()*, *siglongjmp()*, *sigsetjmp()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**setjmp.h**>

25039

25040

CHANGE HISTORY

25041

First released in Issue 1. Derived from Issue 1 of the SVID.

25042

Issue 5

25043

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

25044

Issue 6

25045

Extensions beyond the ISO C standard are marked.

25046

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

25047

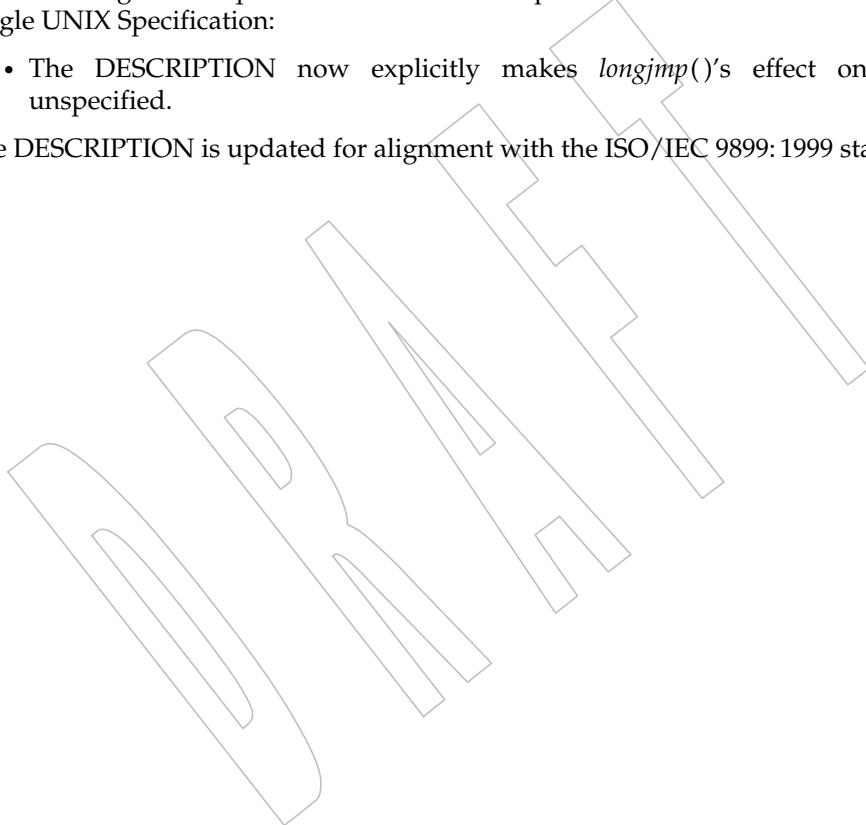
25048

- The DESCRIPTION now explicitly makes *longjmp()*'s effect on the signal mask unspecified.

25049

25050

The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.



25051 **NAME**
25052 lrand48 — generate uniformly distributed pseudo-random non-negative long integers

25053 **SYNOPSIS**

25054 XSI #include <stdlib.h>
25055 long lrand48(void);

25056 **DESCRIPTION**

25057 Refer to *drand48()*.

25058 **NAME**25059 `lrint, lrintf, lrintl` — round to nearest integer value using current rounding direction25060 **SYNOPSIS**

```
25061 #include <math.h>
25062
25062 long lrint(double x);
25063 long lrintf(float x);
25064 long lrintl(long double x);
```

25065 **DESCRIPTION**

25066 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 25067 conflict between the requirements described here and the ISO C standard is unintentional. This
 25068 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25069 These functions shall round their argument to the nearest integer value, rounding according to
 25070 the current rounding direction.

25071 An application wishing to check for error situations should set *errno* to zero and call
 25072 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 25073 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 25074 zero, an error has occurred.

25075 **RETURN VALUE**

25076 Upon successful completion, these functions shall return the rounded integer value.

25077 MX If *x* is NaN, a domain error shall occur and an unspecified value is returned.25078 If *x* is +Inf, a domain error shall occur and an unspecified value is returned.25079 If *x* is -Inf, a domain error shall occur and an unspecified value is returned.

25080 If the correct value is positive and too large to represent as a **long**, an unspecified value shall be
 25081 MX returned. On systems that support the IEC 60559 Floating-Point option, a domain error shall
 25082 occur;

25083 CX otherwise, a **domain** error may occur.

25084 If the correct value is negative and too large to represent as a **long**, an unspecified value shall be
 25085 MX returned. On systems that support the IEC 60559 Floating-Point option, a domain error shall
 25086 occur;

25087 CX otherwise, a **domain** error may occur.25088 **ERRORS**

25089 These functions shall fail if:

25090 MX **Domain Error** The *x* argument is NaN or \pm Inf, or the correct value is not representable as an
 25091 integer.

25092 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 25093 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 25094 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 25095 shall be raised.

25096 These functions may fail if:

25097 **Domain Error** The correct value is not representable as an integer.

25098 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 25099 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 25100 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception

25101 shall be raised.

25102 **EXAMPLES**

25103 None.

25104 **APPLICATION USAGE**

25105 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
25106 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

25107 **RATIONALE**

25108 These functions provide floating-to-integer conversions. They round according to the current
25109 rounding direction. If the rounded value is outside the range of the return type, the numeric
25110 result is unspecified and the invalid floating-point exception is raised. When they raise no other
25111 floating-point exception and the result differs from the argument, they raise the inexact floating-
25112 point exception.

25113 **FUTURE DIRECTIONS**

25114 None.

25115 **SEE ALSO**

25116 *feclearexcept()*, *fetestexcept()*, *llrint()*, the Base Definitions volume of IEEE Std 1003.1-200x,
25117 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

25118 **CHANGE HISTORY**

25119 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

25120 **Issue 7**

25121 ISO/IEC 9899:1999 standard, Technical Corrigendum 2 #53 (SD5-XSH-ERN-77) is applied.

25122 **NAME**25123 `lround, lroundf, lroundl` — round to nearest integer value25124 **SYNOPSIS**

```
25125 #include <math.h>
25126
25126 long lround(double x);
25127 long lroundf(float x);
25128 long lroundl(long double x);
```

25129 **DESCRIPTION**

25130 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 25131 conflict between the requirements described here and the ISO C standard is unintentional. This
 25132 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25133 These functions shall round their argument to the nearest integer value, rounding halfway cases
 25134 away from zero, regardless of the current rounding direction.

25135 An application wishing to check for error situations should set *errno* to zero and call
 25136 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 25137 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 25138 zero, an error has occurred.

25139 **RETURN VALUE**

25140 Upon successful completion, these functions shall return the rounded integer value.

25141 MX If *x* is NaN, a domain error shall occur and an unspecified value is returned.

25142 If *x* is +Inf, a domain error shall occur and an unspecified value is returned.

25143 If *x* is -Inf, a domain error shall occur and an unspecified value is returned.

25144 If the correct value is positive and too large to represent as a **long**, an unspecified value shall be
 25145 MX returned. On systems that support the IEC 60559 Floating-Point option, a domain shall occur;
 25146 CX otherwise, a domain error may occur.

25147 If the correct value is negative and too large to represent as a **long**, an unspecified value shall be
 25148 MX returned. On systems that support the IEC 60559 Floating-Point option, a domain shall occur;
 25149 CX otherwise, a domain error may occur.

25150 **ERRORS**

25151 These functions shall fail if:

25152 MX **Domain Error** The *x* argument is NaN or \pm Inf, or the correct value is not representable as an
 25153 integer.

25154 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 25155 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 25156 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 25157 shall be raised.

25158 These functions may fail if:

25159 **Domain Error** The correct value is not representable as an integer.

25160 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 25161 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 25162 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 25163 shall be raised.

25164

EXAMPLES

25165

None.

25166

APPLICATION USAGE

25167

On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

25168

25169

RATIONALE

25170

These functions differ from the *lrint()* functions in the default rounding direction, with the *lround()* functions rounding halfway cases away from zero and needing not to raise the inexact floating-point exception for non-integer arguments that round to within the range of the return type.

25171

25172

25173

25174

FUTURE DIRECTIONS

25175

None.

25176

SEE ALSO

25177

feclearexcept(), *fetestexcept()*, *llround()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

25178

25179

CHANGE HISTORY

25180

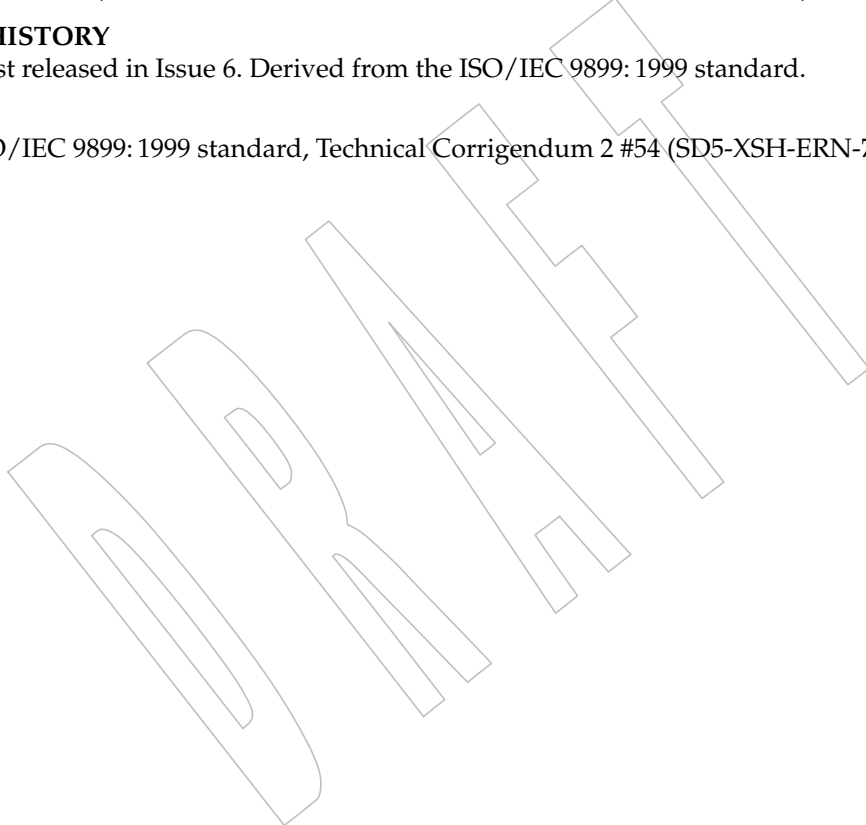
First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

25181

Issue 7

25182

ISO/IEC 9899:1999 standard, Technical Corrigendum 2 #54 (SD5-XSH-ERN-78) is applied.



25183 **NAME**25184 `lsearch, lfind` — linear search and update25185 **SYNOPSIS**

```

25186 XSI #include <search.h>
25187
25187 void *lsearch(const void *key, void *base, size_t *nel, size_t width,
25188             int (*compar)(const void *, const void *));
25189 void *lfind(const void *key, const void *base, size_t *nel,
25190           size_t width, int (*compar)(const void *, const void *));

```

25191 **DESCRIPTION**

25192 The `lsearch()` function shall linearly search the table and return a pointer into the table for the
 25193 matching entry. If the entry does not occur, it shall be added at the end of the table. The `key`
 25194 argument points to the entry to be sought in the table. The `base` argument points to the first
 25195 element in the table. The `width` argument is the size of an element in bytes. The `nel` argument
 25196 points to an integer containing the current number of elements in the table. The integer to which
 25197 `nel` points shall be incremented if the entry is added to the table. The `compar` argument points to
 25198 a comparison function which the application shall supply (for example, `strcmp()`). It is called
 25199 with two arguments that point to the elements being compared. The application shall ensure
 25200 that the function returns 0 if the elements are equal, and non-zero otherwise.

25201 The `lfind()` function shall be equivalent to `lsearch()`, except that if the entry is not found, it is not
 25202 added to the table. Instead, a null pointer is returned.

25203 **RETURN VALUE**

25204 If the searched for entry is found, both `lsearch()` and `lfind()` shall return a pointer to it.
 25205 Otherwise, `lfind()` shall return a null pointer and `lsearch()` shall return a pointer to the newly
 25206 added element.

25207 Both functions shall return a null pointer in case of error.

25208 **ERRORS**

25209 No errors are defined.

25210 **EXAMPLES**25211 **Storing Strings in a Table**

25212 This fragment reads in less than or equal to `TABSIZE` strings of length less than or equal to
 25213 `ELSIZE` and stores them in a table, eliminating duplicates.

```

25214 #include <stdio.h>
25215 #include <string.h>
25216 #include <search.h>
25217
25217 #define TABSIZE 50
25218 #define ELSIZE 120
25219
25219 ...
25220     char line[ELSIZE], tab[TABSIZE][ELSIZE];
25221     size_t nel = 0;
25222     ...
25223     while (fgets(line, ELSIZE, stdin) != NULL && nel < TABSIZE)
25224         (void) lsearch(line, tab, &nel,
25225                       ELSIZE, (int (*)(const void *, const void *)) strcmp);
25226     ...

```

25227

Finding a Matching Entry

25228

The following example finds any line that reads "This is a test.".

25229

```
#include <search.h>
```

25230

```
#include <string.h>
```

25231

```
...
```

25232

```
char line[ELSIZE], tab[TABSIZE][ELSIZE];
```

25233

```
size_t nel = 0;
```

25234

```
char *findline;
```

25235

```
void *entry;
```

25236

```
findline = "This is a test.\n";
```

25237

```
entry = lfind(findline, tab, &nel, ELSIZE, (
```

25238

```
int (*)(const void *, const void *)) strcmp);
```

25239

APPLICATION USAGE

25240

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

25241

25242

Undefined results can occur if there is not enough room in the table to add a new item.

25243

RATIONALE

25244

None.

25245

FUTURE DIRECTIONS

25246

None.

25247

SEE ALSO

25248

[hcreate\(\)](#), [tsearch\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, [<search.h>](#)

25249

CHANGE HISTORY

25250

First released in Issue 1. Derived from Issue 1 of the SVID.

25251

Issue 6

25252

The normative text is updated to avoid use of the term "must" for application requirements.

25253 **NAME**25254 `lseek` — move the read/write file offset25255 **SYNOPSIS**25256 `#include <unistd.h>`25257 `off_t lseek(int fildev, off_t offset, int whence);`25258 **DESCRIPTION**25259 The `lseek()` function shall set the file offset for the open file description associated with the file
25260 descriptor *fildev*, as follows:

- 25261 • If *whence* is `SEEK_SET`, the file offset shall be set to *offset* bytes.
- 25262 • If *whence* is `SEEK_CUR`, the file offset shall be set to its current location plus *offset*.
- 25263 • If *whence* is `SEEK_END`, the file offset shall be set to the size of the file plus *offset*.

25264 The symbolic constants `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` are defined in `<unistd.h>`.25265 The behavior of `lseek()` on devices which are incapable of seeking is implementation-defined.
25266 The value of the file offset associated with such a device is undefined.25267 The `lseek()` function shall allow the file offset to be set beyond the end of the existing data in the
25268 file. If data is later written at this point, subsequent reads of data in the gap shall return bytes
25269 with the value 0 until data is actually written into the gap.25270 The `lseek()` function shall not, by itself, extend the size of a file.25271 SHM If *fildev* refers to a shared memory object, the result of the `lseek()` function is unspecified.25272 TYM If *fildev* refers to a typed memory object, the result of the `lseek()` function is unspecified.25273 **RETURN VALUE**25274 Upon successful completion, the resulting offset, as measured in bytes from the beginning of the
25275 file, shall be returned. Otherwise, `(off_t)-1` shall be returned, *errno* shall be set to indicate the
25276 error, and the file offset shall remain unchanged.25277 **ERRORS**25278 The `lseek()` function shall fail if:

- | | | |
|-------|-------------|--|
| 25279 | [EBADF] | The <i>fildev</i> argument is not an open file descriptor. |
| 25280 | [EINVAL] | The <i>whence</i> argument is not a proper value, or the resulting file offset would be negative for a regular file, block special file, or directory. |
| 25282 | [EOVERFLOW] | The resulting file offset would be a value which cannot be represented correctly in an object of type <code>off_t</code> . |
| 25284 | [ESPIPE] | The <i>fildev</i> argument is associated with a pipe, FIFO, or socket. |

25285 **EXAMPLES**

25286 None.

25287 **APPLICATION USAGE**

25288 None.

25289 **RATIONALE**25290 The ISO C standard includes the functions `fgetpos()` and `fsetpos()`, which work on very large files
25291 by use of a special positioning type.25292 Although `lseek()` may position the file offset beyond the end of the file, this function does not
25293 itself extend the size of the file. While the only function in IEEE Std 1003.1-200x that may directly

25294 extend the size of the file is *write()*, *truncate()*, and *ftruncate()*, several functions originally
 25295 derived from the ISO C standard, such as *fwrite()*, *fprintf()*, and so on, may do so (by causing
 25296 calls on *write()*).

25297 An invalid file offset that would cause [EINVAL] to be returned may be both implementation-
 25298 defined and device-dependent (for example, memory may have few invalid values). A negative
 25299 file offset may be valid for some devices in some implementations.

25300 The POSIX.1-1990 standard did not specifically prohibit *lseek()* from returning a negative offset.
 25301 Therefore, an application was required to clear *errno* prior to the call and check *errno* upon return
 25302 to determine whether a return value of (*off_t*)-1 is a negative offset or an indication of an error
 25303 condition. The standard developers did not wish to require this action on the part of a
 25304 conforming application, and chose to require that *errno* be set to [EINVAL] when the resulting
 25305 file offset would be negative for a regular file, block special file, or directory.

25306 FUTURE DIRECTIONS

25307 None.

25308 SEE ALSO

25309 *open()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<sys/types.h>`, `<unistd.h>`

25310 CHANGE HISTORY

25311 First released in Issue 1. Derived from Issue 1 of the SVID.

25312 Issue 5

25313 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

25314 Large File Summit extensions are added.

25315 Issue 6

25316 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

25317 The following new requirements on POSIX implementations derive from alignment with the
 25318 Single UNIX Specification:

- 25319 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
 25320 required for conforming implementations of previous POSIX specifications, it was not
 25321 required for UNIX applications.
- 25322 • The [Eoverflow] error condition is added. This change is to support large files.

25323 An additional [ESPIPE] error condition is added for sockets.

25324 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that
 25325 *lseek()* results are unspecified for typed memory objects.

25326

NAME

25327

lstat — get file status

25328

SYNOPSIS

25329

#include <sys/stat.h>

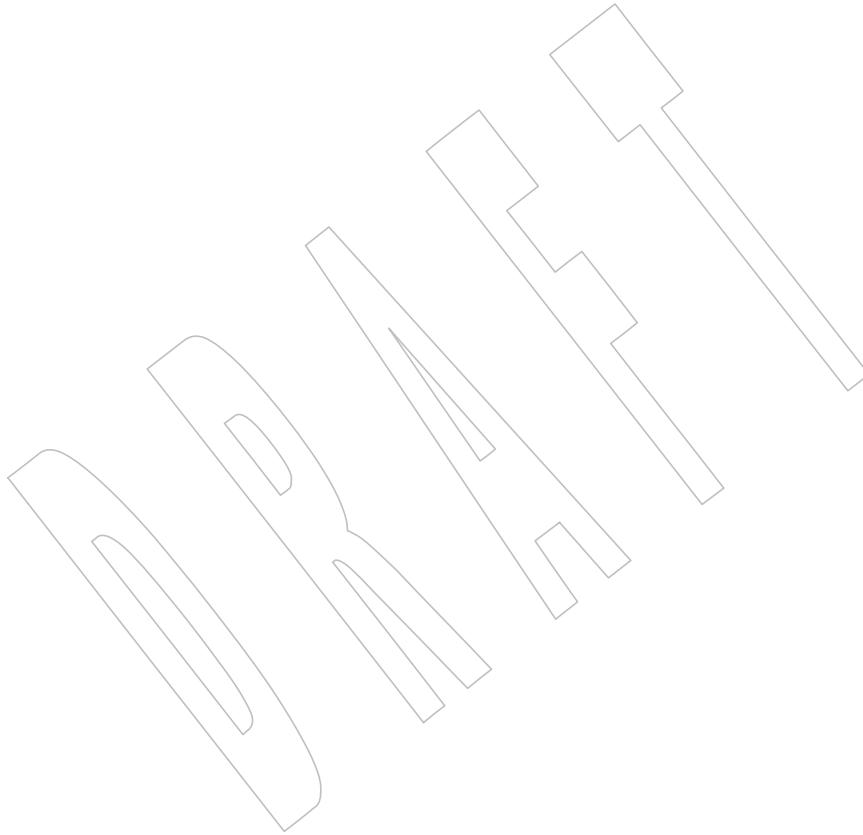
25330

int lstat(const char *restrict path, struct stat *restrict buf);

25331

DESCRIPTION

25332

Refer to *fstatat()*.

NAME

malloc — a memory allocator

SYNOPSIS

```
#include <stdlib.h>

void *malloc(size_t size);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *malloc()* function shall allocate unused space for an object whose size in bytes is specified by *size* and whose value is unspecified.

The order and contiguity of storage allocated by successive calls to *malloc()* is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned. If the size of the space requested is 0, the behavior is implementation-defined: the value returned shall be

25373

25374

25375

25376

25377

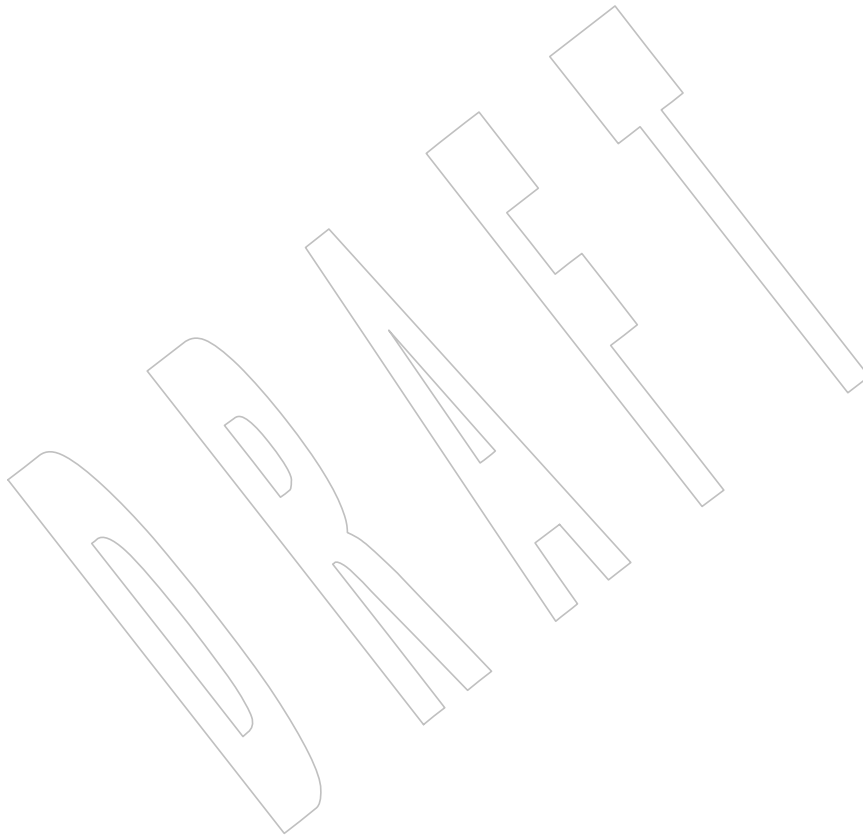
25378

Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, the requirement to set *errno* to indicate an error is added.
- The [ENOMEM] error condition is added.



25379 **NAME**25380 `mblen` — get number of bytes in a character25381 **SYNOPSIS**25382 `#include <stdlib.h>`25383 `int mblen(const char *s, size_t n);`25384 **DESCRIPTION**25385 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25386 conflict between the requirements described here and the ISO C standard is unintentional. This
25387 volume of IEEE Std 1003.1-200x defers to the ISO C standard.25388 If *s* is not a null pointer, *mblen()* shall determine the number of bytes constituting the character
25389 pointed to by *s*. Except that the shift state of *mbtowc()* is not affected, it shall be equivalent to:25390 `mbtowc((wchar_t *)0, s, n);`25391 The implementation shall behave as if no function defined in this volume of
25392 IEEE Std 1003.1-200x calls *mblen()*.25393 The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a
25394 state-dependent encoding, this function shall be placed into its initial state by a call for which its
25395 character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null
25396 pointer shall cause the internal state of the function to be altered as necessary. A call with *s* as a
25397 null pointer shall cause this function to return a non-zero value if encodings have state
25398 dependency, and 0 otherwise. If the implementation employs special bytes to change the shift
25399 state, these bytes shall not produce separate wide-character codes, but shall be grouped with an
25400 adjacent character. Changing the *LC_CTYPE* category causes the shift state of this function to be
25401 unspecified.25402 **RETURN VALUE**25403 If *s* is a null pointer, *mblen()* shall return a non-zero or 0 value, if character encodings,
25404 respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *mblen()* shall
25405 either return 0 (if *s* points to the null byte), or return the number of bytes that constitute the
25406 character (if the next *n* or fewer bytes form a valid character), or return -1 (if they do not form a
25407 valid character) and may set *errno* to indicate the error. In no case shall the value returned be
25408 greater than *n* or the value of the {MB_CUR_MAX} macro.25409 **ERRORS**25410 The *mblen()* function may fail if:

25411 XSI [EILSEQ] Invalid character sequence is detected.

25412 **EXAMPLES**

25413 None.

25414 **APPLICATION USAGE**

25415 None.

25416 **RATIONALE**

25417 None.

25418 **FUTURE DIRECTIONS**

25419 None.

mblen()*System Interfaces*

25420

SEE ALSO

25421

mbtowc(), *mbstowcs()*, *wctomb()*, *wcstombs()*, the Base Definitions volume of
IEEE Std 1003.1-200x, **<stdlib.h>**

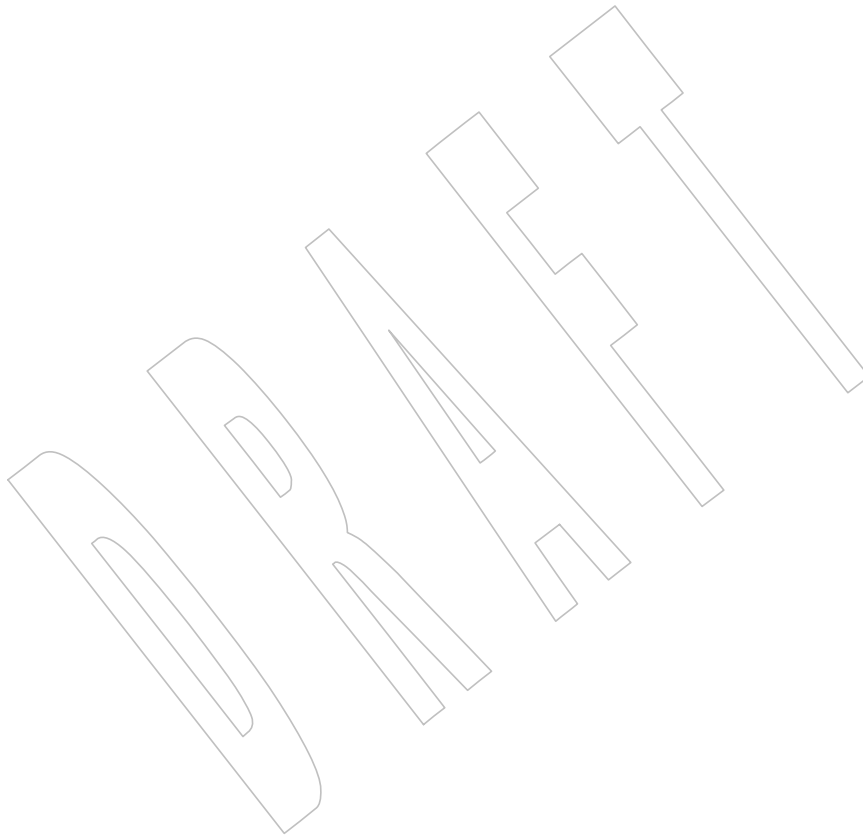
25422

25423

CHANGE HISTORY

25424

First released in Issue 4. Aligned with the ISO C standard.



NAME

mbrlen — get number of bytes in a character (restartable)

SYNOPSIS

```
#include <wchar.h>

size_t mbrlen(const char *restrict s, size_t n,
              mbstate_t *restrict ps);
```

DESCRIPTION

CX



mbrlen()

25463

EXAMPLES

25464

None.

25465

APPLICATION USAGE

25466

None.

25467

RATIONALE

25468

None.

25469

FUTURE DIRECTIONS

25470

None.

25471

SEE ALSO

25472

mbsinit(), *mbrtowc()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<wchar.h>`

25473

CHANGE HISTORY

25474

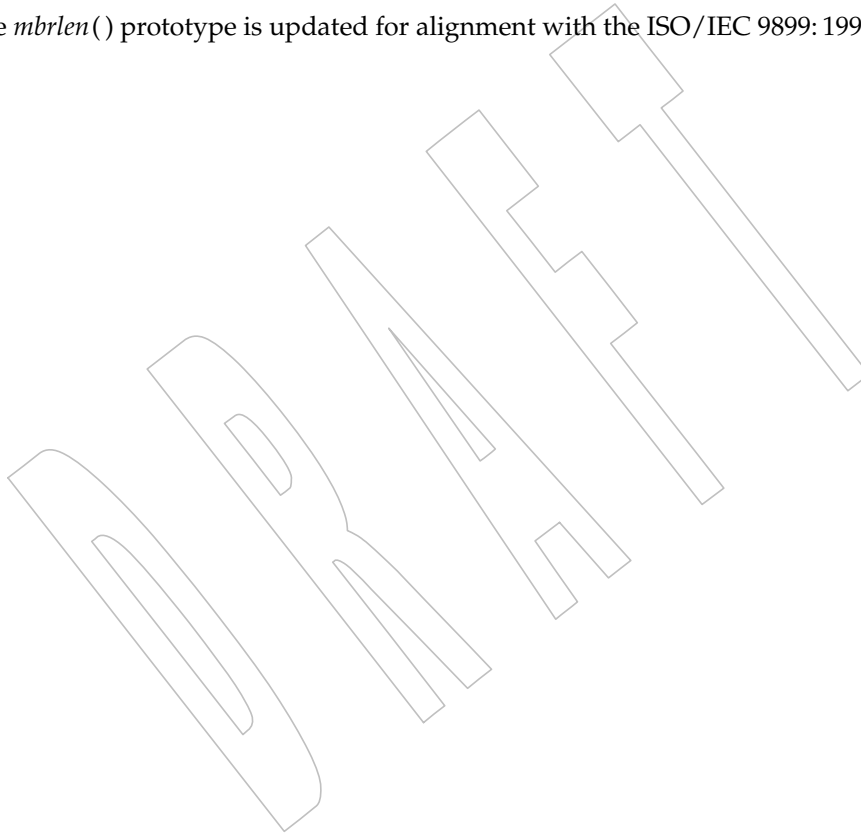
First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

25475

25476

Issue 6

25477

The *mbrlen()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

25478 **NAME**25479 `mbrtowc` — convert a character to a wide-character code (restartable)25480 **SYNOPSIS**25481 `#include <wchar.h>`25482 `size_t mbrtowc(wchar_t *restrict pwc, const char *restrict s,`
25483 `size_t n, mbstate_t *restrict ps);`25484 **DESCRIPTION**25485 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25486 conflict between the requirements described here and the ISO C standard is unintentional. This
25487 volume of IEEE Std 1003.1-200x defers to the ISO C standard.25488 If *s* is a null pointer, the `mbrtowc()` function shall be equivalent to the call:25489 `mbrtowc(NULL, "", 1, ps)`25490 In this case, the values of the arguments *pwc* and *n* are ignored.25491 If *s* is not a null pointer, the `mbrtowc()` function shall inspect at most *n* bytes beginning at the
25492 byte pointed to by *s* to determine the number of bytes needed to complete the next character
25493 (including any shift sequences). If the function determines that the next character is completed,
25494 it shall determine the value of the corresponding wide character and then, if *pwc* is not a null
25495 pointer, shall store that value in the object pointed to by *pwc*. If the corresponding wide
25496 character is the null wide character, the resulting state described shall be the initial conversion
25497 state.25498 If *ps* is a null pointer, the `mbrtowc()` function shall use its own internal `mbstate_t` object, which
25499 shall be initialized at program start-up to the initial conversion state. Otherwise, the `mbstate_t`
25500 object pointed to by *ps* shall be used to completely describe the current conversion state of the
25501 associated character sequence. The implementation shall behave as if no function defined in this
25502 volume of IEEE Std 1003.1-200x calls `mbrtowc()`.25503 The behavior of this function is affected by the `LC_CTYPE` category of the current locale.25504 **RETURN VALUE**25505 The `mbrtowc()` function shall return the first of the following that applies:25506 0 If the next *n* or fewer bytes complete the character that corresponds to the null
25507 wide character (which is the value stored).25508 between 1 and *n* inclusive25509 If the next *n* or fewer bytes complete a valid character (which is the value
25510 stored); the value returned shall be the number of bytes that complete the
25511 character.25512 `(size_t)-2` If the next *n* bytes contribute to an incomplete but potentially valid character,
25513 and all *n* bytes have been processed (no value is stored). When *n* has at least
25514 the value of the `{MB_CUR_MAX}` macro, this case can only occur if *s* points at
25515 a sequence of redundant shift sequences (for implementations with state-
25516 dependent encodings).25517 `(size_t)-1` If an encoding error occurs, in which case the next *n* or fewer bytes do not
25518 contribute to a complete and valid character (no value is stored). In this case,
25519 `[EILSEQ]` shall be stored in `errno` and the conversion state is undefined.

mbrtowc()**25520 ERRORS**

25521 The *mbrtowc()* function may fail if:

25522 CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

25523 [EILSEQ] Invalid character sequence is detected.

25524 EXAMPLES

25525 None.

25526 APPLICATION USAGE

25527 None.

25528 RATIONALE

25529 None.

25530 FUTURE DIRECTIONS

25531 None.

25532 SEE ALSO

25533 *mbsinit()*, *mbsrtowcs()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**wchar.h**>

25534 CHANGE HISTORY

25535 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
25536 (E).

25537 Issue 6

25538 The *mbrtowc()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

25539 The following new requirements on POSIX implementations derive from alignment with the
25540 Single UNIX Specification:

- 25541 • The [EINVAL] error condition is added.

25542 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.



25543 **NAME**
 25544 `mbsinit` — determine conversion object status

25545 **SYNOPSIS**
 25546 `#include <wchar.h>`

25547 `int mbsinit(const mbstate_t *ps);`

25548 **DESCRIPTION**

25549 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 25550 conflict between the requirements described here and the ISO C standard is unintentional. This
 25551 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25552 If *ps* is not a null pointer, the *mbsinit()* function shall determine whether the object pointed to by
 25553 *ps* describes an initial conversion state.

25554 **RETURN VALUE**

25555 The *mbsinit()* function shall return non-zero if *ps* is a null pointer, or if the pointed-to object
 25556 describes an initial conversion state; otherwise, it shall return zero.

25557 If an **mbstate_t** object is altered by any of the functions described as “restartable”, and is then
 25558 used with a different character sequence, or in the other conversion direction, or with a different
 25559 *LC_CTYPE* category setting than on earlier function calls, the behavior is undefined.

25560 **ERRORS**

25561 No errors are defined.

25562 **EXAMPLES**

25563 None.

25564 **APPLICATION USAGE**

25565 The **mbstate_t** object is used to describe the current conversion state from a particular character
 25566 sequence to a wide-character sequence (or *vice versa*) under the rules of a particular setting of the
 25567 *LC_CTYPE* category of the current locale.

25568 The initial conversion state corresponds, for a conversion in either direction, to the beginning of
 25569 a new character sequence in the initial shift state. A zero valued **mbstate_t** object is at least one
 25570 way to describe an initial conversion state. A zero valued **mbstate_t** object can be used to initiate
 25571 conversion involving any character sequence, in any *LC_CTYPE* category setting.

25572 **RATIONALE**

25573 None.

25574 **FUTURE DIRECTIONS**

25575 None.

25576 **SEE ALSO**

25577 *mbrlen()*, *mbrtowc()*, *mbsrtowcs()*, *wcrtomb()*, *wcsrtombs()*, the Base Definitions volume of
 25578 IEEE Std 1003.1-200x, `<wchar.h>`

25579 **CHANGE HISTORY**

25580 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
 25581 (E).

mbsnrtowcs()*System Interfaces*25582 **NAME**25583 `mbsnrtowcs` — convert a multi-byte string to a wide-character string25584 **SYNOPSIS**

```
25585 CX #include <wchar.h>
25586 size_t mbsnrtowcs(wchar_t *restrict dst, const char **restrict src,
25587 size_t nmc, size_t len, mbstate_t *restrict ps);
```

25588 **DESCRIPTION**25589 Refer to [mbsrtowcs\(\)](#).

25590 **NAME**
 25591 `mbsnrrowcs, mbsrtowcs` — convert a character string to a wide-character string (restartable)

25592 **SYNOPSIS**
 25593 `#include <wchar.h>`

25594 CX `size_t mbsnrrowcs(wchar_t *restrict dst, const char **restrict src,`
 25595 `size_t nmc, size_t len, mbstate_t *restrict ps);`
 25596 `size_t mbsrtowcs(wchar_t *restrict dst, const char **restrict src,`
 25597 `size_t len, mbstate_t *restrict ps);`

25598 **DESCRIPTION**

25599 CX For `mbsrtowcs()`: The functionality described on this reference page is aligned with the ISO C
 25600 standard. Any conflict between the requirements described here and the ISO C standard is
 25601 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25602 The `mbsrtowcs()` function shall convert a sequence of characters, beginning in the conversion
 25603 state described by the object pointed to by `ps`, from the array indirectly pointed to by `src` into a
 25604 sequence of corresponding wide characters. If `dst` is not a null pointer, the converted characters
 25605 shall be stored into the array pointed to by `dst`. Conversion continues up to and including a
 25606 terminating null character, which shall also be stored. Conversion shall stop early in either of the
 25607 following cases:

- 25608 • A sequence of bytes is encountered that does not form a valid character.
- 25609 • `len` codes have been stored into the array pointed to by `dst` (and `dst` is not a null pointer).

25610 Each conversion shall take place as if by a call to the `mbrtowc()` function.

25611 If `dst` is not a null pointer, the pointer object pointed to by `src` shall be assigned either a null
 25612 pointer (if conversion stopped due to reaching a terminating null character) or the address just
 25613 past the last character converted (if any). If conversion stopped due to reaching a terminating
 25614 null character, and if `dst` is not a null pointer, the resulting state described shall be the initial
 25615 conversion state.

25616 If `ps` is a null pointer, the `mbsrtowcs()` function shall use its own internal `mbstate_t` object, which
 25617 is initialized at program start-up to the initial conversion state. Otherwise, the `mbstate_t` object
 25618 pointed to by `ps` shall be used to completely describe the current conversion state of the
 25619 associated character sequence.

25620 CX The `mbsnrrowcs()` function shall be equivalent to the `mbsrtowcs()` function, except that the
 25621 conversion of characters pointed to by `src` is limited to at most `nmc` bytes (the size of the input
 25622 buffer).

25623 The behavior of these functions shall be affected by the `LC_CTYPE` category of the current locale.

25624 The implementation shall behave as if no function defined in this volume of
 25625 IEEE Std 1003.1-200x calls these functions.

25626 **RETURN VALUE**

25627 If the input conversion encounters a sequence of bytes that do not form a valid character, an
 25628 encoding error occurs. In this case, these functions shall store the value of the macro `[EILSEQ]` in
 25629 `errno` and shall return `(size_t)-1`; the conversion state is undefined. Otherwise, these functions
 25630 shall return the number of characters successfully converted, not including the terminating null
 25631 (if any).

mbsrtowcs()25632 **ERRORS**

25633 These functions may fail if:

25634 CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

25635 [EILSEQ] Invalid character sequence is detected.

25636 **EXAMPLES**

25637 None.

25638 **APPLICATION USAGE**

25639 None.

25640 **RATIONALE**

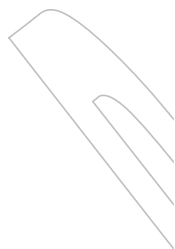
25641 None.

25642 **FUTURE DIRECTIONS**

25643 None.

25644 **SEE ALSO**25645 *iconv()*, *mbrtowc()*, *mbsinit()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**25646 **CHANGE HISTORY**25647 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
25648 (E).25649 **Issue 6**25650 The *mbsrtowcs()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

25651 The [EINVAL] error condition is marked CX.

25652 **Issue 7**25653 The *mbsnrtowcs()* function is added from The Open Group Technical Standard, 2006, Extended
25654 API Set Part 1. <

25655 **NAME**
 25656 `mbstowcs` — convert a character string to a wide-character string

25657 **SYNOPSIS**
 25658 `#include <stdlib.h>`
 25659 `size_t mbstowcs(wchar_t *restrict pwcs, const char *restrict s,`
 25660 `size_t n);`

25661 DESCRIPTION

25662 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 25663 conflict between the requirements described here and the ISO C standard is unintentional. This
 25664 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25665 The `mbstowcs()` function shall convert a sequence of characters that begins in the initial shift
 25666 state from the array pointed to by `s` into a sequence of corresponding wide-character codes and
 25667 shall store not more than `n` wide-character codes into the array pointed to by `pwcs`. No
 25668 characters that follow a null byte (which is converted into a wide-character code with value 0)
 25669 shall be examined or converted. Each character shall be converted as if by a call to `mbtowc()`,
 25670 except that the shift state of `mbtowc()` is not affected.

25671 No more than `n` elements shall be modified in the array pointed to by `pwcs`. If copying takes
 25672 place between objects that overlap, the behavior is undefined.

25673 XSI The behavior of this function shall be affected by the `LC_CTYPE` category of the current locale.
 25674 If `pwcs` is a null pointer, `mbstowcs()` shall return the length required to convert the entire array
 25675 regardless of the value of `n`, but no values are stored.

25676 RETURN VALUE

25677 CX If an invalid character is encountered, `mbstowcs()` shall return `(size_t)-1` and may set `errno` to
 25678 indicate the error.

25679 XSI Otherwise, `mbstowcs()` shall return the number of the array elements modified (or required if
 25680 `pwcs` is null), not including a terminating 0 code, if any. The array shall not be zero-terminated if
 25681 the value returned is `n`.

25682 ERRORS

25683 The `mbstowcs()` function may fail if:

25684 XSI **[EILSEQ]** Invalid byte sequence is detected.

25685 EXAMPLES

25686 None.

25687 APPLICATION USAGE

25688 None.

25689 RATIONALE

25690 None.

25691 FUTURE DIRECTIONS

25692 None.

25693 SEE ALSO

25694 `mblen()`, `mbtowc()`, `wctomb()`, `wcstombs()`, the Base Definitions volume of IEEE Std 1003.1-200x,
 25695 `<stdlib.h>`

mbstowcs()

25696

CHANGE HISTORY

25697

First released in Issue 4. Aligned with the ISO C standard.

25698

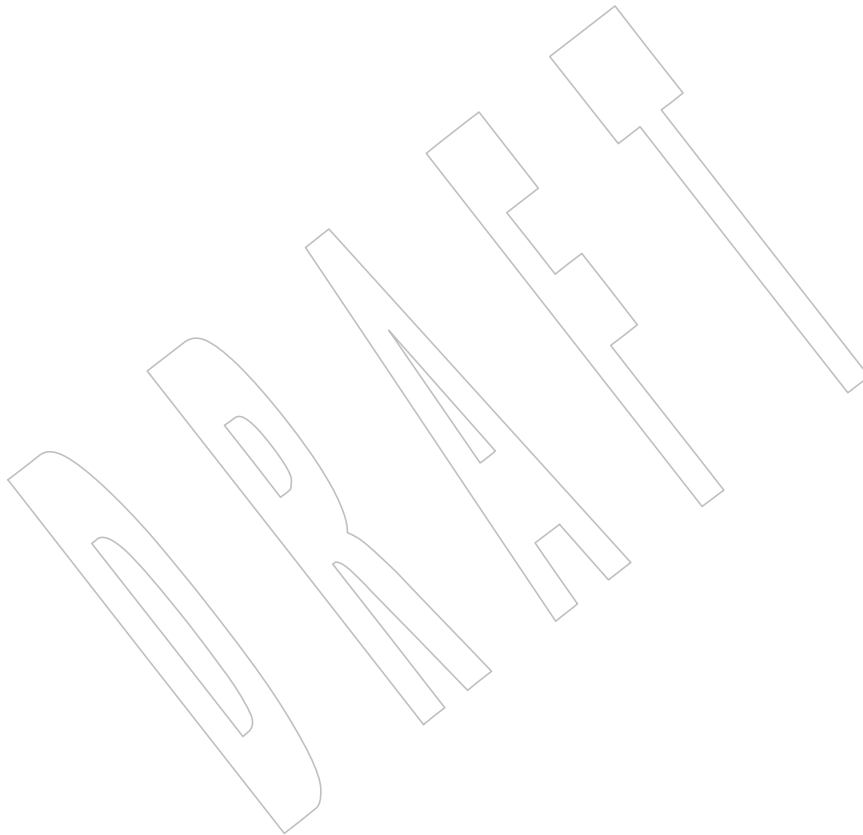
Issue 6

25699

The *mbstowcs()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

25700

Extensions beyond the ISO C standard are marked.



25701 **NAME**
 25702 `mbtowc` — convert a character to a wide-character code

25703 **SYNOPSIS**
 25704 `#include <stdlib.h>`

25705 `int mbtowc(wchar_t *restrict pwc, const char *restrict s, size_t n);`

25706 DESCRIPTION

25707 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 25708 conflict between the requirements described here and the ISO C standard is unintentional. This
 25709 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25710 If *s* is not a null pointer, *mbtowc()* shall determine the number of bytes that constitute the
 25711 character pointed to by *s*. It shall then determine the wide-character code for the value of type
 25712 `wchar_t` that corresponds to that character. (The value of the wide-character code corresponding
 25713 to the null byte is 0.) If the character is valid and *pwc* is not a null pointer, *mbtowc()* shall store
 25714 the wide-character code in the object pointed to by *pwc*.

25715 The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a
 25716 state-dependent encoding, this function is placed into its initial state by a call for which its
 25717 character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null
 25718 pointer shall cause the internal state of the function to be altered as necessary. A call with *s* as a
 25719 null pointer shall cause this function to return a non-zero value if encodings have state
 25720 dependency, and 0 otherwise. If the implementation employs special bytes to change the shift
 25721 state, these bytes shall not produce separate wide-character codes, but shall be grouped with an
 25722 adjacent character. Changing the *LC_CTYPE* category causes the shift state of this function to be
 25723 unspecified. At most *n* bytes of the array pointed to by *s* shall be examined.

25724 The implementation shall behave as if no function defined in this volume of
 25725 IEEE Std 1003.1-200x calls *mbtowc()*.

25726 RETURN VALUE

25727 If *s* is a null pointer, *mbtowc()* shall return a non-zero or 0 value, if character encodings,
 25728 respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *mbtowc()*
 25729 shall either return 0 (if *s* points to the null byte), or return the number of bytes that constitute the
 25730 converted character (if the next *n* or fewer bytes form a valid character), or return -1 and may
 25731 CX set *errno* to indicate the error (if they do not form a valid character).

25732 In no case shall the value returned be greater than *n* or the value of the `{MB_CUR_MAX}` macro.

25733 ERRORS

25734 The *mbtowc()* function may fail if:

25735 XSI `[EILSEQ]` Invalid character sequence is detected.

25736 EXAMPLES

25737 None.

25738 APPLICATION USAGE

25739 None.

25740 RATIONALE

25741 None.

25742

FUTURE DIRECTIONS

25743

None.

25744

SEE ALSO

25745

mblen(), *mbstowcs()*, *wctomb()*, *wcstombs()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdlib.h>`

25746

25747

CHANGE HISTORY

25748

First released in Issue 4. Aligned with the ISO C standard.

25749

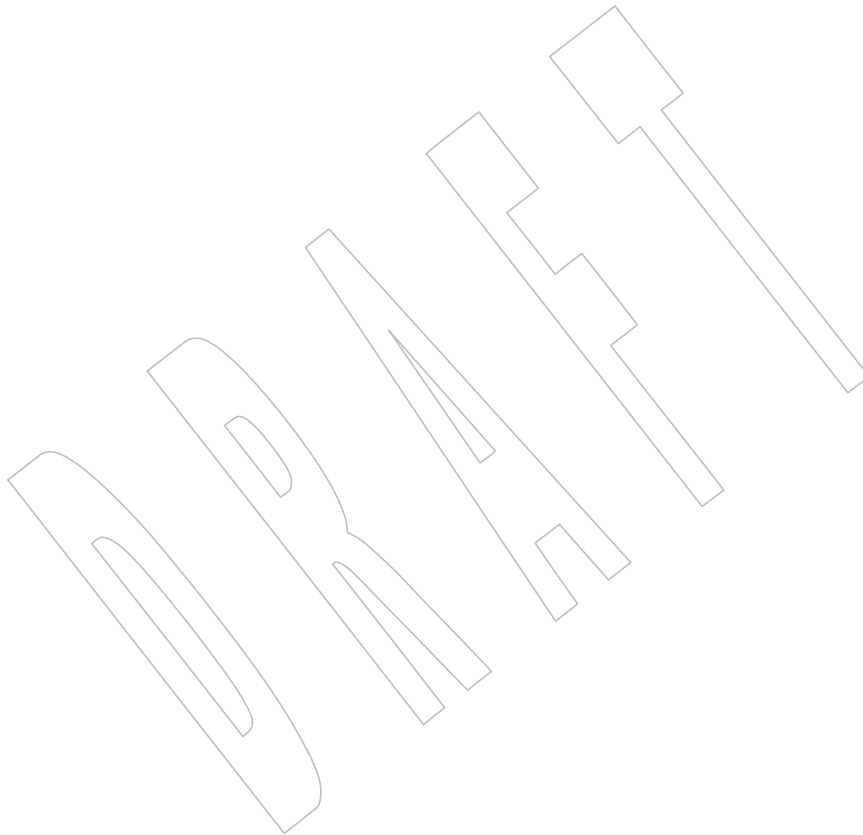
Issue 6

25750

The *mbtowc()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

25751

Extensions beyond the ISO C standard are marked.



25752 **NAME**
 25753 memccpy — copy bytes in memory

25754 **SYNOPSIS**

```
25755 XSI #include <string.h>
25756 void *memccpy(void *restrict s1, const void *restrict s2,
25757 int c, size_t n);
```

25758 **DESCRIPTION**

25759 The *memccpy()* function shall copy bytes from memory area *s2* into *s1*, stopping after the first
 25760 occurrence of byte *c* (converted to an **unsigned char**) is copied, or after *n* bytes are copied,
 25761 whichever comes first. If copying takes place between objects that overlap, the behavior is
 25762 undefined.

25763 **RETURN VALUE**

25764 The *memccpy()* function shall return a pointer to the byte after the copy of *c* in *s1*, or a null
 25765 pointer if *c* was not found in the first *n* bytes of *s2*.

25766 **ERRORS**

25767 No errors are defined.

25768 **EXAMPLES**

25769 None.

25770 **APPLICATION USAGE**

25771 The *memccpy()* function does not check for the overflow of the receiving memory area.

25772 **RATIONALE**

25773 None.

25774 **FUTURE DIRECTIONS**

25775 None.

25776 **SEE ALSO**

25777 The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

25778 **CHANGE HISTORY**

25779 First released in Issue 1. Derived from Issue 1 of the SVID.

25780 **Issue 6**

25781 The **restrict** keyword is added to the *memccpy()* prototype for alignment with the
 25782 ISO/IEC 9899:1999 standard.

25783 **NAME**
 25784 memchr — find byte in memory

25785 **SYNOPSIS**
 25786 #include <string.h>

25787 void *memchr(const void *s, int c, size_t n);

25788 **DESCRIPTION**

25789 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 25790 conflict between the requirements described here and the ISO C standard is unintentional. This
 25791 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25792 The *memchr()* function shall locate the first occurrence of *c* (converted to an **unsigned char**) in
 25793 the initial *n* bytes (each interpreted as **unsigned char**) of the object pointed to by *s*.

25794 **RETURN VALUE**

25795 The *memchr()* function shall return a pointer to the located byte, or a null pointer if the byte does
 25796 not occur in the object.

25797 **ERRORS**

25798 No errors are defined.

25799 **EXAMPLES**

25800 None.

25801 **APPLICATION USAGE**

25802 None.

25803 **RATIONALE**

25804 None.

25805 **FUTURE DIRECTIONS**

25806 None.

25807 **SEE ALSO**

25808 The Base Definitions volume of IEEE Std 1003.1-200x, <**string.h**>

25809 **CHANGE HISTORY**

25810 First released in Issue 1. Derived from Issue 1 of the SVID.

25811 **NAME**

25812 memcmp — compare bytes in memory

25813 **SYNOPSIS**

25814 #include <string.h>

25815 int memcmp(const void *s1, const void *s2, size_t n);

25816 **DESCRIPTION**

25817 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 25818 conflict between the requirements described here and the ISO C standard is unintentional. This
 25819 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25820 The *memcmp()* function shall compare the first *n* bytes (each interpreted as **unsigned char**) of the
 25821 object pointed to by *s1* to the first *n* bytes of the object pointed to by *s2*.

25822 The sign of a non-zero return value shall be determined by the sign of the difference between the
 25823 values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the objects
 25824 being compared.

25825 **RETURN VALUE**

25826 The *memcmp()* function shall return an integer greater than, equal to, or less than 0, if the object
 25827 pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*, respectively.

25828 **ERRORS**

25829 No errors are defined.

25830 **EXAMPLES**

25831 None.

25832 **APPLICATION USAGE**

25833 None.

25834 **RATIONALE**

25835 None.

25836 **FUTURE DIRECTIONS**

25837 None.

25838 **SEE ALSO**25839 The Base Definitions volume of IEEE Std 1003.1-200x, <**string.h**>25840 **CHANGE HISTORY**

25841 First released in Issue 1. Derived from Issue 1 of the SVID.

25842 **NAME**25843 `memcpy` — copy bytes in memory25844 **SYNOPSIS**25845 `#include <string.h>`25846 `void *memcpy(void *restrict s1, const void *restrict s2, size_t n);`25847 **DESCRIPTION**

25848 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 25849 conflict between the requirements described here and the ISO C standard is unintentional. This
 25850 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25851 The `memcpy()` function shall copy *n* bytes from the object pointed to by *s2* into the object pointed
 25852 to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.

25853 **RETURN VALUE**25854 The `memcpy()` function shall return *s1*; no return value is reserved to indicate an error.25855 **ERRORS**

25856 No errors are defined.

25857 **EXAMPLES**

25858 None.

25859 **APPLICATION USAGE**25860 The `memcpy()` function does not check for the overflow of the receiving memory area.25861 **RATIONALE**

25862 None.

25863 **FUTURE DIRECTIONS**

25864 None.

25865 **SEE ALSO**25866 The Base Definitions volume of IEEE Std 1003.1-200x, `<string.h>`25867 **CHANGE HISTORY**

25868 First released in Issue 1. Derived from Issue 1 of the SVID.

25869 **Issue 6**25870 The `memcpy()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

25871 **NAME**
 25872 memmove — copy bytes in memory with overlapping areas

25873 **SYNOPSIS**
 25874 #include <string.h>
 25875 void *memmove(void *s1, const void *s2, size_t n);

25876 **DESCRIPTION**
 25877 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 25878 conflict between the requirements described here and the ISO C standard is unintentional. This
 25879 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25880 The *memmove()* function shall copy *n* bytes from the object pointed to by *s2* into the object
 25881 pointed to by *s1*. Copying takes place as if the *n* bytes from the object pointed to by *s2* are first
 25882 copied into a temporary array of *n* bytes that does not overlap the objects pointed to by *s1* and
 25883 *s2*, and then the *n* bytes from the temporary array are copied into the object pointed to by *s1*.

25884 **RETURN VALUE**
 25885 The *memmove()* function shall return *s1*; no return value is reserved to indicate an error.

25886 **ERRORS**
 25887 No errors are defined.

25888 **EXAMPLES**
 25889 None.

25890 **APPLICATION USAGE**
 25891 None.

25892 **RATIONALE**
 25893 None.

25894 **FUTURE DIRECTIONS**
 25895 None.

25896 **SEE ALSO**
 25897 The Base Definitions volume of IEEE Std 1003.1-200x, <**string.h**>

25898 **CHANGE HISTORY**
 25899 First released in Issue 4. Derived from the ANSI C standard.

25900 **NAME**

25901 memset — set bytes in memory

25902 **SYNOPSIS**

25903 #include <string.h>

25904 void *memset(void *s, int c, size_t n);

25905 **DESCRIPTION**

25906 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 25907 conflict between the requirements described here and the ISO C standard is unintentional. This
 25908 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25909 The *memset()* function shall copy *c* (converted to an **unsigned char**) into each of the first *n* bytes
 25910 of the object pointed to by *s*.

25911 **RETURN VALUE**25912 The *memset()* function shall return *s*; no return value is reserved to indicate an error.25913 **ERRORS**

25914 No errors are defined.

25915 **EXAMPLES**

25916 None.

25917 **APPLICATION USAGE**

25918 None.

25919 **RATIONALE**

25920 None.

25921 **FUTURE DIRECTIONS**

25922 None.

25923 **SEE ALSO**25924 The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**25925 **CHANGE HISTORY**

25926 First released in Issue 1. Derived from Issue 1 of the SVID.

25927 **NAME**

25928 mkdir, mkdirat — make a directory relative to directory file descriptor

25929 **SYNOPSIS**

25930 #include <sys/stat.h>

25931 int mkdir(const char *path, mode_t mode);

25932 int mkdirat(int fd, const char *path, mode_t mode);

25933 **DESCRIPTION**25934 The *mkdir()* function shall create a new directory with name *path*. The file permission bits of the
25935 new directory shall be initialized from *mode*. These file permission bits of the *mode* argument
25936 shall be modified by the process' file creation mask.25937 When bits in *mode* other than the file permission bits are set, the meaning of these additional bits
25938 is implementation-defined.25939 The directory's user ID shall be set to the process' effective user ID. The directory's group ID
25940 shall be set to the group ID of the parent directory or to the effective group ID of the process.
25941 Implementations shall provide a way to initialize the directory's group ID to the group ID of the
25942 parent directory. Implementations may, but need not, provide an implementation-defined way
25943 to initialize the directory's group ID to the effective group ID of the calling process.

25944 The newly created directory shall be an empty directory.

25945 If *path* names a symbolic link, *mkdir()* shall fail and set *errno* to [EEXIST].25946 Upon successful completion, *mkdir()* shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
25947 fields of the directory. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the
25948 new entry shall be marked for update.25949 The *mkdirat()* function shall be equivalent to the *mkdir()* function except in the case where *path*
25950 specifies a relative path. In this case the newly created directory is created relative to the
25951 directory associated with the file descriptor *fd* instead of the current working directory. It is
25952 unspecified whether directory searches are permitted based on whether the file was opened
25953 with search permission or on the current permissions of the directory underlying the file
25954 descriptor.25955 If *mkdirat()* is passed the special value AT_FDCWD in the *fd* parameter, the current working
25956 directory is used and the behavior shall be identical to a call to *mkdir()*.25957 **RETURN VALUE**25958 Upon successful completion, these functions shall return 0. Otherwise, these functions shall
25959 return -1 and set *errno* to indicate the error. If -1 is returned, no directory shall be created.25960 **ERRORS**

25961 These functions shall fail if:

25962 [EACCES] Search permission is denied on a component of the path prefix, or write
25963 permission is denied on the parent directory of the directory to be created.

25964 [EEXIST] The named file exists.

25965 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
25966 argument.

25967 [EMLINK] The link count of the parent directory would exceed {LINK_MAX}.

- 25968 [ENAMETOOLONG]
 25969 The length of the *path* argument exceeds {PATH_MAX} or a pathname
 25970 component is longer than {NAME_MAX}.
- 25971 [ENOENT] A component of the path prefix specified by *path* does not name an existing
 25972 directory or *path* is an empty string.
- 25973 [ENOSPC] The file system does not contain enough space to hold the contents of the new
 25974 directory or to extend the parent directory of the new directory.
- 25975 [ENOTDIR] A component of the path prefix is not a directory.
- 25976 [EROFS] The parent directory resides on a read-only file system.
- 25977 In addition, the *mkdirat*() function shall fail if:
- 25978 [EBADF] The *path* argument does not specify an absolute path and the *fd* argument is
 25979 neither AT_FDCWD nor a valid file descriptor open for searching.

25980 These functions may fail if:

- 25981 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 25982 resolution of the *path* argument.
- 25983 [ENAMETOOLONG]
 25984 As a result of encountering a symbolic link in resolution of the *path* argument,
 25985 the length of the substituted pathname string exceeded {PATH_MAX}.
- 25986 The *mkdirat*() function may fail if:
- 25987 [ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a
 25988 file descriptor associated with a directory.

25989 EXAMPLES

25990 Creating a Directory

25991 The following example shows how to create a directory named **/home/cnd/mod1**, with
 25992 read/write/search permissions for owner and group, and with read/search permissions for
 25993 others.

```
25994 #include <sys/types.h>
25995 #include <sys/stat.h>
25996
25997 int status;
25998 ...
25999 status = mkdir("/home/cnd/mod1", S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
```

25999 APPLICATION USAGE

26000 None.

26001 RATIONALE

26002 The *mkdir*() function originated in 4.2 BSD and was added to System V in Release 3.0.

26003 4.3 BSD detects [ENAMETOOLONG].

26004 The POSIX.1-1990 standard required that the group ID of a newly created directory be set to the
 26005 group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2
 26006 required that implementations provide a way to have the group ID be set to the group ID of the
 26007 containing directory, but did not prohibit implementations also supporting a way to set the
 26008 group ID to the effective group ID of the creating process. Conforming applications should not
 26009 assume which group ID will be used. If it matters, an application can use *chown*() to set the
 26010 group ID after the directory is created, or determine under what conditions the implementation
 26011 will set the desired group ID.

26012 The purpose of the *mkdirat()* function is to create a directory in directories other than the current
 26013 working directory without exposure to race conditions. Any part of the path of a file could be
 26014 changed in parallel to the call to *mkdir()*, resulting in unspecified behavior. By opening a file
 26015 descriptor for the target directory and using the *mkdirat()* function it can be guaranteed that the
 26016 newly created directory is located relative to the desired directory.

26017 FUTURE DIRECTIONS

26018 None.

26019 SEE ALSO

26020 *chmod()*, *mkdtemp()*, *mknod()*, *umask()*, the Base Definitions volume of IEEE Std 1003.1-200x,
 26021 `<sys/stat.h>`, `<sys/types.h>`

26022 CHANGE HISTORY

26023 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

26024 Issue 6

26025 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

26026 The following new requirements on POSIX implementations derive from alignment with the
 26027 Single UNIX Specification:

- 26028 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
 26029 required for conforming implementations of previous POSIX specifications, it was not
 26030 required for UNIX applications.
- 26031 • The [ELOOP] mandatory error condition is added.
- 26032 • A second [ENAMETOOLONG] is added as an optional error condition.

26033 The following changes were made to align with the IEEE P1003.1a draft standard:

- 26034 • The [ELOOP] optional error condition is added.

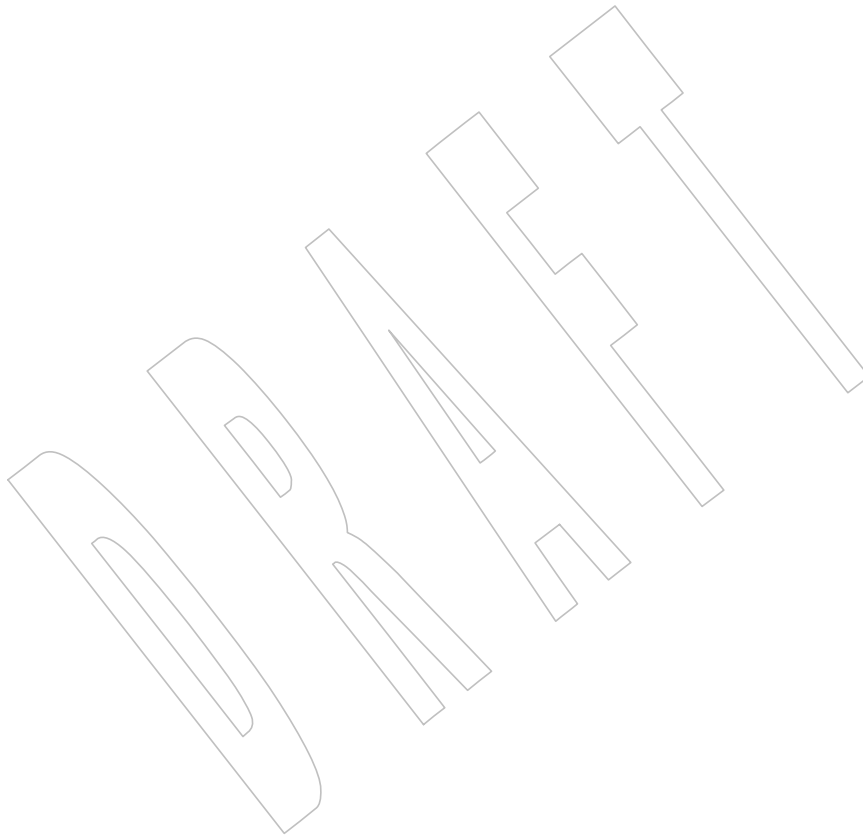
26035 Issue 7

26036 The *mkdirat()* function is added from The Open Group Technical Standard, 2006, Extended API
 26037 Set Part 2.

26038 **NAME**
26039 mkdirat — make a directory relative to directory file descriptor

26040 **SYNOPSIS**
26041 #include <sys/stat.h>
26042 int mkdirat(int *fd*, const char **path*, mode_t *mode*);

26043 **DESCRIPTION**
26044 Refer to *mkdir()*.



26045 **NAME**
 26046 mkdtemp, mkstemp — create a unique directory or file

26047 **SYNOPSIS**

```
26048 CX #include <stdlib.h>
26049 char *mkdtemp(char *template);
26050 int mkstemp(char *template);
```

26051 **DESCRIPTION**

26052 The *mkdtemp()* function uses the contents of *template* to construct a unique directory name. The
 26053 string provided in *template* shall be a filename ending with six trailing 'X's. The *mkdtemp()*
 26054 function shall replace each 'X' with a character from the portable filename character set. The
 26055 characters are chosen such that the resulting name does not duplicate the name of an existing file
 26056 at the time of a call to *mkdtemp()*. The unique directory name is used to attempt to create the
 26057 directory using mode 0700 as modified by the file creation mask.

26058 The *mkstemp()* function shall replace the contents of the string pointed to by *template* by a unique
 26059 filename, and return a file descriptor for the file open for reading and writing. The *mkstemp()*
 26060 function shall create the file, and obtain a file descriptor for it, as if by a call to:

```
26061 open(filename, O_RDWR|O_CREAT|O_EXCL, S_IRUSR|S_IWUSR)
```

26062 The function thus prevents any possible race condition between testing whether the file exists
 26063 and opening it for use. The string in *template* should look like a filename with six trailing 'X's;
 26064 *mkstemp()* replaces each 'X' with a character from the portable filename character set. The
 26065 characters are chosen such that the resulting name does not duplicate the name of an existing file
 26066 at the time of a call to *mkstemp()*.

26067 **RETURN VALUE**

26068 Upon successful completion, the *mkdtemp()* function shall return a pointer to the string
 26069 containing the directory name if it was created. Otherwise, it shall return a null pointer and shall
 26070 set *errno* to indicate the error.

26071 Upon successful completion, *mkstemp()* shall return an open file descriptor. Otherwise, -1 shall
 26072 be returned if no suitable file could be created.

26073 **ERRORS**

26074 These functions shall fail if:

- | | | |
|-------|----------------|--|
| 26075 | [EACCES] | Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the directory to be created. |
| 26076 | | |
| 26077 | [EINVAL] | The string pointed to by <i>template</i> does not end in "XXXXXX". |
| 26078 | [ELOOP] | A loop exists in symbolic links encountered during resolution of the path of the directory to be created. |
| 26079 | | |
| 26080 | [EMLINK] | The link count of the parent directory would exceed {LINK_MAX}. |
| 26081 | [ENAMETOOLONG] | |
| 26082 | | The length of the <i>template</i> argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}. |
| 26083 | | |
| 26084 | [ENOENT] | A component of the path prefix specified by the <i>template</i> argument does not name an existing directory or path is an empty string. |
| 26085 | | |

- 26086 [ENOSPC] The file system does not contain enough space to hold the contents of the new
26087 directory or to extend the parent directory of the new directory.
- 26088 [ENOTDIR] A component of the path prefix is not a directory.
- 26089 [EROFS] The parent directory resides on a read-only file system.
- 26090 These functions may fail if:
- 26091 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
26092 resolution of the path of the directory to be created.
- 26093 [ENAMETOOLONG]
26094 As a result of encountering a symbolic link in resolution of the path of the
26095 directory to be created, the length of the substituted pathname string exceeded
26096 {PATH_MAX}.

EXAMPLES**Generating a Filename**

The following example creates a file with a 10-character name beginning with the characters "file" and opens the file for reading and writing. The value returned as the value of *fd* is a file descriptor that identifies the file.

```
#include <stdlib.h>
...
char template[] = "/tmp/fileXXXXXX";
int fd;

fd = mkstemp(template);
```

APPLICATION USAGE

It is possible to run out of letters.

The *mkdtemp()* and *mkstemp()* functions need not check to determine whether the filename part of *template* exceeds the maximum allowable filename length.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[getpid\(\)](#), [mkdir\(\)](#), [open\(\)](#), [tmpfile\(\)](#), [tmpnam\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, [<stdlib.h>](#)

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 7

The *mkstemp()* function is moved from the XSI option to the Base.

The *mkdtemp()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

SD5-XSH-ERN-168 is applied, clarifying file permissions upon creation.

26127 **NAME**

26128 mkfifo, mkfifoat — make a FIFO special file relative to directory file descriptor

26129 **SYNOPSIS**

26130 #include <sys/stat.h>

26131 int mkfifo(const char *path, mode_t mode);

26132 int mkfifoat(int fd, const char *path, mode_t mode);

26133 **DESCRIPTION**26134 The *mkfifo()* function shall create a new FIFO special file named by the pathname pointed to by
26135 *path*. The file permission bits of the new FIFO shall be initialized from *mode*. The file permission
26136 bits of the *mode* argument shall be modified by the process' file creation mask.26137 When bits in *mode* other than the file permission bits are set, the effect is implementation-
26138 defined.26139 If *path* names a symbolic link, *mkfifo()* shall fail and set *errno* to [EEXIST].26140 The FIFO's user ID shall be set to the process' effective user ID. The FIFO's group ID shall be set
26141 to the group ID of the parent directory or to the effective group ID of the process.
26142 Implementations shall provide a way to initialize the FIFO's group ID to the group ID of the
26143 parent directory. Implementations may, but need not, provide an implementation-defined way
26144 to initialize the FIFO's group ID to the effective group ID of the calling process.26145 Upon successful completion, *mkfifo()* shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
26146 fields of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new
26147 entry shall be marked for update.26148 The *mkfifoat()* function shall be equivalent to the *mkfifo()* function except in the case where *path*
26149 specifies a relative path. In this case the newly created FIFO is created relative to the directory
26150 associated with the file descriptor *fd* instead of the current working directory. It is unspecified
26151 whether directory searches are permitted based on whether the file was opened with search
26152 permission or on the current permissions of the directory underlying the file descriptor.26153 If *mkfifoat()* is passed the special value AT_FDCWD in the *fd* parameter, the current working
26154 directory is used and the behavior shall be identical to a call to *mkfifo()*.26155 **RETURN VALUE**26156 Upon successful completion, these functions shall return 0. Otherwise, these functions shall
26157 return -1 and set *errno* to indicate the error. If -1 is returned, no FIFO shall be created.26158 **ERRORS**

26159 These functions shall fail if:

26160 [EACCES] A component of the path prefix denies search permission, or write permission
26161 is denied on the parent directory of the FIFO to be created.

26162 [EEXIST] The named file already exists.

26163 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
26164 argument.

26165 [ENAMETOOLONG]

26166 The length of the *path* argument exceeds {PATH_MAX} or a pathname
26167 component is longer than {NAME_MAX}.26168 [ENOENT] A component of the path prefix specified by *path* does not name an existing
26169 directory or *path* is an empty string.

- 26170 [ENOSPC] The directory that would contain the new file cannot be extended or the file
26171 system is out of file-allocation resources.
- 26172 [ENOTDIR] A component of the path prefix is not a directory.
- 26173 [EROFS] The named file resides on a read-only file system.
- 26174 The *mkfifoat()* function shall fail if:
- 26175 [EBADF] The *path* argument does not specify an absolute path and the *fd* argument is
26176 neither AT_FDCWD nor a valid file descriptor open for searching.
- 26177 These functions may fail if:
- 26178 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
26179 resolution of the *path* argument.
- 26180 [ENAMETOOLONG]
26181 As a result of encountering a symbolic link in resolution of the *path* argument,
26182 the length of the substituted pathname string exceeded {PATH_MAX}.
- 26183 The *mkfifoat()* function may fail if:
- 26184 [ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a
26185 file descriptor associated with a directory.

EXAMPLES**Creating a FIFO File**

The following example shows how to create a FIFO file named `/home/cnd/mod_done`, with read/write permissions for owner, and with read permissions for group and others.

```
26190 #include <sys/types.h>
26191 #include <sys/stat.h>
26192
26193 int status;
26194 ...
26195 status = mkfifo("/home/cnd/mod_done", S_IWUSR | S_IRUSR |
26196               S_IRGRP | S_IROTH);
```

APPLICATION USAGE

None.

RATIONALE

The syntax of this function is intended to maintain compatibility with historical implementations of *mknod()*. The latter function was included in the 1984 /usr/group standard but only for use in creating FIFO special files. The *mknod()* function was originally excluded from the POSIX.1-1988 standard as implementation-defined and replaced by *mkdir()* and *mkfifo()*. The *mknod()* function is now included for alignment with the Single UNIX Specification.

The POSIX.1-1990 standard required that the group ID of a newly created FIFO be set to the group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required that implementations provide a way to have the group ID be set to the group ID of the containing directory, but did not prohibit implementations also supporting a way to set the group ID to the effective group ID of the creating process. Conforming applications should not assume which group ID will be used. If it matters, an application can use *chown()* to set the group ID after the FIFO is created, or determine under what conditions the implementation will set the desired group ID.

The purpose of the *mkfifoat()* function is to create a FIFO special file in directories other than the current working directory without exposure to race conditions. Any part of the path of a file

26215 could be changed in parallel to a call to *mkfifo()*, resulting in unspecified behavior. By opening a
 26216 file descriptor for the target directory and using the *mkfifoat()* function it can be guaranteed that
 26217 the newly created FIFO is located relative to the desired directory.

26218 FUTURE DIRECTIONS

26219 None.

26220 SEE ALSO

26221 *chmod()*, *mknod()*, *umask()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<sys/stat.h>`,
 26222 `<sys/types.h>`

26223 CHANGE HISTORY

26224 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

26225 Issue 6

26226 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

26227 The following new requirements on POSIX implementations derive from alignment with the
 26228 Single UNIX Specification:

- 26229 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
 26230 required for conforming implementations of previous POSIX specifications, it was not
 26231 required for UNIX applications.
- 26232 • The [ELOOP] mandatory error condition is added.
- 26233 • A second [ENAMETOOLONG] is added as an optional error condition.

26234 The following changes were made to align with the IEEE P1003.1a draft standard:

- 26235 • The [ELOOP] optional error condition is added.

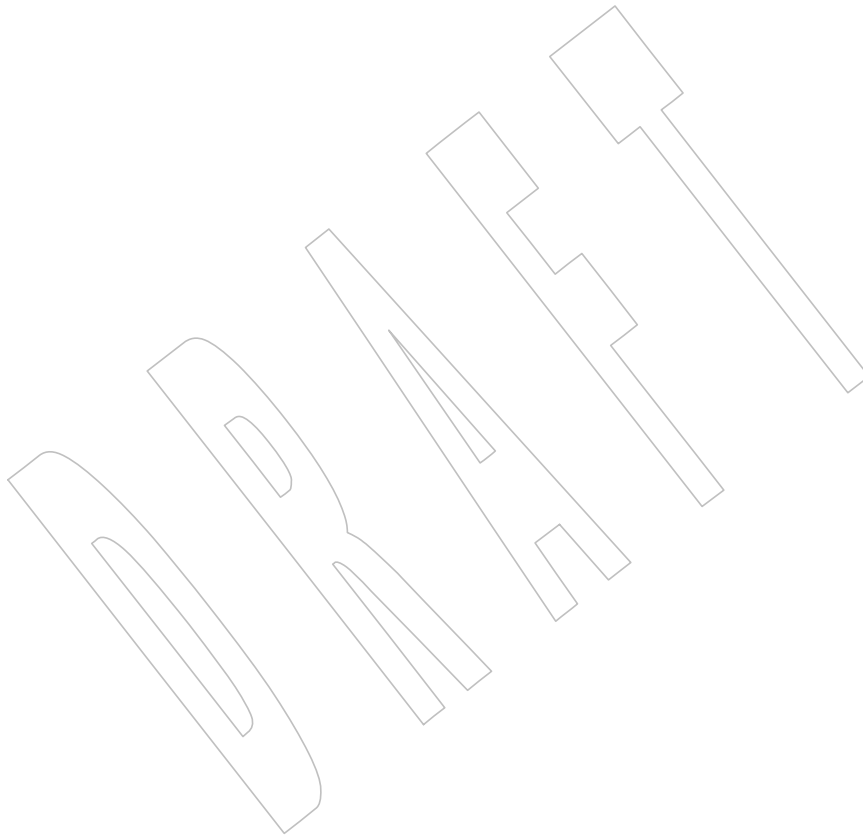
26236 Issue 7

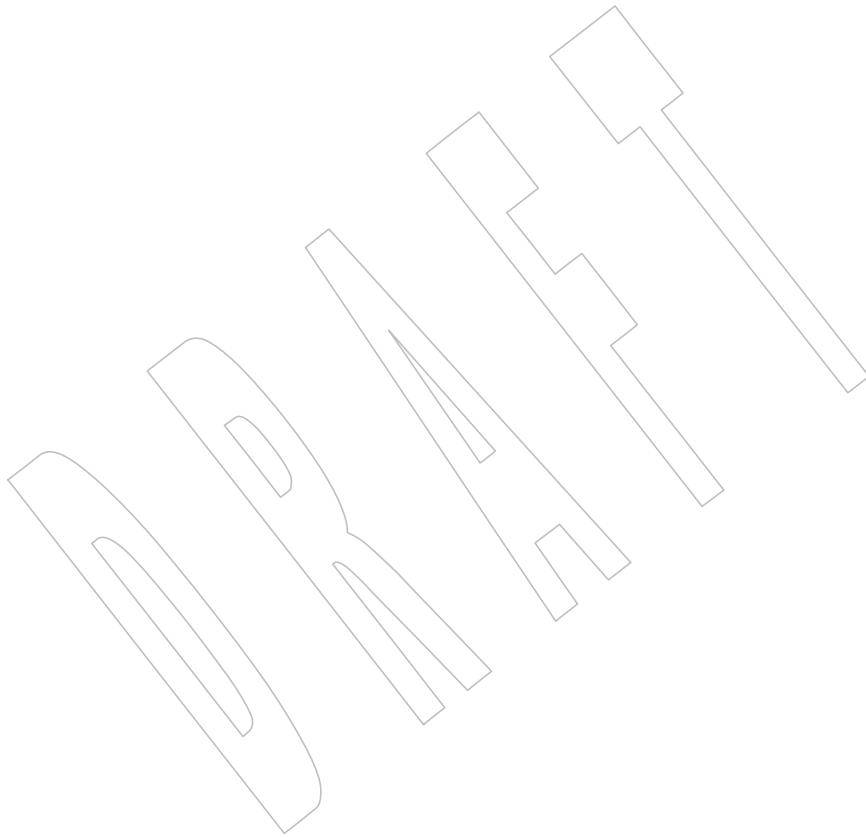
26237 The *mkfifoat()* function is added from The Open Group Technical Standard, 2006, Extended API
 26238 Set Part 2.

26239 **NAME**
26240 `mkfifoat` — make a FIFO special file relative to directory file descriptor

26241 **SYNOPSIS**
26242 `#include <sys/stat.h>`
26243 `int mkfifoat(int fd, const char *path, mode_t mode);`

26244 **DESCRIPTION**
26245 Refer to *mkfifo()*.





If *path* names a symbolic link, *mknod()* shall fail and set *errno* to [EEXIST].

Upon successful completion, *mknod()* shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new entry shall be marked for update.

Only a process with appropriate privileges may invoke *mknod()* for file types other than FIFO-special.

The *mknodat()* function shall be equivalent to the *mknod()* function except in the case where *path* specifies a relative path. In this case the newly created directory, special file, or regular file is located relative to the directory associated with the file descriptor *fd* instead of the current working directory. It is unspecified whether directory searches are permitted based on whether the file was opened with search permission or on the current permissions of the directory underlying the file descriptor.

If *mknodat()* is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *mknod()*.

RETURN VALUE

Upon successful completion, these functions shall return 0. Otherwise, these functions shall return -1 and set *errno* to indicate the error. If -1 is returned, the new file shall not be created.

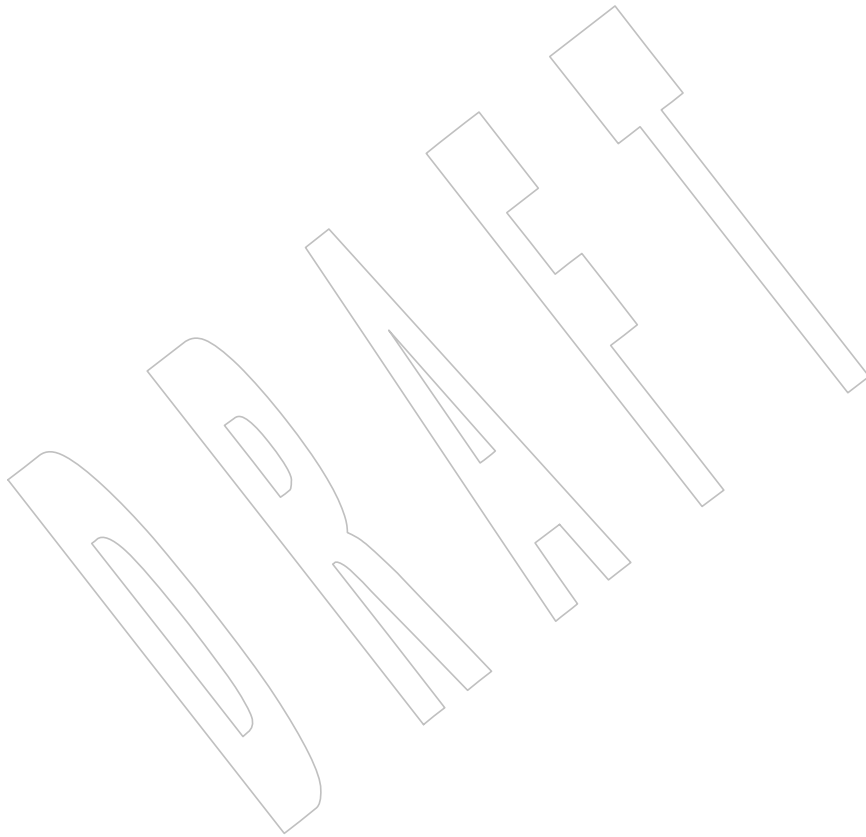
ERRORS

These functions shall fail if:

[EACCES]	A component of the path prefix denies search permission, or write permission is denied on the parent directory.
[EEXIST]	The named file exists.
[EINVAL]	An invalid argument exists.
[EIO]	An I/O error occurred while accessing the file system.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the <i>path</i> argument.
[ENAMETOOLONG]	

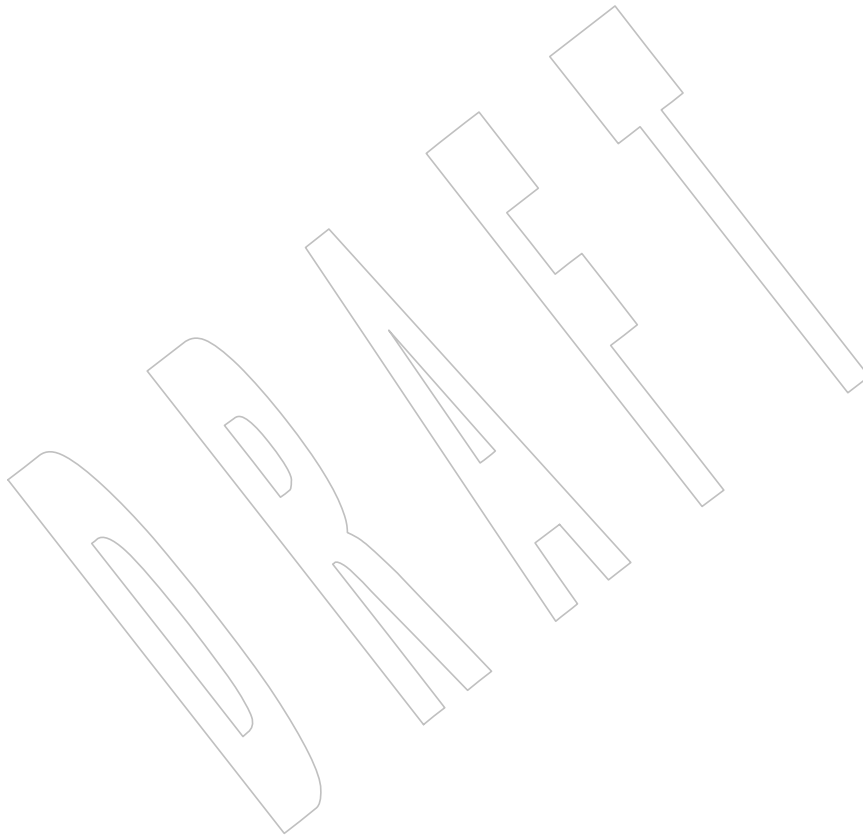
[ELOOP]

More t



mknod()

26376	Issue 5	
26377		Moved from X/OPEN UNIX extension to BASE.
26378	Issue 6	
26379		The normative text is updated to avoid use of the term “must” for application requirements.
26380		The wording of the mandatory [ELOOP] error condition is updated, and a second optional
26381		[ELOOP] error condition is added.
26382	Issue 7	
26383		The <i>mknodat()</i> function is added from The Open Group Technical Standard, 2006, Extended API
26384		Set Part 2.



26385 **NAME**
26386 mknodat — make directory, special file, or regular file

26387 **SYNOPSIS**

26388 XSI `#include <sys/stat.h>`
26389 `int mknodat(int fd, const char *path, mode_t mode, dev_t dev);`

26390 **DESCRIPTION**

26391 Refer to *mknod()*.

mkstemp()

26392 **NAME**
26393 `mkstemp` — create a unique directory

SYNOPSIS

26394 CX `#include <stdlib.h>`
26396 `int mkstemp(char *template);`

DESCRIPTION

26397 Refer to *mkdtemp()*.
26398

26399 **NAME**

26400 mktime — convert broken-down time into time since the Epoch

26401 **SYNOPSIS**

26402 #include <time.h>

26403 time_t mktime(struct tm *timeptr);

26404 **DESCRIPTION**

26405 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 26406 conflict between the requirements described here and the ISO C standard is unintentional. This
 26407 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

26408 The *mktime()* function shall convert the broken-down time, expressed as local time, in the
 26409 structure pointed to by *timeptr*, into a time since the Epoch value with the same encoding as that
 26410 of the values returned by *time()*. The original values of the *tm_wday* and *tm_yday* components of
 26411 the structure are ignored, and the original values of the other components are not restricted to
 26412 the ranges described in <time.h>.

26413 CX A positive or 0 value for *tm_isdst* shall cause *mktime()* to presume initially that Daylight Savings
 26414 Time, respectively, is or is not in effect for the specified time. A negative value for *tm_isdst* shall
 26415 cause *mktime()* to attempt to determine whether Daylight Savings Time is in effect for the
 26416 specified time.

26417 Local timezone information shall be set as though *mktime()* called *tzset()*.

26418 The relationship between the **tm** structure (defined in the <time.h> header) and the time in
 26419 seconds since the Epoch is that the result shall be as specified in the expression given in the
 26420 definition of seconds since the Epoch (see the Base Definitions volume of IEEE Std 1003.1-200x,
 26421 Section 4.14, Seconds Since the Epoch) corrected for timezone and any seasonal time
 26422 adjustments, where the names in the structure and in the expression correspond.

26423 Upon successful completion, the values of the *tm_wday* and *tm_yday* components of the structure
 26424 shall be set appropriately, and the other components are set to represent the specified time since
 26425 the Epoch, but with their values forced to the ranges indicated in the <time.h> entry; the final
 26426 value of *tm_mday* shall not be set until *tm_mon* and *tm_year* are determined.

26427 **RETURN VALUE**

26428 The *mktime()* function shall return the specified time since the Epoch encoded as a value of type
 26429 **time_t**. If the time since the Epoch cannot be represented, the function shall return the value
 26430 **(time_t)-1** and may set *errno* to indicate the error.

26431 **ERRORS**

26432 The *mktime()* function may fail if:

26433 CX [EOVERFLOW] The result cannot be represented.

26434 **EXAMPLES**

26435 What day of the week is July 4, 2001?

26436 #include <stdio.h>

26437 #include <time.h>

26438 struct tm time_str;

26439 char daybuf[20];

26440 int main(void)

26441 {
 26442 time_str.tm_year = 2001 - 1900;

```

26443         time_str.tm_mon = 7 - 1;
26444         time_str.tm_mday = 4;
26445         time_str.tm_hour = 0;
26446         time_str.tm_min = 0;
26447         time_str.tm_sec = 1;
26448         time_str.tm_isdst = -1;
26449         if (mktime(&time_str) == -1)
26450             (void)puts("-unknown-");
26451         else {
26452             (void)strftime(daybuf, sizeof(daybuf), "%A", &time_str);
26453             (void)puts(daybuf);
26454         }
26455         return 0;
26456     }

```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asctime(), *clock()*, *ctime()*, *difftime()*, *gmtime()*, *localtime()*, *strftime()*, *strptime()*, *time()*, *tzset()*, *utime()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard and the ANSI C standard.

Issue 6

Extensions beyond the ISO C standard are marked.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/58 is applied, updating the RETURN VALUE and ERRORS sections to add the optional [Eoverflow] error as a CX extension.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/59 is applied, adding the *tzset()* function to the SEE ALSO section.

26475 **NAME**
 26476 `mlock, munlock` — lock or unlock a range of process address space (**REALTIME**)

26477 **SYNOPSIS**

```
26478 MLR #include <sys/mman.h>
26479 int mlock(const void *addr, size_t len);
26480 int munlock(const void *addr, size_t len);
```

26481 **DESCRIPTION**

26482 The `mlock()` function shall cause those whole pages containing any part of the address space of
 26483 the process starting at address `addr` and continuing for `len` bytes to be memory-resident until
 26484 unlocked or until the process exits or *execs* another process image. The implementation may
 26485 require that `addr` be a multiple of {PAGESIZE}.

26486 The `munlock()` function shall unlock those whole pages containing any part of the address space
 26487 of the process starting at address `addr` and continuing for `len` bytes, regardless of how many
 26488 times `mlock()` has been called by the process for any of the pages in the specified range. The
 26489 implementation may require that `addr` be a multiple of {PAGESIZE}.

26490 If any of the pages in the range specified to a call to `munlock()` are also mapped into the address
 26491 spaces of other processes, any locks established on those pages by another process are
 26492 unaffected by the call of this process to `munlock()`. If any of the pages in the range specified by a
 26493 call to `munlock()` are also mapped into other portions of the address space of the calling process
 26494 outside the range specified, any locks established on those pages via the other mappings are also
 26495 unaffected by this call.

26496 Upon successful return from `mlock()`, pages in the specified range shall be locked and memory-
 26497 resident. Upon successful return from `munlock()`, pages in the specified range shall be unlocked
 26498 with respect to the address space of the process. Memory residency of unlocked pages is
 26499 unspecified.

26500 The appropriate privilege is required to lock process memory with `mlock()`.

26501 **RETURN VALUE**

26502 Upon successful completion, the `mlock()` and `munlock()` functions shall return a value of zero.
 26503 Otherwise, no change is made to any locks in the address space of the process, and the function
 26504 shall return a value of `-1` and set `errno` to indicate the error.

26505 **ERRORS**

26506 The `mlock()` and `munlock()` functions shall fail if:

26507 [ENOMEM] Some or all of the address range specified by the `addr` and `len` arguments does
 26508 not correspond to valid mapped pages in the address space of the process.

26509 The `mlock()` function shall fail if:

26510 [EAGAIN] Some or all of the memory identified by the operation could not be locked
 26511 when the call was made.

26512 The `mlock()` and `munlock()` functions may fail if:

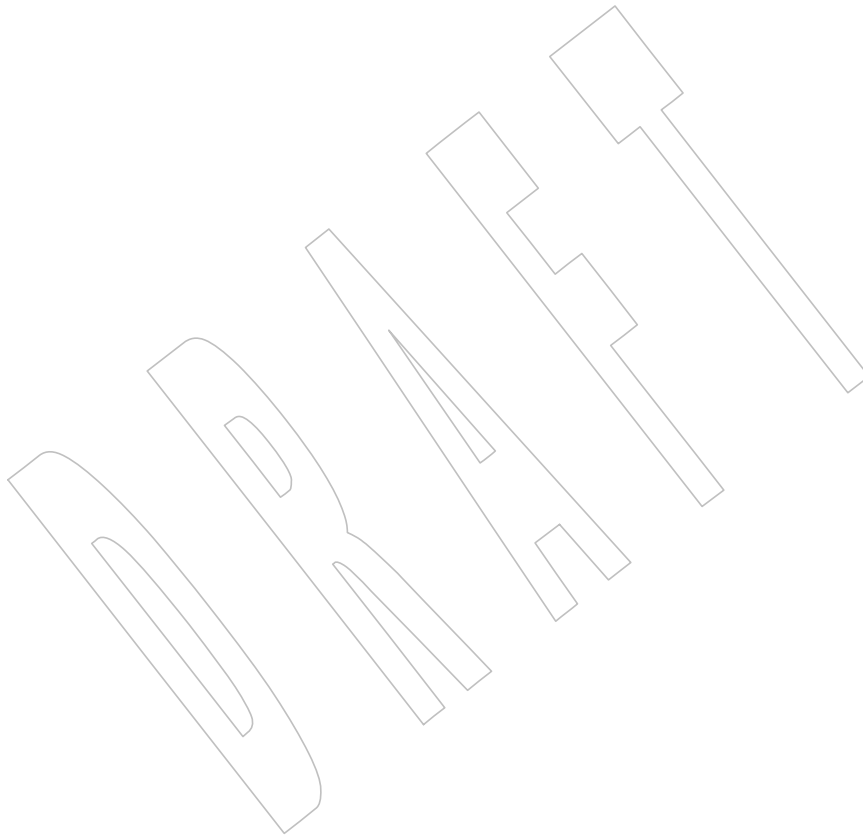
26513 [EINVAL] The `addr` argument is not a multiple of {PAGESIZE}.

26514 The `mlock()` function may fail if:

26515 [ENOMEM] Locking the pages mapped by the specified range would exceed an
 26516 implementation-defined limit on the amount of memory that the process may
 26517 lock.

mlock()

[E_{PERM}] The calling process does not have the appr



26537 **NAME**26538 `mlockall`, `munlockall` — lock/unlock the address space of a process (**REALTIME**)26539 **SYNOPSIS**

```
26540 ML      #include <sys/mman.h>
26541         int mlockall(int flags);
26542         int munlockall(void);
```

26543 **DESCRIPTION**

26544 The `mlockall()` function shall cause all of the pages mapped by the address space of a process to
 26545 be memory-resident until unlocked or until the process exits or *execs* another process image. The
 26546 *flags* argument determines whether the pages to be locked are those currently mapped by the
 26547 address space of the process, those that are mapped in the future, or both. The *flags* argument is
 26548 constructed from the bitwise-inclusive OR of one or more of the following symbolic constants,
 26549 defined in `<sys/mman.h>`:

26550 `MCL_CURRENT` Lock all of the pages currently mapped into the address space of the process.

26551 `MCL_FUTURE` Lock all of the pages that become mapped into the address space of the
 26552 process in the future, when those mappings are established.

26553 If `MCL_FUTURE` is specified, and the automatic locking of future mappings eventually causes
 26554 the amount of locked memory to exceed the amount of available physical memory or any other
 26555 implementation-defined limit, the behavior is implementation-defined. The manner in which the
 26556 implementation informs the application of these situations is also implementation-defined.

26557 The `munlockall()` function shall unlock all currently mapped pages of the address space of the
 26558 process. Any pages that become mapped into the address space of the process after a call to
 26559 `munlockall()` shall not be locked, unless there is an intervening call to `mlockall()` specifying
 26560 `MCL_FUTURE` or a subsequent call to `mlockall()` specifying `MCL_CURRENT`. If pages mapped
 26561 into the address space of the process are also mapped into the address spaces of other processes
 26562 and are locked by those processes, the locks established by the other processes shall be
 26563 unaffected by a call by this process to `munlockall()`.

26564 Upon successful return from the `mlockall()` function that specifies `MCL_CURRENT`, all currently
 26565 mapped pages of the address space of the process shall be memory-resident and locked. Upon
 26566 return from the `munlockall()` function, all currently mapped pages of the address space of the
 26567 process shall be unlocked with respect to the address space of the process. The memory
 26568 residency of unlocked pages is unspecified.

26569 The appropriate privilege is required to lock process memory with `mlockall()`.

26570 **RETURN VALUE**

26571 Upon successful completion, the `mlockall()` function shall return a value of zero. Otherwise, no
 26572 additional memory shall be locked, and the function shall return a value of `-1` and set *errno* to
 26573 indicate the error. The effect of failure of `mlockall()` on previously existing locks in the address
 26574 space is unspecified.

26575 If it is supported by the implementation, the `munlockall()` function shall always return a value of
 26576 zero. Otherwise, the function shall return a value of `-1` and set *errno* to indicate the error.

26577 **ERRORS**

26578 The `mlockall()` function shall fail if:

mlockall()

26579	[EAGAIN]	Some or all of the memory identified by the operation could not be locked
26580		when the call was made.
26581	[EINVAL]	The <i>flags</i> argument is zero, or includes unimplemented flags.
26582		The <i>mlockall()</i> function may fail if:
26583	[ENOMEM]	Locking all of the pages currently mapped into the address space of the
26584		process would exceed an implementation-defined limit on the amount of
26585		memory that the process may lock.
26586	[EPERM]	The calling process does not have the appropriate privilege to perform the
26587		requested operation.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *exit()*, *fork()*, *mlock()*, *munmap()*, the Base Definitions volume of IEEE Std 1003.1-200x, [<sys/mman.h>](#)

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mlockall()* and *munlockall()* functions are marked as part of the Process Memory Locking option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Memory Locking option.

26606 **NAME**

26607 mmap — map pages of memory

26608 **SYNOPSIS**

26609 #include <sys/mman.h>

26610 void *mmap(void *addr, size_t len, int prot, int flags,
26611 int fildes, off_t off);26612 **DESCRIPTION**26613 The *mmap()* function shall establish a mapping between an address space of a process and a
26614 memory object.26615 The *mmap()* function shall be supported for the following memory objects:

- 26616 • Regular files
- 26617 SHM • Shared memory objects
- 26618 TYM • Typed memory objects

26619 Support for any other type of file is unspecified.

26620 The format of the call is as follows:

26621 *pa*=mmap(*addr*, *len*, *prot*, *flags*, *fildes*, *off*);

26622 The *mmap()* function shall establish a mapping between the address space of the process at an
26623 address *pa* for *len* bytes to the memory object represented by the file descriptor *fildes* at offset *off*
26624 for *len* bytes. The value of *pa* is an implementation-defined function of the parameter *addr* and
26625 the values of *flags*, further described below. A successful *mmap()* call shall return *pa* as its result.
26626 The address range starting at *pa* and continuing for *len* bytes shall be legitimate for the possible
26627 (not necessarily current) address space of the process. The range of bytes starting at *off* and
26628 continuing for *len* bytes shall be legitimate for the possible (not necessarily current) offsets in the
26629 memory object represented by *fildes*.

26630 TYM If *fildes* represents a typed memory object opened with either the
26631 POSIX_TYPED_MEM_ALLOCATE flag or the POSIX_TYPED_MEM_ALLOCATE_CONTIG
26632 flag, the memory object to be mapped shall be that portion of the typed memory object allocated
26633 by the implementation as specified below. In this case, if *off* is non-zero, the behavior of *mmap()*
26634 is undefined. If *fildes* refers to a valid typed memory object that is not accessible from the calling
26635 process, *mmap()* shall fail.

26636 The mapping established by *mmap()* shall replace any previous mappings for those whole pages
26637 containing any part of the address space of the process starting at *pa* and continuing for *len*
26638 bytes.

26639 If the size of the mapped file changes after the call to *mmap()* as a result of some other operation
26640 on the mapped file, the effect of references to portions of the mapped region that correspond to
26641 added or removed portions of the file is unspecified.

26642 If *len* is zero, *mmap()* shall fail and no mapping shall be established.

26643 The parameter *prot* determines whether read, write, execute, or some combination of accesses
26644 are permitted to the data being mapped. The *prot* shall be either PROT_NONE or the bitwise-
26645 inclusive OR of one or more of the other flags in the following table, defined in the
26646 <sys/mman.h> header.

26647
26648
26649
26650
26651

Symbolic Constant	Description
PROT_READ	Data can be read.
PROT_WRITE	Data can be written.
PROT_EXEC	Data can be executed.
PROT_NONE	Data cannot be accessed.

26652
26653

If an implementation cannot support the combination of access types specified by *prot*, the call to *mmap()* shall fail.

26654
26655
26656
26657
26658
26659
26660
26661
26662

An implementation may permit accesses other than those specified by *prot*; however, the implementation shall not permit a write to succeed where PROT_WRITE has not been set and shall not permit any access where PROT_NONE alone has been set. The implementation shall support at least the following values of *prot*: PROT_NONE, PROT_READ, PROT_WRITE, and the bitwise-inclusive OR of PROT_READ and PROT_WRITE. The file descriptor *fdes* shall have been opened with read permission, regardless of the protection options specified. If PROT_WRITE is specified, the application shall ensure that it has opened the file descriptor *fdes* with write permission unless MAP_PRIVATE is specified in the *flags* parameter as described below.

26663
26664

The parameter *flags* provides other information about the handling of the mapped data. The value of *flags* is the bitwise-inclusive OR of these options, defined in `<sys/mman.h>`:

26665
26666
26667
26668

Symbolic Constant	Description
MAP_SHARED	Changes are shared.
MAP_PRIVATE	Changes are private.
MAP_FIXED	Interpret <i>addr</i> exactly.

26669
26670

XSI

It is implementation-defined whether MAP_FIXED shall be supported. MAP_FIXED shall be supported on XSI-conformant systems.

26671
26672
26673
26674
26675
26676
26677

MAP_SHARED and MAP_PRIVATE describe the disposition of write references to the memory object. If MAP_SHARED is specified, write references shall change the underlying object. If MAP_PRIVATE is specified, modifications to the mapped data by the calling process shall be visible only to the calling process and shall not change the underlying object. It is unspecified whether modifications to the underlying object done after the MAP_PRIVATE mapping is established are visible through the MAP_PRIVATE mapping. Either MAP_SHARED or MAP_PRIVATE can be specified, but not both. The mapping type is retained across *fork()*.

26678
26679
26680

The state of synchronization objects such as mutexes, semaphores, barriers, and conditional variables placed in shared memory mapped with MAP_SHARED becomes undefined when the last region in any process containing the synchronization object is unmapped.

26681
26682
26683
26684
26685
26686
26687
26688
26689
26690
26691
26692
26693
26694
26695

TYM

When *fdes* represents a typed memory object opened with either the POSIX_TYPED_MEM_ALLOCATE flag or the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, *mmap()* shall, if there are enough resources available, map *len* bytes allocated from the corresponding typed memory object which were not previously allocated to any process in any processor that may access that typed memory object. If there are not enough resources available, the function shall fail. If *fdes* represents a typed memory object opened with the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, these allocated bytes shall be contiguous within the typed memory object. If *fdes* represents a typed memory object opened with the POSIX_TYPED_MEM_ALLOCATE flag, these allocated bytes may be composed of non-contiguous fragments within the typed memory object. If *fdes* represents a typed memory object opened with neither the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag nor the POSIX_TYPED_MEM_ALLOCATE flag, *len* bytes starting at offset *off* within the typed memory object are mapped, exactly as when mapping a file or shared memory object. In this case, if two processes map an area of typed memory using the same *off* and *len* values and using file descriptors that refer to the same memory pool (either from the same port or from a different

26696 port), both processes shall map the same region of storage.

26697 When MAP_FIXED is set in the *flags* argument, the implementation is informed that the value of
 26698 *pa* shall be *addr*, exactly. If MAP_FIXED is set, *mmap()* may return MAP_FAILED and set *errno* to
 26699 [EINVAL]. If a MAP_FIXED request is successful, the mapping established by *mmap()* replaces
 26700 any previous mappings for the pages in the range [*pa,pa+len*) of the process.

26701 When MAP_FIXED is not set, the implementation uses *addr* in an implementation-defined
 26702 manner to arrive at *pa*. The *pa* so chosen shall be an area of the address space that the
 26703 implementation deems suitable for a mapping of *len* bytes to the file. All implementations
 26704 interpret an *addr* value of 0 as granting the implementation complete freedom in selecting *pa*,
 26705 subject to constraints described below. A non-zero value of *addr* is taken to be a suggestion of a
 26706 process address near which the mapping should be placed. When the implementation selects a
 26707 value for *pa*, it never places a mapping at address 0, nor does it replace any extant mapping.

26708 If MAP_FIXED is specified and *addr* is non-zero, it shall have the same remainder as the *off*
 26709 parameter, modulo the page size as returned by *sysconf()* when passed _SC_PAGESIZE or
 26710 _SC_PAGE_SIZE. The implementation may require that *off* is a multiple of the page size. If
 26711 MAP_FIXED is specified, the implementation may require that *addr* is a multiple of the page
 26712 size. The system performs mapping operations over whole pages. Thus, while the parameter *len*
 26713 need not meet a size or alignment constraint, the system shall include, in any mapping
 26714 operation, any partial page specified by the address range starting at *pa* and continuing for *len*
 26715 bytes.

26716 The system shall always zero-fill any partial page at the end of an object. Further, the system
 26717 shall never write out any modified portions of the last page of an object which are beyond its
 26718 end. References within the address range starting at *pa* and continuing for *len* bytes to whole
 26719 pages following the end of an object shall result in delivery of a SIGBUS signal.

26720 An implementation may generate SIGBUS signals when a reference would cause an error in the
 26721 mapped object, such as out-of-space condition.

26722 The *mmap()* function shall add an extra reference to the file associated with the file descriptor
 26723 *fd* which is not removed by a subsequent *close()* on that file descriptor. This reference shall be
 26724 removed when there are no more mappings to the file.

26725 The *st_atime* field of the mapped file may be marked for update at any time between the *mmap()*
 26726 call and the corresponding *munmap()* call. The initial read or write reference to a mapped region
 26727 shall cause the file's *st_atime* field to be marked for update if it has not already been marked for
 26728 update.

26729 The *st_ctime* and *st_mtime* fields of a file that is mapped with MAP_SHARED and PROT_WRITE
 26730 shall be marked for update at some point in the interval between a write reference to the
 26731 mapped region and the next call to *msync()* with MS_ASYNC or MS_SYNC for that portion of
 26732 the file by any process. If there is no such call and if the underlying file is modified as a result of
 26733 a write reference, then these fields shall be marked for update at some time after the write
 26734 reference.

26735 There may be implementation-defined limits on the number of memory regions that can be
 26736 mapped (per process or per system).

26737 XSI If such a limit is imposed, whether the number of memory regions that can be mapped by a
 26738 process is decreased by the use of *shmat()* is implementation-defined.

26739 If *mmap()* fails for reasons other than [EBADF], [EINVAL], or [ENOTSUP], some of the
 26740 mappings in the address range starting at *addr* and continuing for *len* bytes may have been
 26741 unmapped.

mmap()

26742

RETURN VALUE

26743

Upon successful completion, the *mmap()* function shall return the address at which the mapping was placed (*pa*); otherwise, it shall return a value of `MAP_FAILED` and set *errno* to indicate the error. The symbol `MAP_FAILED` is defined in the `<sys/mman.h>` header. No successful return from *mmap()* shall return the value `MAP_FAILED`.

26744

26745

26746

26747

ERRORS

26748

The *mmap()* function shall fail if:

26749

[EACCES] The *fildev* argument is not open for read, regardless of the protection specified, or *fildev* is not open for write and `PROT_WRITE` was specified for a `MAP_SHARED` type mapping.

26750

26751

26752

ML

[EAGAIN] The mapping could not be locked in memory, if required by *mlockall()*, due to a lack of resources.

26753

26754

[EBADF] The *fildev* argument is not a valid open file descriptor.

26755

[EINVAL] The value of *len* is zero.

26756

26757

[EINVAL] The value of *flags* is invalid (neither `MAP_PRIVATE` nor `MAP_SHARED` is set).

26758

26759

[EMFILE] The number of mapped regions would exceed an implementation-defined limit (per process or per system).

26760

[ENODEV] The *fildev* argument refers to a file whose type is not supported by *mmap()*.

26761

26762

26763

[ENOMEM] `MAP_FIXED` was specified, and the range [*addr*,*addr+len*) exceeds that allowed for the address space of a process; or, if `MAP_FIXED` was not specified and there is insufficient room in the address space to effect the mapping.

26764

ML

[ENOMEM] The mapping could not be locked in memory, if required by *mlockall()*, because it would require more space than the system is able to supply.

26765

26766

TYM

[ENOMEM] Not enough unallocated memory resources remain in the typed memory object designated by *fildev* to allocate *len* bytes.

26767

26768

26769

[ENOTSUP] `MAP_FIXED` or `MAP_PRIVATE` was specified in the *flags* argument and the implementation does not support this functionality.

26770

26771

The implementation does not support the combination of accesses requested in the *prot* argument.

26772

[ENXIO] Addresses in the range [*off*,*off+len*) are invalid for the object specified by *fildev*.

26773

26774

[ENXIO] `MAP_FIXED` was specified in *flags* and the combination of *addr*, *len*, and *off* is invalid for the object specified by *fildev*.

26775

TYM

[ENXIO] The *fildev* argument refers to a typed memory object that is not accessible from the calling process.

26776

26777

26778

[EOVERFLOW] The file is a regular file and the value of *off* plus *len* exceeds the offset maximum established in the open file description associated with *fildev*.

26779

The *mmap()* function may fail if:

26780

26781

26782

[EINVAL] The *addr* argument (if `MAP_FIXED` was specified) or *off* is not a multiple of the page size as returned by *sysconf()*, or is considered invalid by the implementation.

EXAMPLES

None.

APPLICATION USAGE

Use of *mmap()* may reduce the amount of memory available to other memory allocation functions.

Use of `MAP_FIXED` may result in unspecified behavior in further use of *malloc()* and *shmat()*. The use of `MAP_FIXED` is discouraged, as it may prevent an implementation from making the most effective use of resources. Most implementations require that *off* and *addr* are multiples of the page size as returned by *sysconf()*.

The application must ensure correct synchronization when using *mmap()* in conjunction with any other file access method, such as *read()* and *write()*, standard input/output, and *shmat()*.

The *mmap()* function allows access to resources via address space manipulations, instead of *read()/write()*. Once a file is mapped, all a process has to do to access it is use the data at the address to which the file was mapped. So, using pseudo-code to illustrate the way in which an existing program might be changed to use *mmap()*, the following:

```
fildes = open(...)
lseek(fildes, some_offset)
read(fildes, buf, len)
/* Use data in buf. */
```

becomes:

```
fildes = open(...)
address = mmap(0, len, PROT_READ, MAP_PRIVATE, fildes, some_offset)
/* Use data at address. */
```

RATIONALE

After considering several other alternatives, it was decided to adopt the *mmap()* definition found in SVR4 for mapping memory objects into process address spaces. The SVR4 definition is minimal, in that it describes only what has been built, and what appears to be necessary for a general and portable mapping facility.

Note that while *mmap()* was first designed for mapping files, it is actually a general-purpose mapping facility. It can be used to map any appropriate object, such as memory, files, devices, and so on, into the address space of a process.

When a mapping is established, it is possible that the implementation may need to map more than is requested into the address space of the process because of hardware requirements. An application, however, cannot count on this behavior. Implementations that do not use a paged architecture may simply allocate a common memory region and return the address of it; such implementations probably do not allocate any more than is necessary. References past the end of the requested area are unspecified.

If an application requests a mapping that would overlay existing mappings in the process, it might be desirable that an implementation detect this and inform the application. However, the default, portable (not `MAP_FIXED`) operation does not overlay existing mappings. On the other hand, if the program specifies a fixed address mapping (which requires some implementation knowledge to determine a suitable address, if the function is supported at all), then the program is presumed to be successfully managing its own address space and should be trusted when it asks to map over existing data structures. Furthermore, it is also desirable to make as few system calls as possible, and it might be considered onerous to require an *munmap()* before an *mmap()* to the same address range. This volume of IEEE Std 1003.1-200x specifies that the new mappings replace any existing mappings, following existing practice in this regard.

It is not expected that all hardware implementations are able to support all combinations of

26831 permissions at all addresses. Implementations are required to disallow write access to mappings
 26832 without write permission and to disallow access to mappings without any access permission.
 26833 Other than these restrictions, implementations may allow access types other than those
 26834 requested by the application. For example, if the application requests only PROT_WRITE, the
 26835 implementation may also allow read access. A call to *mmap()* fails if the implementation cannot
 26836 support allowing all the access requested by the application. For example, some
 26837 implementations cannot support a request for both write access and execute access
 26838 simultaneously. All implementations must support requests for no access, read access, write
 26839 access, and both read and write access. Strictly conforming code must only rely on the required
 26840 checks. These restrictions allow for portability across a wide range of hardware.

26841 The MAP_FIXED address treatment is likely to fail for non-page-aligned values and for certain
 26842 architecture-dependent address ranges. Conforming implementations cannot count on being
 26843 able to choose address values for MAP_FIXED without utilizing non-portable, implementation-
 26844 defined knowledge. Nonetheless, MAP_FIXED is provided as a standard interface conforming
 26845 to existing practice for utilizing such knowledge when it is available.

26846 Similarly, in order to allow implementations that do not support virtual addresses, support for
 26847 directly specifying any mapping addresses via MAP_FIXED is not required and thus a
 26848 conforming application may not count on it.

26849 The MAP_PRIVATE function can be implemented efficiently when memory protection hardware
 26850 is available. When such hardware is not available, implementations can implement such
 26851 “mappings” by simply making a real copy of the relevant data into process private memory,
 26852 though this tends to behave similarly to *read()*.

26853 The function has been defined to allow for many different models of using shared memory.
 26854 However, all uses are not equally portable across all machine architectures. In particular, the
 26855 *mmap()* function allows the system as well as the application to specify the address at which to
 26856 map a specific region of a memory object. The most portable way to use the function is always to
 26857 let the system choose the address, specifying NULL as the value for the argument *addr* and not
 26858 to specify MAP_FIXED.

26859 If it is intended that a particular region of a memory object be mapped at the same address in a
 26860 group of processes (on machines where this is even possible), then MAP_FIXED can be used to
 26861 pass in the desired mapping address. The system can still be used to choose the desired address
 26862 if the first such mapping is made without specifying MAP_FIXED, and then the resulting
 26863 mapping address can be passed to subsequent processes for them to pass in via MAP_FIXED.
 26864 The availability of a specific address range cannot be guaranteed, in general.

26865 The *mmap()* function can be used to map a region of memory that is larger than the current size
 26866 of the object. Memory access within the mapping but beyond the current end of the underlying
 26867 objects may result in SIGBUS signals being sent to the process. The reason for this is that the size
 26868 of the object can be manipulated by other processes and can change at any moment. The
 26869 implementation should tell the application that a memory reference is outside the object where
 26870 this can be detected; otherwise, written data may be lost and read data may not reflect actual
 26871 data in the object.

26872 Note that references beyond the end of the object do not extend the object as the new end cannot
 26873 be determined precisely by most virtual memory hardware. Instead, the size can be directly
 26874 manipulated by *ftruncate()*.

26875 Process memory locking does apply to shared memory regions, and the MEMLOCK_FUTURE
 26876 argument to *mlockall()* can be relied upon to cause new shared memory regions to be
 26877 automatically locked.

26878 Existing implementations of *mmap()* return the value `-1` when unsuccessful. Since the casting of
 26879 this value to type `void *` cannot be guaranteed by the ISO C standard to be distinct from a
 26880 successful value, this volume of IEEE Std 1003.1-200x defines the symbol MAP_FAILED, which a

26881 conforming implementation does not return as the result of a successful call.

26882 FUTURE DIRECTIONS

26883 None.

26884 SEE ALSO

26885 *exec*, *fcntl()*, *fork()*, *lockf()*, *msync()*, *munmap()*, *mprotect()*, *posix_typed_mem_open()*, *shmat()*,
26886 *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-200x, <*sys/mman.h*>

26887 CHANGE HISTORY

26888 First released in Issue 4, Version 2.

26889 Issue 5

26890 Moved from X/OPEN UNIX extension to BASE.

26891 Aligned with *mmap()* in the POSIX Realtime Extension as follows:

- 26892 • The DESCRIPTION is extensively reworded.
- 26893 • The [EAGAIN] and [ENOTSUP] mandatory error conditions are added.
- 26894 • New cases of [ENOMEM] and [ENXIO] are added as mandatory error conditions.
- 26895 • The value returned on failure is the value of the constant MAP_FAILED; this was
26896 previously defined as -1.

26897 Large File Summit extensions are added.

26898 Issue 6

26899 The *mmap()* function is marked as part of the Memory Mapped Files option.

26900 The Open Group Corrigendum U028/6 is applied, changing (**void ***)-1 to MAP_FAILED.

26901 The following new requirements on POSIX implementations derive from alignment with the
26902 Single UNIX Specification:

- 26903 • The DESCRIPTION is updated to describe the use of MAP_FIXED.
- 26904 • The DESCRIPTION is updated to describe the addition of an extra reference to the file
26905 associated with the file descriptor passed to *mmap()*.
- 26906 • The DESCRIPTION is updated to state that there may be implementation-defined limits on
26907 the number of memory regions that can be mapped.
- 26908 • The DESCRIPTION is updated to describe constraints on the alignment and size of the *off*
26909 argument.
- 26910 • The [EINVAL] and [EMFILE] error conditions are added.
- 26911 • The [EOVERFLOW] error condition is added. This change is to support large files.

26912 The following changes are made for alignment with the ISO POSIX-1:1996 standard:

- 26913 • The DESCRIPTION is updated to describe the cases when MAP_PRIVATE and
26914 MAP_FIXED need not be supported.

26915 The following changes are made for alignment with IEEE Std 1003.1j-2000:

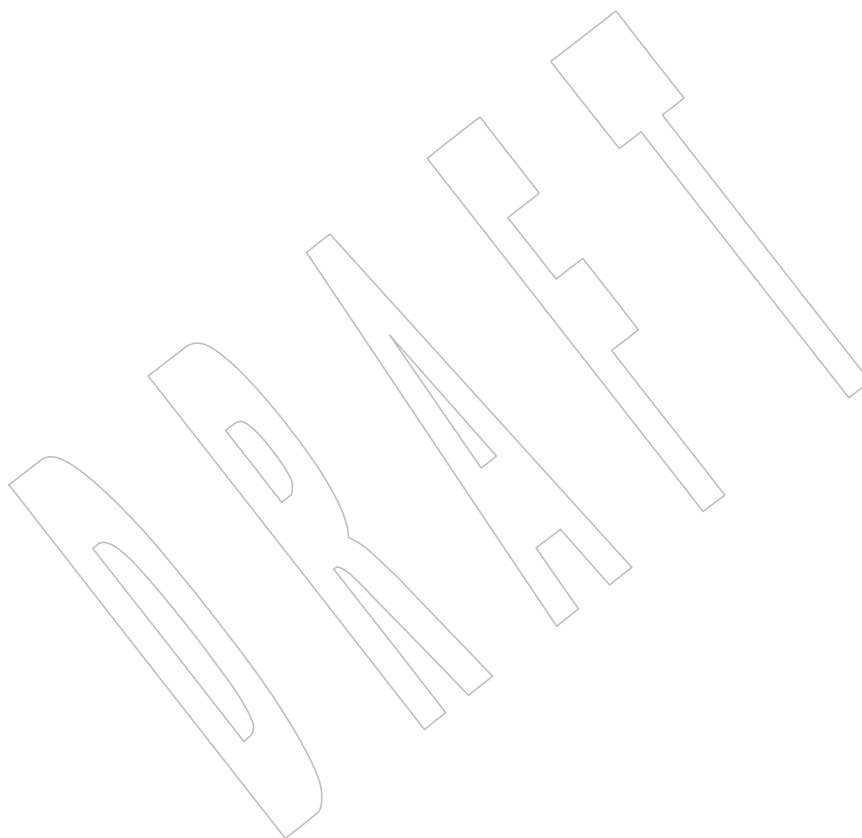
- 26916 • Semantics for typed memory objects are added to the DESCRIPTION.
- 26917 • New [ENOMEM] and [ENXIO] errors are added to the ERRORS section.
- 26918 • The *posix_typed_mem_open()* function is added to the SEE ALSO section.

26919 The normative text is updated to avoid use of the term “must” for application requirements.

26920 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/34 is applied, changing the margin code
26921 in the SYNOPSIS from MF|SHM to MC3 (notation for MF|SHM|TYM).

mmap()

- 26922 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/60 is applied, updating the
26923 DESCRIPTION and ERRORS sections to add the [EINVAL] error when *len* is zero.
- 26924 **Issue 7**
26925 Austin Group Interpretations 1003.1-2001 #078 and #079 are applied, clarifying page alignment
26926 requirements and adding a note about the state of synchronization objects becoming undefined
26927 when a shared region is unmapped.
- 26928 Functionality relating to the Memory Protection and Memory Mapped Files options is moved to
26929 the Base.



26930 **NAME**

26931 modf, modff, modfl — decompose a floating-point number

26932 **SYNOPSIS**

26933 #include <math.h>

26934 double modf(double *x*, double **iptr*);26935 float modff(float *value*, float **iptr*);26936 long double modfl(long double *value*, long double **iptr*);26937 **DESCRIPTION**26938 CX The functionality described on this reference page is aligned with the ISO C standard. Any
26939 conflict between the requirements described here and the ISO C standard is unintentional. This
26940 volume of IEEE Std 1003.1-200x defers to the ISO C standard.26941 These functions shall break the argument *x* into integral and fractional parts, each of which has
26942 the same sign as the argument. It stores the integral part as a **double** (for the *modf()* function), a
26943 **float** (for the *modff()* function), or a **long double** (for the *modfl()* function), in the object pointed
26944 to by *iptr*.26945 **RETURN VALUE**26946 Upon successful completion, these functions shall return the signed fractional part of *x*.26947 MX If *x* is NaN, a NaN shall be returned, and **iptr* shall be set to a NaN.26948 If *x* is ±Inf, ±0 shall be returned, and **iptr* shall be set to ±Inf.26949 **ERRORS**

26950 No errors are defined.

26951 **EXAMPLES**

26952 None.

26953 **APPLICATION USAGE**26954 The *modf()* function computes the function result and **iptr* such that:

26955 a = modf(x, iptr) ;

26956 x == a+*iptr ;

26957 allowing for the usual floating-point inaccuracies.

26958 **RATIONALE**

26959 None.

26960 **FUTURE DIRECTIONS**

26961 None.

26962 **SEE ALSO**26963 *frexp()*, *isnan()*, *ldexp()*, the Base Definitions volume of IEEE Std 1003.1-200x, <math.h>26964 **CHANGE HISTORY**

26965 First released in Issue 1. Derived from Issue 1 of the SVID.

26966 **Issue 5**26967 The DESCRIPTION is updated to indicate how an application should check for an error. This
26968 text was previously published in the APPLICATION USAGE section.

26969

Issue 6

26970

The *modff()* and *modfl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

26971

26972

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

26973

26974

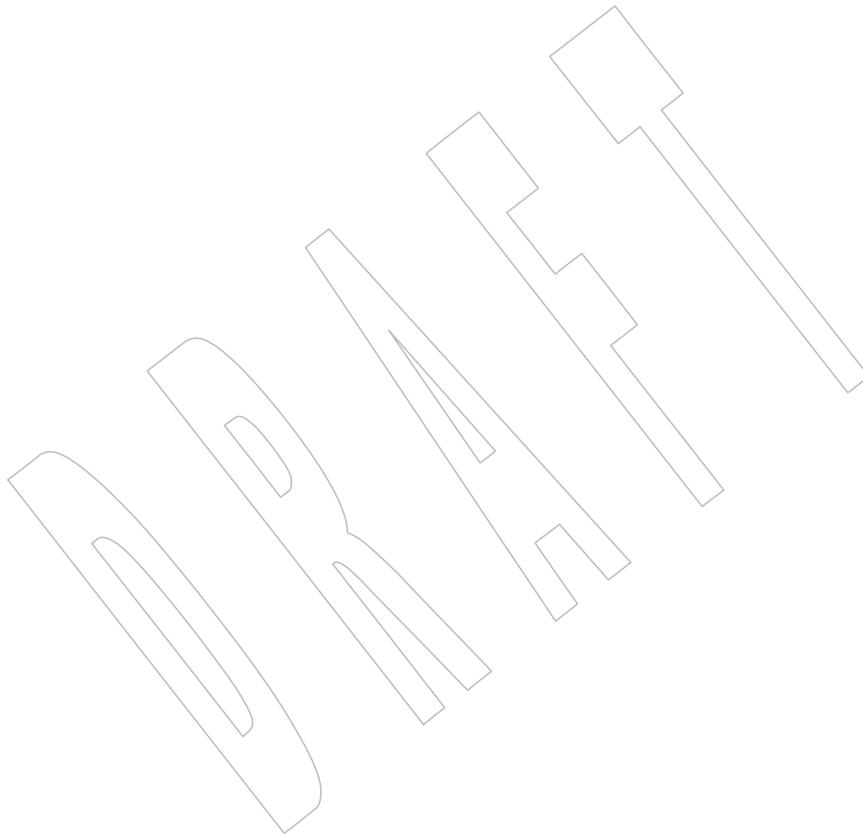
IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

26975

26976

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/35 is applied, correcting the code example in the APPLICATION USAGE section.

26977



26978 **NAME**
 26979 `mprotect` — set protection of memory mapping

26980 **SYNOPSIS**
 26981 `#include <sys/mman.h>`

26982 `int mprotect(void *addr, size_t len, int prot);`

26983 **DESCRIPTION**

26984 The `mprotect()` function shall change the access protections to be that specified by `prot` for those
 26985 whole pages containing any part of the address space of the process starting at address `addr` and
 26986 continuing for `len` bytes. The parameter `prot` determines whether read, write, execute, or some
 26987 combination of accesses are permitted to the data being mapped. The `prot` argument should be
 26988 either `PROT_NONE` or the bitwise-inclusive OR of one or more of `PROT_READ`, `PROT_WRITE`,
 26989 and `PROT_EXEC`.

26990 If an implementation cannot support the combination of access types specified by `prot`, the call to
 26991 `mprotect()` shall fail.

26992 An implementation may permit accesses other than those specified by `prot`; however, no
 26993 implementation shall permit a write to succeed where `PROT_WRITE` has not been set or shall
 26994 permit any access where `PROT_NONE` alone has been set. Implementations shall support at
 26995 least the following values of `prot`: `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, and the bitwise-
 26996 inclusive OR of `PROT_READ` and `PROT_WRITE`. If `PROT_WRITE` is specified, the application
 26997 shall ensure that it has opened the mapped objects in the specified address range with write
 26998 permission, unless `MAP_PRIVATE` was specified in the original mapping, regardless of whether
 26999 the file descriptors used to map the objects have since been closed.

27000 The implementation may require that `addr` be a multiple of the page size as returned by
 27001 `sysconf()`.

27002 The behavior of this function is unspecified if the mapping was not established by a call to
 27003 `mmap()`.

27004 When `mprotect()` fails for reasons other than `[EINVAL]`, the protections on some of the pages in
 27005 the range `[addr,addr+len)` may have been changed.

27006 **RETURN VALUE**

27007 Upon successful completion, `mprotect()` shall return 0; otherwise, it shall return `-1` and set `errno`
 27008 to indicate the error.

27009 **ERRORS**

27010 The `mprotect()` function shall fail if:

27011 `[EACCES]` The `prot` argument specifies a protection that violates the access permission the
 27012 process has to the underlying memory object.

27013 `[EAGAIN]` The `prot` argument specifies `PROT_WRITE` over a `MAP_PRIVATE` mapping
 27014 and there are insufficient memory resources to reserve for locking the private
 27015 page.

27016 `[ENOMEM]` Addresses in the range `[addr,addr+len)` are invalid for the address space of a
 27017 process, or specify one or more pages which are not mapped.

27018 `[ENOMEM]` The `prot` argument specifies `PROT_WRITE` on a `MAP_PRIVATE` mapping, and
 27019 it would require more space than the system is able to supply for locking the
 27020 private pages, if required.

mprotect()

27021 [ENOTSUP] The implementation does not support the combination of accesses requested
27022 in the *prot* argument.

27023 The *mprotect()* function may fail if:

27024 [EINVAL] The *addr* argument is not a multiple of the page size as returned by *sysconf()*.

EXAMPLES

27025 None.
27026

APPLICATION USAGE

27027 Most implementations require that *addr* is a multiple of the page size as returned by *sysconf()*.
27028

RATIONALE

27029 None.
27030

FUTURE DIRECTIONS

27031 None.
27032

SEE ALSO

27033 *mmap()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-200x, <sys/mman.h>
27034

CHANGE HISTORY

27035 First released in Issue 4, Version 2.
27036

Issue 5

27037 Moved from X/OPEN UNIX extension to BASE.
27038

27039 Aligned with *mprotect()* in the POSIX Realtime Extension as follows:

- 27040 • The DESCRIPTION is largely reworded.
- 27041 • [ENOTSUP] and a second form of [ENOMEM] are added as mandatory error conditions.
- 27042 • [EAGAIN] is moved from the optional to the mandatory error conditions.

Issue 6

27043 The *mprotect()* function is marked as part of the Memory Protection option.
27044

27045 The following new requirements on POSIX implementations derive from alignment with the
27046 Single UNIX Specification:

- 27047 • The DESCRIPTION is updated to state that implementations require *addr* to be a multiple
27048 of the page size as returned by *sysconf()*.
- 27049 • The [EINVAL] error condition is added.

27050 The normative text is updated to avoid use of the term “must” for application requirements.

Issue 7

27051 SD5-XSH-ERN-22 is applied, deleting erroneous APPLICATION USAGE.
27052

27053 Austin Group Interpretation 1003.1-2001 #078 is applied, clarifying page alignment
27054 requirements.

27055 The *mprotect()* function is moved from the Memory Protection option to the Base.

27056 **NAME**
 27057 `mq_close` — close a message queue (**REALTIME**)

27058 **SYNOPSIS**

27059 MSG `#include <mqueue.h>`
 27060 `int mq_close(mqd_t mqdes);`

27061 **DESCRIPTION**

27062 The `mq_close()` function shall remove the association between the message queue descriptor,
 27063 `mqdes`, and its message queue. The results of using this message queue descriptor after successful
 27064 return from this `mq_close()`, and until the return of this message queue descriptor from a
 27065 subsequent `mq_open()`, are undefined.

27066 If the process has successfully attached a notification request to the message queue via this
 27067 `mqdes`, this attachment shall be removed, and the message queue is available for another process
 27068 to attach for notification.

27069 **RETURN VALUE**

27070 Upon successful completion, the `mq_close()` function shall return a value of zero; otherwise, the
 27071 function shall return a value of `-1` and set `errno` to indicate the error.

27072 **ERRORS**

27073 The `mq_close()` function shall fail if:

27074 [EBADF] The `mqdes` argument is not a valid message queue descriptor.

27075 **EXAMPLES**

27076 None.

27077 **APPLICATION USAGE**

27078 None.

27079 **RATIONALE**

27080 None.

27081 **FUTURE DIRECTIONS**

27082 None.

27083 **SEE ALSO**

27084 `mq_open()`, `mq_unlink()`, `msgctl()`, `msgget()`, `msgrcv()`, `msgsnd()`, the Base Definitions volume of
 27085 IEEE Std 1003.1-200x, `<mqueue.h>`

27086 **CHANGE HISTORY**

27087 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

27088 **Issue 6**

27089 The `mq_close()` function is marked as part of the Message Passing option.

27090 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 27091 implementation does not support the Message Passing option.

27092 **NAME**27093 `mq_getattr` — get message queue attributes (**REALTIME**)27094 **SYNOPSIS**

```
27095 MSG #include <mqueue.h>
27096 int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);
```

27097 **DESCRIPTION**

27098 The `mq_getattr()` function shall obtain status information and attributes of the message queue
 27099 and the open message queue description associated with the message queue descriptor.

27100 The `mqdes` argument specifies a message queue descriptor.

27101 The results shall be returned in the **mq_attr** structure referenced by the `mqstat` argument.

27102 Upon return, the following members shall have the values associated with the open message
 27103 queue description as set when the message queue was opened and as modified by subsequent
 27104 `mq_setattr()` calls: `mq_flags`.

27105 The following attributes of the message queue shall be returned as set at message queue
 27106 creation: `mq_maxmsg`, `mq_msgsize`.

27107 Upon return, the following members within the **mq_attr** structure referenced by the `mqstat`
 27108 argument shall be set to the current state of the message queue:

27109 `mq_curmsgs` The number of messages currently on the queue.

27110 **RETURN VALUE**

27111 Upon successful completion, the `mq_getattr()` function shall return zero. Otherwise, the function
 27112 shall return `-1` and set `errno` to indicate the error.

27113 **ERRORS**

27114 The `mq_getattr()` function may fail if:

27115 [EBADF] The `mqdes` argument is not a valid message queue descriptor.

27116 **EXAMPLES**

27117 None.

27118 **APPLICATION USAGE**

27119 None.

27120 **RATIONALE**

27121 None.

27122 **FUTURE DIRECTIONS**

27123 None.

27124 **SEE ALSO**

27125 `mq_open()`, `mq_send()`, `mq_setattr()`, `mq_timedsend()`, `msgctl()`, `msgget()`, `msgrcv()`, `msgsnd()`, the
 27126 Base Definitions volume of IEEE Std 1003.1-200x, **<mqueue.h>**

27127 **CHANGE HISTORY**

27128 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

27129

Issue 6

27130

The `mq_getattr()` function is marked as part of the Message Passing option.

27131

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

27132

27133

The `mq_timedsend()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

27134

27135

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/61 is applied, updating the ERRORS section to change the [EBADF] error from mandatory to optional.

27136



27137 **NAME**27138 `mq_notify` — notify process that a message is available (**REALTIME**)27139 **SYNOPSIS**

```
27140 MSG #include <mqueue.h>
27141 int mq_notify(mqd_t mqdes, const struct sigevent *notification);
```

27142 **DESCRIPTION**

27143 If the argument *notification* is not NULL, this function shall register the calling process to be
 27144 notified of message arrival at an empty message queue associated with the specified message
 27145 queue descriptor, *mqdes*. The notification specified by the *notification* argument shall be sent to
 27146 the process when the message queue transitions from empty to non-empty. At any time, only
 27147 one process may be registered for notification by a message queue. If the calling process or any
 27148 other process has already registered for notification of message arrival at the specified message
 27149 queue, subsequent attempts to register for that message queue shall fail.

27150 If *notification* is NULL and the process is currently registered for notification by the specified
 27151 message queue, the existing registration shall be removed.

27152 When the notification is sent to the registered process, its registration shall be removed. The
 27153 message queue shall then be available for registration.

27154 If a process has registered for notification of message arrival at a message queue and some
 27155 thread is blocked in *mq_receive()* or *mq_timedreceive()* waiting to receive a message when a
 27156 message arrives at the queue, the arriving message shall satisfy the appropriate *mq_receive()* or
 27157 *mq_timedreceive()*, respectively. The resulting behavior is as if the message queue remains empty,
 27158 and no notification shall be sent.

27159 **RETURN VALUE**

27160 Upon successful completion, the *mq_notify()* function shall return a value of zero; otherwise, the
 27161 function shall return a value of -1 and set *errno* to indicate the error.

27162 **ERRORS**

27163 The *mq_notify()* function shall fail if:

- | | | |
|-------|--|---|
| 27164 | [EBADF] | The <i>mqdes</i> argument is not a valid message queue descriptor. |
| 27165 | [EBUSY] | A process is already registered for notification by the message queue. |
| 27166 | The <i>mq_notify()</i> function may fail if: | |
| 27167 | [EINVAL] | The <i>notification</i> argument is NULL and the process is currently not registered. |

27168 **EXAMPLES**

27169 None.

27170 **APPLICATION USAGE**

27171 None.

27172 **RATIONALE**

27173 None.

27174 **FUTURE DIRECTIONS**

27175 None.

27176
27177
27178

27179
27180

27181
27182

27183
27184

27185
27186

27187
27188
27189

SEE ALSO

mq_open(), *mq_send()*, *mq_receive()*, *mq_timedsend()*, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*, the Base Definitions volume of IEEE Std 1003.1-200x, <mqqueue.h>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mq_notify()* function is marked as part of the Message Passing option.

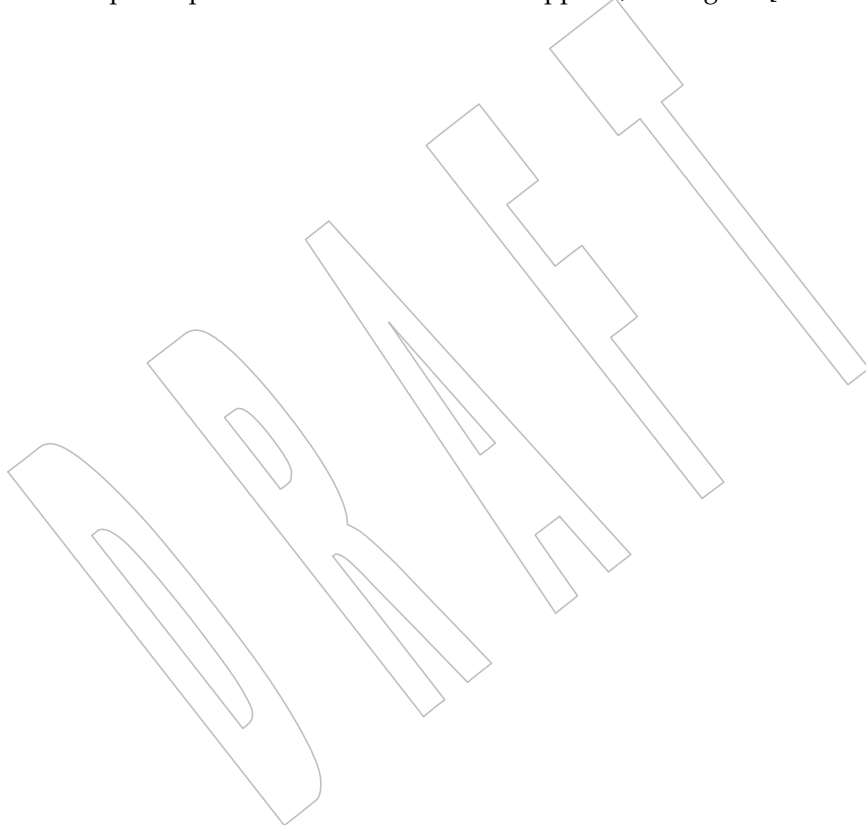
The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

The *mq_timedsend()* function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

Issue 7

SD5-XSH-ERN-38 is applied, adding the *mq_timedreceive()* function to the DESCRIPTION.

Austin Group Interpretation 1003.1-2001 #032 is applied, adding the [EINVAL] error.



NAME

mq_open — open a message queue (**REALTIME**)

SYNOPSIS

MSG



27235 O_EXCL. Otherwise, a message queue shall be created without any messages
 27236 in it. The user ID of the message queue shall be set to the effective user ID of
 27237 the process, and the group ID of the message queue shall be set to the effective
 27238 group ID of the process. The permission bits of the message queue shall be set
 27239 to the value of the *mode* argument, except those set in the file mode creation
 27240 mask of the process. When bits in *mode* other than the file permission bits are
 27241 specified, the effect is unspecified. If *attr* is NULL, the message queue shall be
 27242 created with implementation-defined default message queue attributes. If *attr*
 27243 is non-NULL and the calling process has the appropriate privilege on *name*,
 27244 the message queue *mq_maxmsg* and *mq_msgsize* attributes shall be set to the
 27245 values of the corresponding members in the **mq_attr** structure referred to by
 27246 *attr*. If *attr* is non-NULL, but the calling process does not have the appropriate
 27247 privilege on *name*, the *mq_open()* function shall fail and return an error
 27248 without creating the message queue.

27249 O_EXCL If O_EXCL and O_CREAT are set, *mq_open()* shall fail if the message queue
 27250 *name* exists. The check for the existence of the message queue and the creation
 27251 of the message queue if it does not exist shall be atomic with respect to other
 27252 threads executing *mq_open()* naming the same *name* with O_EXCL and
 27253 O_CREAT set. If O_EXCL is set and O_CREAT is not set, the result is
 27254 undefined.

27255 O_NONBLOCK Determines whether an *mq_send()* or *mq_receive()* waits for resources or
 27256 messages that are not currently available, or fails with *errno* set to [EAGAIN];
 27257 see *mq_send()* and *mq_receive()* for details.

27258 The *mq_open()* function does not add or remove messages from the queue.

27259 RETURN VALUE

27260 Upon successful completion, the function shall return a message queue descriptor; otherwise,
 27261 the function shall return (**mqd_t**)−1 and set *errno* to indicate the error.

27262 ERRORS

27263 The *mq_open()* function shall fail if:

27264 [EACCES] The message queue exists and the permissions specified by *oflag* are denied, or
 27265 the message queue does not exist and permission to create the message queue
 27266 is denied.

27267 [EEXIST] O_CREAT and O_EXCL are set and the named message queue already exists.

27268 [EINTR] The *mq_open()* function was interrupted by a signal.

27269 [EINVAL] The *mq_open()* function is not supported for the given name.

27270 [EINVAL] O_CREAT was specified in *oflag*, the value of *attr* is not NULL, and either
 27271 *mq_maxmsg* or *mq_msgsize* was less than or equal to zero.

27272 [EMFILE] Too many message queue descriptors or file descriptors are currently in use by
 27273 this process.

27274 [ENFILE] Too many message queues are currently open in the system.

27275 [ENOENT] O_CREAT is not set and the named message queue does not exist.

27276 [ENOSPC] There is insufficient space for the creation of the new message queue.

27277 If any of the following conditions occur, the *mq_open()* function may return (**mqd_t**)−1 and set
 27278 *errno* to the corresponding value.

mq_open()

27279 [ENAMETOOLONG]
 27280 The length of the *name* argument exceeds `{_POSIX_PATH_MAX}` on systems
 27281 XSI that do not support the XSI option or exceeds `{_XOPEN_PATH_MAX}` on XSI
 27282 systems, or has a pathname component that is longer than
 27283 XSI `{_POSIX_NAME_MAX}` on systems that do not support the XSI option or
 27284 longer than `{_XOPEN_NAME_MAX}` on XSI systems.

EXAMPLES

27285 None.
 27286

APPLICATION USAGE

27287 None.
 27288

RATIONALE

27289 None.
 27290

FUTURE DIRECTIONS

27291 None.
 27292

SEE ALSO

27293 *mq_close()*, *mq_getattr()*, *mq_receive()*, *mq_send()*, *mq_setattr()*, *mq_timedreceive()*, *mq_timedsend()*,
 27294 *mq_unlink()*, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*, the Base Definitions volume of
 27295 IEEE Std 1003.1-200x, <**mqqueue.h**>
 27296

CHANGE HISTORY

27297 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.
 27298

Issue 6

27299 The *mq_open()* function is marked as part of the Message Passing option.
 27300

27301 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 27302 implementation does not support the Message Passing option.

27303 The *mq_timedreceive()* and *mq_timedsend()* functions are added to the SEE ALSO section for
 27304 alignment with IEEE Std 1003.1d-1999.

27305 The DESCRIPTION of O_EXCL is updated in response to IEEE PASC Interpretation 1003.1c #48.

27306 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/62 is applied, updating the description of
 27307 the permission bits in the DESCRIPTION section. The change is made for consistency with the
 27308 *shm_open()* and *sem_open()* functions.

Issue 7

27309 Austin Group Interpretation 1003.1-2001 #077 is applied, clarifying the *name* argument and
 27310 changing [ENAMETOOLONG] from a “shall fail” to a “may fail” error.
 27311

27312 **NAME**27313 mq_receive, mq_timedreceive — receive a message from a message queue (**REALTIME**)27314 **SYNOPSIS**

```

27315 MSG #include <mqqueue.h>
27316
27316 ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
27317 unsigned *msg_prio);
27318
27318 #include <mqqueue.h>
27319 #include <time.h>
27320
27320 ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr,
27321 size_t msg_len, unsigned *restrict msg_prio,
27322 const struct timespec *restrict abs_timeout);

```

27323 **DESCRIPTION**

27324 The *mq_receive()* function shall receive the oldest of the highest priority message(s) from the
 27325 message queue specified by *mqdes*. If the size of the buffer in bytes, specified by the *msg_len*
 27326 argument, is less than the *mq_msgsize* attribute of the message queue, the function shall fail and
 27327 return an error. Otherwise, the selected message shall be removed from the queue and copied to
 27328 the buffer pointed to by the *msg_ptr* argument.

27329 If the value of *msg_len* is greater than {SSIZE_MAX}, the result is implementation-defined.

27330 If the argument *msg_prio* is not NULL, the priority of the selected message shall be stored in the
 27331 location referenced by *msg_prio*.

27332 If the specified message queue is empty and O_NONBLOCK is not set in the message queue
 27333 description associated with *mqdes*, *mq_receive()* shall block until a message is enqueued on the
 27334 message queue or until *mq_receive()* is interrupted by a signal. If more than one thread is waiting
 27335 to receive a message when a message arrives at an empty queue and the Priority Scheduling
 27336 option is supported, then the thread of highest priority that has been waiting the longest shall be
 27337 selected to receive the message. Otherwise, it is unspecified which waiting thread receives the
 27338 message. If the specified message queue is empty and O_NONBLOCK is set in the message
 27339 queue description associated with *mqdes*, no message shall be removed from the queue, and
 27340 *mq_receive()* shall return an error.

27341 The *mq_timedreceive()* function shall receive the oldest of the highest priority messages from the
 27342 message queue specified by *mqdes* as described for the *mq_receive()* function. However, if
 27343 O_NONBLOCK was not specified when the message queue was opened via the *mq_open()*
 27344 function, and no message exists on the queue to satisfy the receive, the wait for such a message
 27345 shall be terminated when the specified timeout expires. If O_NONBLOCK is set, this function is
 27346 equivalent to *mq_receive()*.

27347 The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the
 27348 clock on which timeouts are based (that is, when the value of that clock equals or exceeds
 27349 *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time
 27350 of the call.

27351 The timeout shall be based on the CLOCK_REALTIME clock. The resolution of the timeout shall
 27352 be the resolution of the clock on which it is based. The *timespec* argument is defined in the
 27353 **<time.h>** header.

27354 Under no circumstance shall the operation fail with a timeout if a message can be removed from
 27355 the message queue immediately. The validity of the *abs_timeout* parameter need not be checked
 27356 if a message can be removed from the message queue immediately.

mq_receive()27357 **RETURN VALUE**

27358 Upon successful completion, the *mq_receive()* and *mq_timedreceive()* functions shall return the
 27359 length of the selected message in bytes and the message shall be removed from the queue.
 27360 Otherwise, no message shall be removed from the queue, the functions shall return a value of -1,
 27361 and set *errno* to indicate the error.

27362 **ERRORS**

27363 The *mq_receive()* and *mq_timedreceive()* functions shall fail if:

27364 [EAGAIN] O_NONBLOCK was set in the message description associated with *mqdes*, and
 27365 the specified message queue is empty.

27366 [EBADF] The *mqdes* argument is not a valid message queue descriptor open for reading.

27367 [EMSGSIZE] The specified message buffer size, *msg_len*, is less than the message size
 27368 attribute of the message queue.

27369 [EINTR] The *mq_receive()* or *mq_timedreceive()* operation was interrupted by a signal.

27370 [EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter
 27371 specified a nanoseconds field value less than zero or greater than or equal to
 27372 1 000 million.

27373 [ETIMEDOUT] The O_NONBLOCK flag was not set when the message queue was opened,
 27374 but no message arrived on the queue before the specified timeout expired.

27375 The *mq_receive()* and *mq_timedreceive()* functions may fail if:

27376 [EBADMSG] The implementation has detected a data corruption problem with the
 27377 message.

27378 **EXAMPLES**

27379 None.

27380 **APPLICATION USAGE**

27381 None.

27382 **RATIONALE**

27383 None.

27384 **FUTURE DIRECTIONS**

27385 None.

27386 **SEE ALSO**

27387 *mq_open()*, *mq_send()*, *mq_timedsend()*, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*, *time()*, the Base
 27388 Definitions volume of IEEE Std 1003.1-200x, <mqqueue.h>, <time.h>

27389 **CHANGE HISTORY**

27390 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

27391 **Issue 6**

27392 The *mq_receive()* function is marked as part of the Message Passing option.

27393 The Open Group Corrigendum U021/4 is applied. The DESCRIPTION is changed to refer to
 27394 *msg_len* rather than *maxsize*.

27395 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 27396 implementation does not support the Message Passing option.

27397 The following new requirements on POSIX implementations derive from alignment with the
 27398 Single UNIX Specification:

27399
27400
27401
27402
27403
27404
27405
27406
27407
27408
27409

- In this function it is possible for the return value to exceed the range of the type `ssize_t` (since `size_t` has a larger range of positive values than `ssize_t`). A sentence restricting the size of the `size_t` object is added to the description to resolve this conflict.

The `mq_timedreceive()` function is added for alignment with IEEE Std 1003.1d-1999.

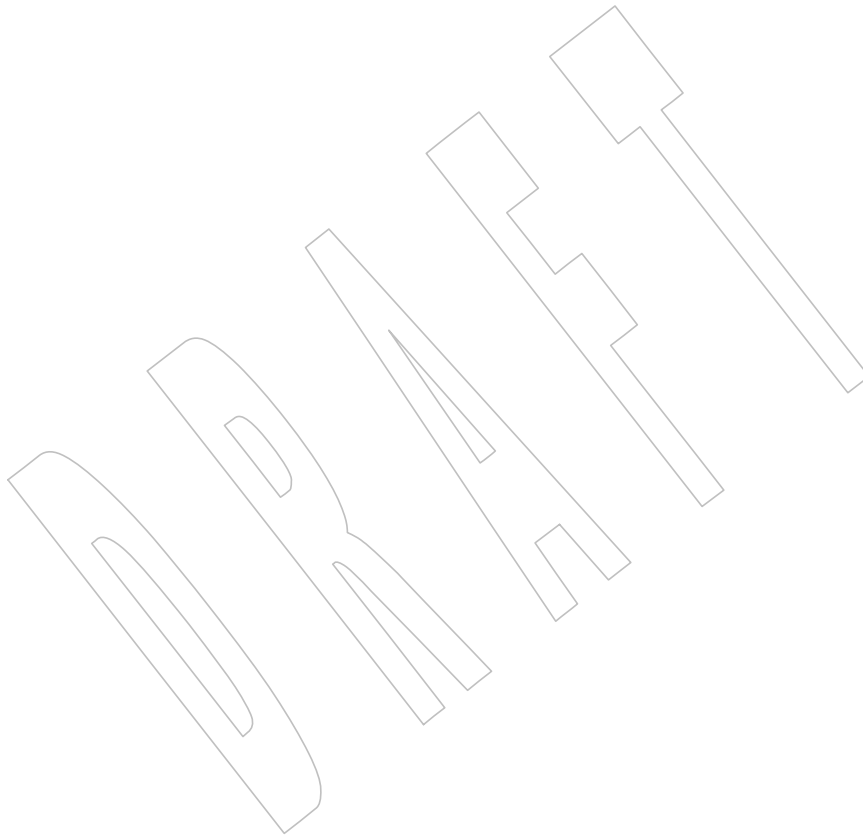
The `restrict` keyword is added to the `mq_timedreceive()` prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE PASC Interpretation 1003.1 #109 is applied, correcting the return type for `mq_timedreceive()` from `int` to `ssize_t`.

Issue 7

The `mq_timedreceive()` function is moved from the Timeouts option to the Base.

Functionality relating to the Timmers option is moved to the Base.



27410 **NAME**27411 mq_send, mq_timedsend — send a message to a message queue (**REALTIME**)27412 **SYNOPSIS**

```

27413 MSG #include <mqqueue.h>
27414
27414 int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
27415            unsigned msg_prio);
27416
27416 #include <mqqueue.h>
27417 #include <time.h>
27418
27418 int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
27419                unsigned msg_prio, const struct timespec *abs_timeout);

```

27420 **DESCRIPTION**

27421 The *mq_send()* function shall add the message pointed to by the argument *msg_ptr* to the
 27422 message queue specified by *mqdes*. The *msg_len* argument specifies the length of the message, in
 27423 bytes, pointed to by *msg_ptr*. The value of *msg_len* shall be less than or equal to the *mq_msgsize*
 27424 attribute of the message queue, or *mq_send()* shall fail.

27425 If the specified message queue is not full, *mq_send()* shall behave as if the message is inserted
 27426 into the message queue at the position indicated by the *msg_prio* argument. A message with a
 27427 larger numeric value of *msg_prio* shall be inserted before messages with lower values of
 27428 *msg_prio*. A message shall be inserted after other messages in the queue, if any, with equal
 27429 *msg_prio*. The value of *msg_prio* shall be less than {MQ_PRIO_MAX}.

27430 If the specified message queue is full and O_NONBLOCK is not set in the message queue
 27431 description associated with *mqdes*, *mq_send()* shall block until space becomes available to
 27432 enqueue the message, or until *mq_send()* is interrupted by a signal. If more than one thread is
 27433 waiting to send when space becomes available in the message queue and the Priority Scheduling
 27434 option is supported, then the thread of the highest priority that has been waiting the longest
 27435 shall be unblocked to send its message. Otherwise, it is unspecified which waiting thread is
 27436 unblocked. If the specified message queue is full and O_NONBLOCK is set in the message
 27437 queue description associated with *mqdes*, the message shall not be queued and *mq_send()* shall
 27438 return an error.

27439 The *mq_timedsend()* function shall add a message to the message queue specified by *mqdes* in the
 27440 manner defined for the *mq_send()* function. However, if the specified message queue is full and
 27441 O_NONBLOCK is not set in the message queue description associated with *mqdes*, the wait for
 27442 sufficient room in the queue shall be terminated when the specified timeout expires. If
 27443 O_NONBLOCK is set in the message queue description, this function shall be equivalent to
 27444 *mq_send()*.

27445 The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by
 27446 the clock on which timeouts are based (that is, when the value of that clock equals or exceeds
 27447 *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time
 27448 of the call.

27449 The timeout shall be based on the CLOCK_REALTIME clock. The resolution of the timeout shall
 27450 be the resolution of the clock on which it is based. The *timespec* argument is defined in the
 27451 **<time.h>** header.

27452 Under no circumstance shall the operation fail with a timeout if there is sufficient room in the
 27453 queue to add the message immediately. The validity of the *abs_timeout* parameter need not be
 27454 checked when there is sufficient room in the queue.

27455 **RETURN VALUE**

27456 Upon successful completion, the `mq_send()` and `mq_timedsend()` functions shall return a value of
 27457 zero. Otherwise, no message shall be enqueued, the functions shall return `-1`, and `errno` shall be
 27458 set to indicate the error.

27459 **ERRORS**

27460 The `mq_send()` and `mq_timedsend()` functions shall fail if:

- 27461 [EAGAIN] The `O_NONBLOCK` flag is set in the message queue description associated
 27462 with `mqdes`, and the specified message queue is full.
- 27463 [EBADF] The `mqdes` argument is not a valid message queue descriptor open for writing.
- 27464 [EINTR] A signal interrupted the call to `mq_send()` or `mq_timedsend()`.
- 27465 [EINVAL] The value of `msg_prio` was outside the valid range.
- 27466 [EINVAL] The process or thread would have blocked, and the `abs_timeout` parameter
 27467 specified a nanoseconds field value less than zero or greater than or equal to
 27468 1 000 million.
- 27469 [EMSGSIZE] The specified message length, `msg_len`, exceeds the message size attribute of
 27470 the message queue.
- 27471 [ETIMEDOUT] The `O_NONBLOCK` flag was not set when the message queue was opened,
 27472 but the timeout expired before the message could be added to the queue.

27473 **EXAMPLES**

27474 None.

27475 **APPLICATION USAGE**

27476 The value of the symbol `{MQ_PRIO_MAX}` limits the number of priority levels supported by the
 27477 application. Message priorities range from 0 to `{MQ_PRIO_MAX}-1`.

27478 **RATIONALE**

27479 None.

27480 **FUTURE DIRECTIONS**

27481 None.

27482 **SEE ALSO**

27483 `mq_open()`, `mq_receive()`, `mq_setattr()`, `mq_timedreceive()`, `time()`, the Base Definitions volume of
 27484 IEEE Std 1003.1-200x, `<mqqueue.h>`, `<time.h>`

27485 **CHANGE HISTORY**

27486 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

27487 **Issue 6**

27488 The `mq_send()` function is marked as part of the Message Passing option.

27489 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 27490 implementation does not support the Message Passing option.

27491 The `mq_timedsend()` function is added for alignment with IEEE Std 1003.1d-1999.

27492 **Issue 7**

27493 The `mq_timedsend()` function is moved from the Timeouts option to the Base.

27494 Functionality relating to the Timers option is moved to the Base.

27495 **NAME**
 27496 `mq_setattr` — set message queue attributes (**REALTIME**)

27497 **SYNOPSIS**

```
27498 MSG #include <mqqueue.h>
27499
27499 int mq_setattr(mqd_t mqdes, const struct mq_attr *restrict mqstat,
27500               struct mq_attr *restrict omqstat);
```

27501 **DESCRIPTION**

27502 The `mq_setattr()` function shall set attributes associated with the open message queue
 27503 description referenced by the message queue descriptor specified by `mqdes`.

27504 The message queue attributes corresponding to the following members defined in the **mq_attr**
 27505 structure shall be set to the specified values upon successful completion of `mq_setattr()`:

27506 `mq_flags` The value of this member is the bitwise-logical OR of zero or more of
 27507 `O_NONBLOCK` and any implementation-defined flags.

27508 The values of the `mq_maxmsg`, `mq_msgsize`, and `mq_curmsgs` members of the **mq_attr** structure
 27509 shall be ignored by `mq_setattr()`.

27510 If `omqstat` is non-NULL, the `mq_setattr()` function shall store, in the location referenced by
 27511 `omqstat`, the previous message queue attributes and the current queue status. These values shall
 27512 be the same as would be returned by a call to `mq_getattr()` at that point.

27513 **RETURN VALUE**

27514 Upon successful completion, the function shall return a value of zero and the attributes of the
 27515 message queue shall have been changed as specified.

27516 Otherwise, the message queue attributes shall be unchanged, and the function shall return a
 27517 value of `-1` and set `errno` to indicate the error.

27518 **ERRORS**

27519 The `mq_setattr()` function shall fail if:

27520 [EBADF] The `mqdes` argument is not a valid message queue descriptor.

27521 **EXAMPLES**

27522 None.

27523 **APPLICATION USAGE**

27524 None.

27525 **RATIONALE**

27526 None.

27527 **FUTURE DIRECTIONS**

27528 None.

27529 **SEE ALSO**

27530 `mq_open()`, `mq_send()`, `mq_timedsend()`, `msgctl()`, `msgget()`, `msgrcv()`, `msgsnd()`, the Base
 27531 Definitions volume of IEEE Std 1003.1-200x, **<mqqueue.h>**

27532 **CHANGE HISTORY**

27533 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

27534

Issue 6

27535

The `mq_setattr()` function is marked as part of the Message Passing option.

27536

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

27537

27538

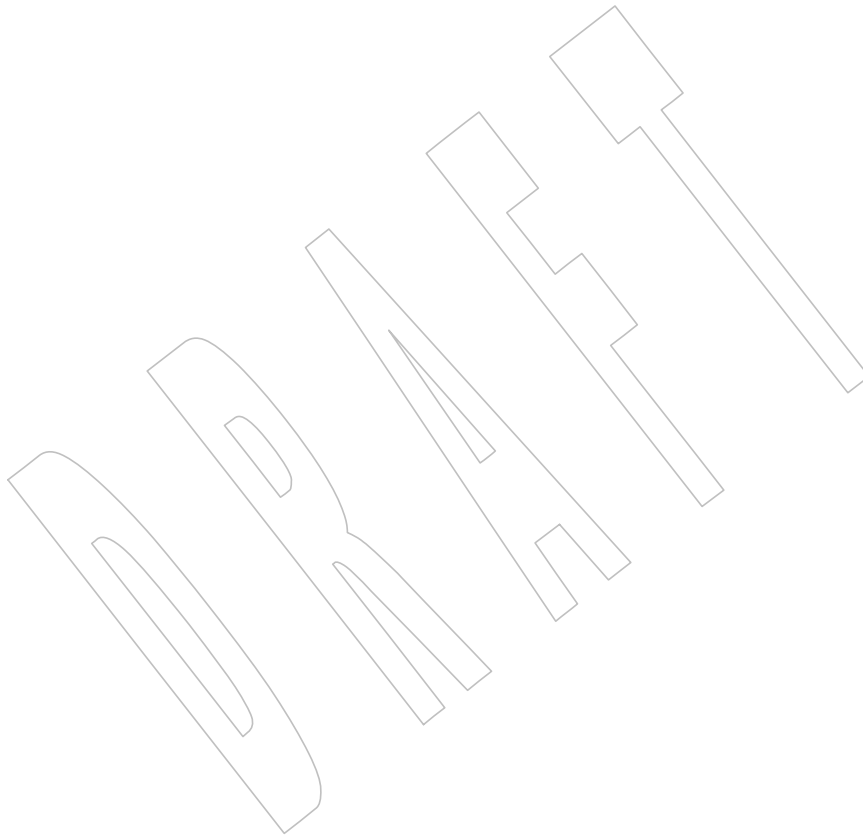
The `mq_timedsend()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

27539

27540

The **restrict** keyword is added to the `mq_setattr()` prototype for alignment with the ISO/IEC 9899:1999 standard.

27541



27542 **NAME**27543 mq_timedreceive — receive a message from a message queue (**ADVANCED REALTIME**)27544 **SYNOPSIS**

```
27545 MSG #include <mqueue.h>  
27546 #include <time.h>  
  
27547 ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr,  
27548 size_t msg_len, unsigned *restrict msg_prio,  
27549 const struct timespec *restrict abs_timeout);
```

27550 **DESCRIPTION**27551 Refer to *mq_receive()*.

27552 **NAME**
27553 mq_timedsend — send a message to a message queue (**ADVANCED REALTIME**)

27554 **SYNOPSIS**

```
27555 MSG #include <mqueue.h>  
27556 #include <time.h>  
  
27557 int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,  
27558 unsigned msg_prio, const struct timespec *abs_timeout);
```

27559 **DESCRIPTION**

27560 Refer to [mq_send\(\)](#).

27561 **NAME**
 27562 `mq_unlink` — remove a message queue (**REALTIME**)

27563 **SYNOPSIS**

27564 MSG `#include <mqqueue.h>`
 27565 `int mq_unlink(const char *name);`

27566 **DESCRIPTION**

27567 The `mq_unlink()` function shall remove the message queue named by the pathname *name*. After
 27568 a successful call to `mq_unlink()` with *name*, a call to `mq_open()` with *name* shall fail if the flag
 27569 `O_CREAT` is not set in *flags*. If one or more processes have the message queue open when
 27570 `mq_unlink()` is called, destruction of the message queue shall be postponed until all references to
 27571 the message queue have been closed.

27572 Calls to `mq_open()` to recreate the message queue may fail until the message queue is actually
 27573 removed. However, the `mq_unlink()` call need not block until all references have been closed; it
 27574 may return immediately.

27575 **RETURN VALUE**

27576 Upon successful completion, the function shall return a value of zero. Otherwise, the named
 27577 message queue shall be unchanged by this function call, and the function shall return a value of
 27578 `-1` and set *errno* to indicate the error.

27579 **ERRORS**

27580 The `mq_unlink()` function shall fail if:

27581 `[EACCES]` Permission is denied to unlink the named message queue.

27582 `[ENOENT]` The named message queue does not exist.

27583 The `mq_unlink()` function may fail if:

27584 `[ENAMETOOLONG]`

27585 The length of the *name* argument exceeds `{_POSIX_PATH_MAX}` on systems
 27586 XSI that do not support the XSI option `or exceeds {_XOPEN_PATH_MAX}` on XSI
 27587 systems, `or has a pathname component that is longer than`
 27588 XSI `{_POSIX_NAME_MAX}` on systems that do not support the XSI option `or`
 27589 `longer than {_XOPEN_NAME_MAX}` on XSI systems. A call to `mq_unlink()`
 27590 with a *name* argument that contains the same message queue name as was
 27591 previously used in a successful `mq_open()` call shall not give an
 27592 `[ENAMETOOLONG]` error.

27593 **EXAMPLES**

27594 None.

27595 **APPLICATION USAGE**

27596 None.

27597 **RATIONALE**

27598 None.

27599 **FUTURE DIRECTIONS**

27600 None.

27601

SEE ALSO

27602

mq_close(), *mq_open()*, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<mqqueue.h>**

27603

27604

CHANGE HISTORY

27605

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

27606

Issue 6

27607

The *mq_unlink()* function is marked as part of the Message Passing option.

27608

The Open Group Corrigendum U021/5 is applied, clarifying that upon unsuccessful completion, the named message queue is unchanged by this function.

27609

27610

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

27611

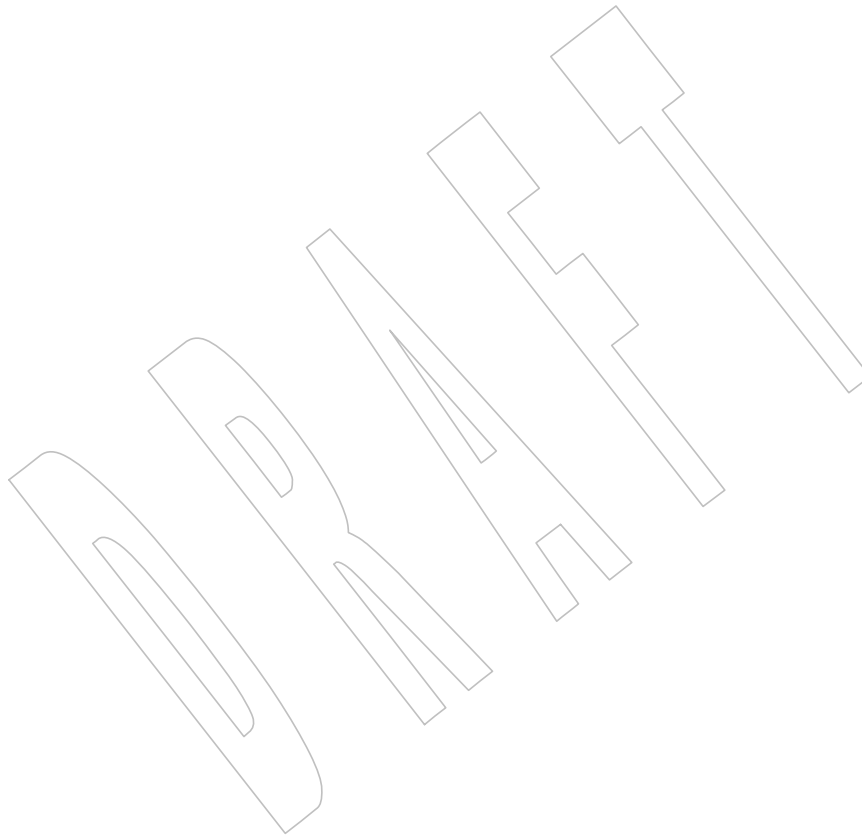
27612

Issue 7

27613

Austin Group Interpretation 1003.1-2001 #077 is applied, changing [ENAMETOOLONG] from a “shall fail” to a “may fail” error .

27614



mrnd48()

27615 **NAME**
27616 `mrnd48` — generate uniformly distributed pseudo-random signed long integers

SYNOPSIS

27617 XSI `#include <stdlib.h>`
27618 `long mrnd48(void);`

DESCRIPTION

27620 Refer to *drand48()*.
27621

27622 **NAME**

27623 msgctl — XSI message control operations

27624 **SYNOPSIS**

```
27625 XSI #include <sys/msg.h>
27626 int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

27627 **DESCRIPTION**

27628 The *msgctl()* function operates on XSI message queues (see the Base Definitions volume of
 27629 IEEE Std 1003.1-200x, Section 3.224, Message Queue). It is unspecified whether this function
 27630 interoperates with the realtime interprocess communication facilities defined in [Section 2.8](#) (on
 27631 page 40).

27632 The *msgctl()* function shall provide message control operations as specified by *cmd*. The
 27633 following values for *cmd*, and the message control operations they specify, are:

27634 **IPC_STAT** Place the current value of each member of the **msqid_ds** data structure
 27635 associated with *msqid* into the structure pointed to by *buf*. The contents of this
 27636 structure are defined in **<sys/msg.h>**.

27637 **IPC_SET** Set the value of the following members of the **msqid_ds** data structure
 27638 associated with *msqid* to the corresponding value found in the structure
 27639 pointed to by *buf*:

```
27640 msg_perm.uid
27641 msg_perm.gid
27642 msg_perm.mode
27643 msg_qbytes
```

27644 **IPC_SET** can only be executed by a process with appropriate privileges or that
 27645 has an effective user ID equal to the value of **msg_perm.cuid** or
 27646 **msg_perm.uid** in the **msqid_ds** data structure associated with *msqid*. Only a
 27647 process with appropriate privileges can raise the value of **msg_qbytes**.

27648 **IPC_RMID** Remove the message queue identifier specified by *msqid* from the system and
 27649 destroy the message queue and **msqid_ds** data structure associated with it.
 27650 **IPC_RMID** can only be executed by a process with appropriate privileges or
 27651 one that has an effective user ID equal to the value of **msg_perm.cuid** or
 27652 **msg_perm.uid** in the **msqid_ds** data structure associated with *msqid*.

27653 **RETURN VALUE**

27654 Upon successful completion, *msgctl()* shall return 0; otherwise, it shall return -1 and set *errno* to
 27655 indicate the error.

27656 **ERRORS**

27657 The *msgctl()* function shall fail if:

27658 **[EACCES]** The argument *cmd* is **IPC_STAT** and the calling process does not have read
 27659 permission; see [Section 2.7](#) (on page 39).

27660 **[EINVAL]** The value of *msqid* is not a valid message queue identifier; or the value of *cmd*
 27661 is not a valid command.

27662 **[EPERM]** The argument *cmd* is **IPC_RMID** or **IPC_SET** and the effective user ID of the
 27663 calling process is not equal to that of a process with appropriate privileges and
 27664 it is not equal to the value of **msg_perm.cuid** or **msg_perm.uid** in the data
 27665 structure associated with *msqid*.

msgctl()

27666 [EPERM] The argument *cmd* is IPC_SET, an attempt is being made to increase to the
 27667 value of **msg_qbytes**, and the effective user ID of the calling process does not
 27668 have appropriate privileges.

EXAMPLES

27669 None.
 27670

APPLICATION USAGE

27671 The POSIX Realtime Extension defines alternative interfaces for interprocess communication
 27672 (IPC). Application developers who need to use IPC should design their applications so that
 27673 modules using the IPC routines described in [Section 2.7](#) can be easily modified to use the
 27674 alternative interfaces.
 27675

RATIONALE

27676 None.
 27677

FUTURE DIRECTIONS

27678 None.
 27679

SEE ALSO

27680 [Section 2.7](#) (on page 39), [Section 2.8](#) (on page 40), [mq_close\(\)](#), [mq_getattr\(\)](#), [mq_notify\(\)](#),
 27681 [mq_open\(\)](#), [mq_receive\(\)](#), [mq_send\(\)](#), [mq_setattr\(\)](#), [mq_unlink\(\)](#), [msgget\(\)](#), [msgrcv\(\)](#), [msgsnd\(\)](#), the
 27682 Base Definitions volume of IEEE Std 1003.1-200x, [<sys/msg.h>](#)
 27683

CHANGE HISTORY

27684 First released in Issue 2. Derived from Issue 2 of the SVID.
 27685

Issue 5

27686 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
 27687 DIRECTIONS to a new APPLICATION USAGE section.
 27688

27689 **NAME**

27690 msgget — get the XSI message queue identifier

27691 **SYNOPSIS**

```
27692 XSI #include <sys/msg.h>
27693 int msgget(key_t key, int msgflg);
```

27694 **DESCRIPTION**

27695 The *msgget()* function operates on XSI message queues (see the Base Definitions volume of
 27696 IEEE Std 1003.1-200x, Section 3.224, Message Queue). It is unspecified whether this function
 27697 interoperates with the realtime interprocess communication facilities defined in [Section 2.8](#) (on
 27698 page 40).

27699 The *msgget()* function shall return the message queue identifier associated with the argument
 27700 *key*.

27701 A message queue identifier, associated message queue, and data structure (see *<sys/msg.h>*),
 27702 shall be created for the argument *key* if one of the following is true:

- 27703 • The argument *key* is equal to `IPC_PRIVATE`.
- 27704 • The argument *key* does not already have a message queue identifier associated with it, and
 27705 (*msgflg* & `IPC_CREAT`) is non-zero.

27706 Upon creation, the data structure associated with the new message queue identifier shall be
 27707 initialized as follows:

- 27708 • `msg_perm.cuid`, `msg_perm.uid`, `msg_perm.cgid`, and `msg_perm.gid` shall be set equal to
 27709 the effective user ID and effective group ID, respectively, of the calling process.
- 27710 • The low-order 9 bits of `msg_perm.mode` shall be set equal to the low-order 9 bits of *msgflg*.
- 27711 • `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, and `msg_rtime` shall be set equal to 0.
- 27712 • `msg_ctime` shall be set equal to the current time.
- 27713 • `msg_qbytes` shall be set equal to the system limit.

27714 **RETURN VALUE**

27715 Upon successful completion, *msgget()* shall return a non-negative integer, namely a message
 27716 queue identifier. Otherwise, it shall return `-1` and set *errno* to indicate the error.

27717 **ERRORS**

27718 The *msgget()* function shall fail if:

- | | | |
|-------|----------|--|
| 27719 | [EACCES] | A message queue identifier exists for the argument <i>key</i> , but operation
27720 permission as specified by the low-order 9 bits of <i>msgflg</i> would not be granted;
27721 see Section 2.7 (on page 39). |
| 27722 | [EEXIST] | A message queue identifier exists for the argument <i>key</i> but $((msgflg \&$
27723 $IPC_CREAT) \&\& (msgflg \& IPC_EXCL))$ is non-zero. |
| 27724 | [ENOENT] | A message queue identifier does not exist for the argument <i>key</i> and $(msgflg \&$
27725 $IPC_CREAT)$ is 0. |
| 27726 | [ENOSPC] | A message queue identifier is to be created but the system-imposed limit on
27727 the maximum number of allowed message queue identifiers system-wide
27728 would be exceeded. |

27729

EXAMPLES

27730

None.

27731

APPLICATION USAGE

27732

27733

27734

27735

The POSIX Realtime Extension defines alternative interfaces for interprocess communication (IPC). Application developers who need to use IPC should design their applications so that modules using the IPC routines described in [Section 2.7](#) can be easily modified to use the alternative interfaces.

27736

RATIONALE

27737

None.

27738

FUTURE DIRECTIONS

27739

None.

27740

SEE ALSO

27741

27742

27743

[Section 2.7](#) (on page 39), [Section 2.8](#) (on page 40), [mq_close\(\)](#), [mq_getattr\(\)](#), [mq_notify\(\)](#), [mq_open\(\)](#), [mq_receive\(\)](#), [mq_send\(\)](#), [mq_setattr\(\)](#), [mq_unlink\(\)](#), [msgctl\(\)](#), [msgrcv\(\)](#), [msgsnd\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, [<sys/msg.h>](#)

27744

CHANGE HISTORY

27745

First released in Issue 2. Derived from Issue 2 of the SVID.

27746

Issue 5

27747

27748

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

27749 **NAME**

27750 msgrcv — XSI message receive operation

27751 **SYNOPSIS**

```
27752 XSI #include <sys/msg.h>
27753
27753 ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
27754               int msgflg);
```

27755 **DESCRIPTION**

27756 The *msgrcv()* function operates on XSI message queues (see the Base Definitions volume of
 27757 IEEE Std 1003.1-200x, Section 3.224, Message Queue). It is unspecified whether this function
 27758 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
 27759 page 40).

27760 The *msgrcv()* function shall read a message from the queue associated with the message queue
 27761 identifier specified by *msqid* and place it in the user-defined buffer pointed to by *msgp*.

27762 The application shall ensure that the argument *msgp* points to a user-defined buffer that contains
 27763 first a field of type **long** specifying the type of the message, and then a data portion that holds
 27764 the data bytes of the message. The structure below is an example of what this user-defined
 27765 buffer might look like:

```
27766 struct mymsg {
27767     long    mtype;    /* Message type. */
27768     char    mtext[1]; /* Message text. */
27769 }
```

27770 The structure member *mtype* is the received message's type as specified by the sending process.

27771 The structure member *mtext* is the text of the message.

27772 The argument *msgsz* specifies the size in bytes of *mtext*. The received message shall be truncated
 27773 to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & MSG_NOERROR) is non-zero. The
 27774 truncated part of the message shall be lost and no indication of the truncation shall be given to
 27775 the calling process.

27776 If the value of *msgsz* is greater than {SSIZE_MAX}, the result is implementation-defined.

27777 The argument *msgtyp* specifies the type of message requested as follows:

- 27778 • If *msgtyp* is 0, the first message on the queue shall be received.
- 27779 • If *msgtyp* is greater than 0, the first message of type *msgtyp* shall be received.
- 27780 • If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the
 27781 absolute value of *msgtyp* shall be received.

27782 The argument *msgflg* specifies the action to be taken if a message of the desired type is not on the
 27783 queue. These are as follows:

- 27784 • If (*msgflg* & IPC_NOWAIT) is non-zero, the calling thread shall return immediately with a
 27785 return value of -1 and *errno* set to [ENOMSG].
- 27786 • If (*msgflg* & IPC_NOWAIT) is 0, the calling thread shall suspend execution until one of the
 27787 following occurs:
 - 27788 — A message of the desired type is placed on the queue.

- 27789 — The message queue identifier *msqid* is removed from the system; when this occurs,
27790 *errno* shall be set equal to [EIDRM] and -1 shall be returned.
- 27791 — The calling thread receives a signal that is to be caught; in this case a message is not
27792 received and the calling thread resumes execution in the manner prescribed in
27793 *sigaction()*.

27794 Upon successful completion, the following actions are taken with respect to the data structure
27795 associated with *msqid*:

- 27796 • **msg_qnum** shall be decremented by 1.
- 27797 • **msg_lrpid** shall be set equal to the process ID of the calling process.
- 27798 • **msg_rtime** shall be set equal to the current time.

RETURN VALUE

27799 Upon successful completion, *msgrcv()* shall return a value equal to the number of bytes actually
27800 placed into the buffer *mtext*. Otherwise, no message shall be received, *msgrcv()* shall return
27801 **(ssize_t)-1**, and *errno* shall be set to indicate the error.

ERRORS

27803 The *msgrcv()* function shall fail if:

- | | | |
|-------|----------|--|
| 27805 | [E2BIG] | The value of <i>mtext</i> is greater than <i>msgsz</i> and (<i>msgflg</i> & MSG_NOERROR) is 0. |
| 27806 | [EACCES] | Operation permission is denied to the calling process; see Section 2.7 (on page
27807 39). |
| 27808 | [EIDRM] | The message queue identifier <i>msqid</i> is removed from the system. |
| 27809 | [EINTR] | The <i>msgrcv()</i> function was interrupted by a signal. |
| 27810 | [EINVAL] | <i>msqid</i> is not a valid message queue identifier. |
| 27811 | [ENOMSG] | The queue does not contain a message of the desired type and (<i>msgflg</i> &
27812 IPC_NOWAIT) is non-zero. |

EXAMPLES**Receiving a Message**

27814 The following example receives the first message on the queue (based on the value of the *msgtyp*
27815 argument, 0). The queue is identified by the *msqid* argument (assuming that the value has
27816 previously been set). This call specifies that an error should be reported if no message is
27817 available, but not if the message is too large. The message size is calculated directly using the
27818 *sizeof* operator.
27819

```

27820 #include <sys/msg.h>
27821 ...
27822 int result;
27823 int msqid;
27824 struct message {
27825     long type;
27826     char text[20];
27827 } msg;
27828 long msgtyp = 0;
27829 ...
27830 result = msgrcv(msqid, (void *) &msg, sizeof(msg.text),
27831               msgtyp, MSG_NOERROR | IPC_NOWAIT);

```

27832
27833
27834
27835
27836

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication (IPC). Application developers who need to use IPC should design their applications so that modules using the IPC routines described in [Section 2.7](#) can be easily modified to use the alternative interfaces.

27837
27838

RATIONALE

None.

27839
27840

FUTURE DIRECTIONS

None.

27841
27842
27843
27844

SEE ALSO

[Section 2.7](#) (on page 39), [Section 2.8](#) (on page 40), [mq_close\(\)](#), [mq_getattr\(\)](#), [mq_notify\(\)](#), [mq_open\(\)](#), [mq_receive\(\)](#), [mq_send\(\)](#), [mq_setattr\(\)](#), [mq_unlink\(\)](#), [msgctl\(\)](#), [msgget\(\)](#), [msgsnd\(\)](#), [sigaction\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, [<sys/msg.h>](#)

27845
27846

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

27847
27848
27849

Issue 5

The type of the return value is changed from **int** to **ssize_t**, and a warning is added to the DESCRIPTION about values of *msgsz* larger than `{SSIZE_MAX}`.

27850
27851

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to the APPLICATION USAGE section.

27852
27853

Issue 6

The normative text is updated to avoid use of the term “must” for application requirements.

27854 **NAME**
 27855 msgsnd — XSI message send operation

27856 **SYNOPSIS**

```
27857 XSI #include <sys/msg.h>
27858 int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

27859 **DESCRIPTION**

27860 The *msgsnd()* function operates on XSI message queues (see the Base Definitions volume of
 27861 IEEE Std 1003.1-200x, Section 3.224, Message Queue). It is unspecified whether this function
 27862 interoperates with the realtime interprocess communication facilities defined in [Section 2.8](#) (on
 27863 page 40).

27864 The *msgsnd()* function shall send a message to the queue associated with the message queue
 27865 identifier specified by *msqid*.

27866 The application shall ensure that the argument *msgp* points to a user-defined buffer that contains
 27867 first a field of type **long** specifying the type of the message, and then a data portion that holds
 27868 the data bytes of the message. The structure below is an example of what this user-defined
 27869 buffer might look like:

```
27870 struct mymsg {
27871     long   mtype;           /* Message type. */
27872     char   mtext[1];       /* Message text. */
27873 }
```

27874 The structure member *mtype* is a non-zero positive type **long** that can be used by the receiving
 27875 process for message selection.

27876 The structure member *mtext* is any text of length *msgsz* bytes. The argument *msgsz* can range
 27877 from 0 to a system-imposed maximum.

27878 The argument *msgflg* specifies the action to be taken if one or more of the following is true:

- 27879 • The number of bytes already on the queue is equal to **msg_qbytes**; see [<sys/msg.h>](#).
- 27880 • The total number of messages on all queues system-wide is equal to the system-imposed
 27881 limit.

27882 These actions are as follows:

- 27883 • If (*msgflg* & IPC_NOWAIT) is non-zero, the message shall not be sent and the calling
 27884 thread shall return immediately.
- 27885 • If (*msgflg* & IPC_NOWAIT) is 0, the calling thread shall suspend execution until one of the
 27886 following occurs:
 - 27887 — The condition responsible for the suspension no longer exists, in which case the
 27888 message is sent.
 - 27889 — The message queue identifier *msqid* is removed from the system; when this occurs,
 27890 *errno* shall be set equal to [EIDRM] and -1 shall be returned.
 - 27891 — The calling thread receives a signal that is to be caught; in this case the message is not
 27892 sent and the calling thread resumes execution in the manner prescribed in [sigaction\(\)](#).

27893 Upon successful completion, the following actions are taken with respect to the data structure
 27894 associated with *msqid*; see [<sys/msg.h>](#):

- 27895 • **msg_qnum** shall be incremented by 1.
- 27896 • **msg_lspid** shall be set equal to the process ID of the calling process.
- 27897 • **msg_stime** shall be set equal to the current time.

RETURN VALUE

27898 Upon successful completion, *msgsnd()* shall return 0; otherwise, no message shall be sent,
 27899 *msgsnd()* shall return -1, and *errno* shall be set to indicate the error.
 27900

ERRORS

27901 The *msgsnd()* function shall fail if:
 27902

- | | | |
|-------|----------|---|
| 27903 | [EACCES] | Operation permission is denied to the calling process; see Section 2.7 (on page 39). |
| 27905 | [EAGAIN] | The message cannot be sent for one of the reasons cited above and (<i>msgflg</i> & <i>IPC_NOWAIT</i>) is non-zero. |
| 27907 | [EIDRM] | The message queue identifier <i>msqid</i> is removed from the system. |
| 27908 | [EINTR] | The <i>msgsnd()</i> function was interrupted by a signal. |
| 27909 | [EINVAL] | The value of <i>msqid</i> is not a valid message queue identifier, or the value of <i>mtype</i> is less than 1; or the value of <i>msgsz</i> is less than 0 or greater than the system-imposed limit. |

EXAMPLES**Sending a Message**

27913 The following example sends a message to the queue identified by the *msqid* argument
 27914 (assuming that value has previously been set). This call specifies that an error should be
 27915 reported if no message is available. The message size is calculated directly using the *sizeof*
 27916 operator.
 27917

```

27918 #include <sys/msg.h>
27919 ...
27920 int result;
27921 int msqid;
27922 struct message {
27923     long type;
27924     char text[20];
27925 } msg;
27926 msg.type = 1;
27927 strcpy(msg.text, "This is message 1");
27928 ...
27929 result = msgsnd(msqid, (void *) &msg, sizeof(msg.text), IPC_NOWAIT);
  
```

APPLICATION USAGE

27930 The POSIX Realtime Extension defines alternative interfaces for interprocess communication
 27931 (IPC). Application developers who need to use IPC should design their applications so that
 27932 modules using the IPC routines described in [Section 2.7](#) can be easily modified to use the
 27933 alternative interfaces.
 27934

RATIONALE

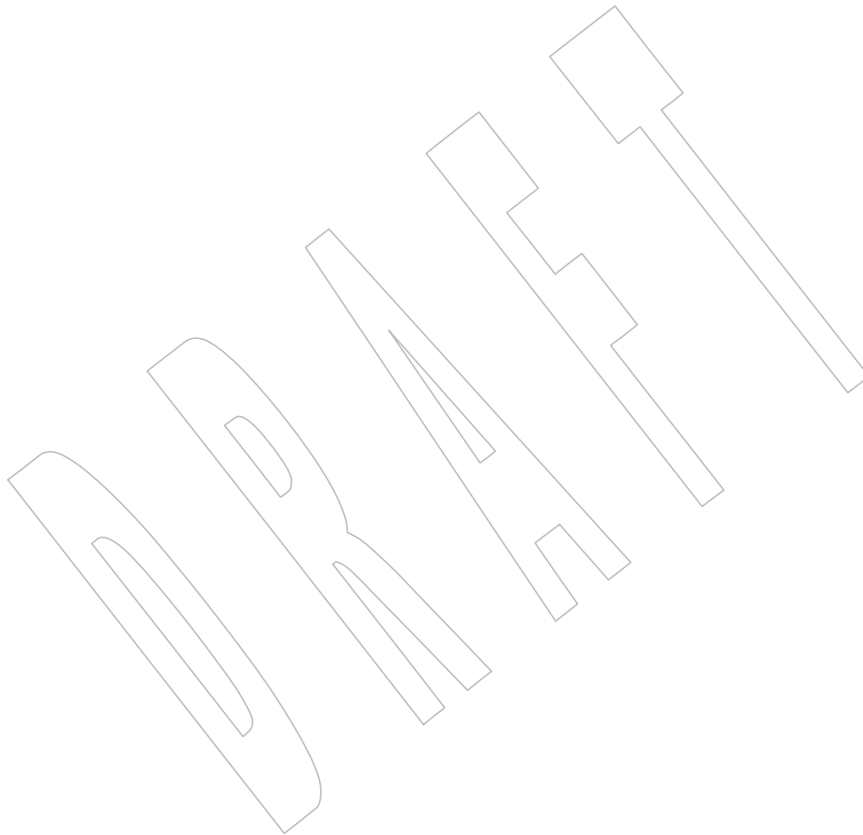
27935 None.
 27936

msgsnd()*System Interfaces***FUTURE DIRECTIONS**

None.

SEE ALSO

Section 2.7 (on page 39), Section 2.8 (on page 40), *mq_close()*, *mq_getattr()*, *mq_notify()*, *mq_open()*, *mq_receive()*



27950 **NAME**
 27951 `msync` — synchronize memory with physical storage

27952 **SYNOPSIS**

```
27953 SIO #include <sys/mman.h>
27954 int msync(void *addr, size_t len, int flags);
```

27955 **DESCRIPTION**

27956 The `msync()` function shall write all modified data to permanent storage locations, if any, in
 27957 those whole pages containing any part of the address space of the process starting at address
 27958 `addr` and continuing for `len` bytes. If no such storage exists, `msync()` need not have any effect. If
 27959 requested, the `msync()` function shall then invalidate cached copies of data.

27960 The implementation may require that `addr` be a multiple of the page size as returned by
 27961 `sysconf()`.

27962 For mappings to files, the `msync()` function shall ensure that all write operations are completed
 27963 as defined for synchronized I/O data integrity completion. It is unspecified whether the
 27964 implementation also writes out other file attributes. When the `msync()` function is called on
 27965 MAP_PRIVATE mappings, any modified data shall not be written to the underlying object and
 27966 shall not cause such data to be made visible to other processes. It is unspecified whether data in
 27967 SHM|TYM MAP_PRIVATE mappings has any permanent storage locations. The effect of `msync()` on a
 27968 shared memory object or a typed memory object is unspecified. The behavior of this function is
 27969 unspecified if the mapping was not established by a call to `mmap()`.

27970 The `flags` argument is constructed from the bitwise-inclusive OR of one or more of the following
 27971 flags defined in the `<sys/mman.h>` header:

Symbolic Constant	Description
MS_ASYNC	Perform asynchronous writes.
MS_SYNC	Perform synchronous writes.
MS_INVALIDATE	Invalidate cached data.

27976 When MS_ASYNC is specified, `msync()` shall return immediately once all the write operations
 27977 are initiated or queued for servicing; when MS_SYNC is specified, `msync()` shall not return until
 27978 all write operations are completed as defined for synchronized I/O data integrity completion.
 27979 Either MS_ASYNC or MS_SYNC shall be specified, but not both.

27980 When MS_INVALIDATE is specified, `msync()` shall invalidate all cached copies of mapped data
 27981 that are inconsistent with the permanent storage locations such that subsequent references shall
 27982 obtain data that was consistent with the permanent storage locations sometime between the call
 27983 to `msync()` and the first subsequent memory reference to the data.

27984 If `msync()` causes any write to a file, the file's `st_ctime` and `st_mtime` fields shall be marked for
 27985 update.

27986 **RETURN VALUE**

27987 Upon successful completion, `msync()` shall return 0; otherwise, it shall return -1 and set `errno` to
 27988 indicate the error.

27989 **ERRORS**

27990 The `msync()` function shall fail if:

27991 [EBUSY] Some or all of the addresses in the range starting at `addr` and continuing for `len`
 27992 bytes are locked, and MS_INVALIDATE is specified.

- 27993 [EINVAL] The value of *flags* is invalid.
- 27994 [ENOMEM] The addresses in the range starting at *addr* and continuing for *len* bytes are outside the range allowed for the address space of a process or specify one or more pages that are not mapped.
- 27995
- 27996
- 27997 The *msync()* function may fail if:
- 27998 [EINVAL] The value of *addr* is not a multiple of the page size as returned by *sysconf()*.

EXAMPLES

27999 None.

28000

APPLICATION USAGE

28001 The *msync()* function is only supported if the Synchronized Input and Output option is supported, and thus need not be available on all implementations.

28002

28003

28004 The *msync()* function should be used by programs that require a memory object to be in a known state; for example, in building transaction facilities.

28005

28006 Normal system activity can cause pages to be written to disk. Therefore, there are no guarantees that *msync()* is the only control over when pages are or are not written to disk.

28007

RATIONALE

28008 The *msync()* function writes out data in a mapped region to the permanent storage for the underlying object. The call to *msync()* ensures data integrity of the file.

28009

28010

28011 After the data is written out, any cached data may be invalidated if the MS_INVALIDATE flag was specified. This is useful on systems that do not support read/write consistency.

28012

FUTURE DIRECTIONS

28013 None.

28014

SEE ALSO

28015 *mmap()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-200x, <sys/mman.h>

28016

CHANGE HISTORY

28017 First released in Issue 4, Version 2.

28018

Issue 5

28019 Moved from X/OPEN UNIX extension to BASE.

28020

28021 Aligned with *msync()* in the POSIX Realtime Extension as follows:

- 28022 • The DESCRIPTION is extensively reworded.
- 28023 • [EBUSY] and a new form of [EINVAL] are added as mandatory error conditions.

Issue 6

28024 The *msync()* function is marked as part of the Memory Mapped Files and Synchronized Input and Output options.

28025

28026

28027 The following changes are made for alignment with the ISO POSIX-1:1996 standard:

- 28028 • The [EBUSY] mandatory error condition is added.

28029 The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

28030

- 28031 • The DESCRIPTION is updated to state that implementations require *addr* to be a multiple of the page size.
- 28032
- 28033 • The second [EINVAL] error condition is made mandatory.

28034 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding reference to typed memory objects.

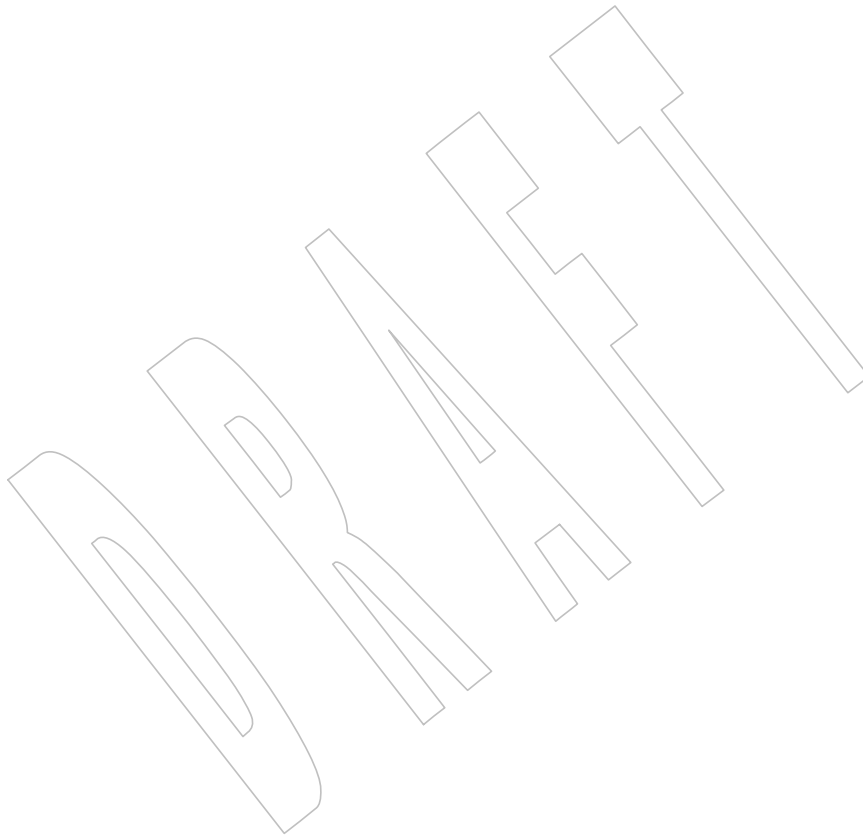
28035

28036
28037
28038
28039
28040**Issue 7**

SD5-XSH-ERN-110 is applied.

Austin Group Interpretation 1003.1-2001 #078 is applied, clarifying page alignment requirements.

The *msync()* function is moved from the Memory Mapped Files option.



munlock()

28041 **NAME**
28042 munlock — unlock a range of process address space

28043 **SYNOPSIS**

28044 MLR #include <sys/mman.h>
28045 int munlock(const void *addr, size_t len);

28046 **DESCRIPTION**

28047 Refer to *mlock()*.

28048 **NAME**
28049 `munlockall` — unlock the address space of a process

28050 **SYNOPSIS**

```
28051 ML #include <sys/mman.h>  
28052 int munlockall(void);
```

28053 **DESCRIPTION**

28054 Refer to *mlockall()*.

28055 **NAME**
 28056 `munmap` — unmap pages of memory

28057 **SYNOPSIS**
 28058 `#include <sys/mman.h>`
 28059 `int munmap(void *addr, size_t len);`

28060 **DESCRIPTION**

28061 The `munmap()` function shall remove any mappings for those entire pages containing any part of
 28062 the address space of the process starting at `addr` and continuing for `len` bytes. Further references
 28063 to these pages shall result in the generation of a SIGSEGV signal to the process. If there are no
 28064 mappings in the specified address range, then `munmap()` has no effect.

28065 The implementation may require that `addr` be a multiple of the page size as returned by
 28066 `sysconf()`.

28067 If a mapping to be removed was private, any modifications made in this address range shall be
 28068 discarded.

28069 MLIMLR Any memory locks (see `mlock()` and `mlockall()`) associated with this address range shall be
 28070 removed, as if by an appropriate call to `munlock()`.

28071 TYM If a mapping removed from a typed memory object causes the corresponding address range of
 28072 the memory pool to be inaccessible by any process in the system except through allocatable
 28073 mappings (that is, mappings of typed memory objects opened with the
 28074 POSIX_TYPED_MEM_MAP_ALLOCATABLE flag), then that range of the memory pool shall
 28075 become deallocated and may become available to satisfy future typed memory allocation
 28076 requests.

28077 A mapping removed from a typed memory object opened with the
 28078 POSIX_TYPED_MEM_MAP_ALLOCATABLE flag shall not affect in any way the availability of
 28079 that typed memory for allocation.

28080 The behavior of this function is unspecified if the mapping was not established by a call to
 28081 `mmap()`.

28082 **RETURN VALUE**

28083 Upon successful completion, `munmap()` shall return 0; otherwise, it shall return -1 and set `errno`
 28084 to indicate the error.

28085 **ERRORS**

28086 The `munmap()` function shall fail if:

28087 [EINVAL] Addresses in the range [`addr`,`addr+len`) are outside the valid range for the
 28088 address space of a process.

28089 [EINVAL] The `len` argument is 0.

28090 The `munmap()` function may fail if:

28091 [EINVAL] The `addr` argument is not a multiple of the page size as returned by `sysconf()`.

28092
28093
28094
28095
28096
28097
28098
28099
28100
28101
28102
28103
28104
28105
28106
28107
28108
28109
28110
28111
28112
28113
28114
28115
28116
28117
28118
28119
28120
28121
28122
28123
28124
28125
28126
28127
28128
28129
28130

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *munmap()* function corresponds to SVR4, just as the *mmap()* function does.

It is possible that an application has applied process memory locking to a region that contains shared memory. If this has occurred, the *munmap()* call ignores those locks and, if necessary, causes those locks to be removed.

Most implementations require that *addr* is a multiple of the page size as returned by *sysconf()*.

FUTURE DIRECTIONS

None.

SEE ALSO

mlock(), *mlockall()*, *mmap()*, *posix_typed_mem_open()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Aligned with *munmap()* in the POSIX Realtime Extension as follows:

- The DESCRIPTION is extensively reworded.
- The SIGBUS error is no longer permitted to be generated.

Issue 6

The *munmap()* function is marked as part of the Memory Mapped Files and Shared Memory Objects option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to state that implementations require *addr* to be a multiple of the page size.
- The [EINVAL] error conditions are added.

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- Semantics for typed memory objects are added to the DESCRIPTION.
- The *posix_typed_mem_open()* function is added to the SEE ALSO section.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/36 is applied, changing the margin code in the SYNOPSIS from MF|SHM to MC3 (notation for MF|SHM|TYM).

Issue 7

Austin Group Interpretation 1003.1-2001 #078 is applied, clarifying page alignment requirements.

The *munmap()* function is moved from the Memory Mapped Files option to the Base.

28131 **NAME**

28132 nan, nanf, nanl — return quiet NaN

28133 **SYNOPSIS**

28134 #include <math.h>

28135 double nan(const char *tagp);

28136 float nanf(const char *tagp);

28137 long double nanl(const char *tagp);

28138 **DESCRIPTION**

28139 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 28140 conflict between the requirements described here and the ISO C standard is unintentional. This
 28141 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

28142 The function call *nan("n-char-sequence")* shall be equivalent to:28143 strtod("NAN(*n-char-sequence*)", (char **) NULL);28144 The function call *nan("")* shall be equivalent to:

28145 strtod("NAN()", (char **) NULL)

28146 If *tagp* does not point to an *n-char* sequence or an empty string, the function call shall be
 28147 equivalent to:

28148 strtod("NAN", (char **) NULL)

28149 Function calls to *nanf()* and *nanl()* are equivalent to the corresponding function calls to *strtof()*
 28150 and *strtold()*.28151 **RETURN VALUE**28152 These functions shall return a quiet NaN, if available, with content indicated through *tagp*.

28153 If the implementation does not support quiet NaNs, these functions shall return zero.

28154 **ERRORS**

28155 No errors are defined.

28156 **EXAMPLES**

28157 None.

28158 **APPLICATION USAGE**

28159 None.

28160 **RATIONALE**

28161 None.

28162 **FUTURE DIRECTIONS**

28163 None.

28164 **SEE ALSO**28165 *strtod()*, *strtold()*, the Base Definitions volume of IEEE Std 1003.1-200x, <math.h>28166 **CHANGE HISTORY**

28167 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

28168 **NAME**

28169 nanosleep — high resolution sleep

28170 **SYNOPSIS**

```
28171 CX #include <time.h>
28172 int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```

28173 **DESCRIPTION**

28174 The *nanosleep()* function shall cause the current thread to be suspended from execution until
 28175 either the time interval specified by the *rqtp* argument has elapsed or a signal is delivered to the
 28176 calling thread, and its action is to invoke a signal-catching function or to terminate the process.
 28177 The suspension time may be longer than requested because the argument value is rounded up to
 28178 an integer multiple of the sleep resolution or because of the scheduling of other activity by the
 28179 system. But, except for the case of being interrupted by a signal, the suspension time shall not be
 28180 less than the time specified by *rqtp*, as measured by the system clock `CLOCK_REALTIME`.

28181 The use of the *nanosleep()* function has no effect on the action or blockage of any signal.

28182 **RETURN VALUE**

28183 If the *nanosleep()* function returns because the requested time has elapsed, its return value shall
 28184 be zero.

28185 If the *nanosleep()* function returns because it has been interrupted by a signal, it shall return a
 28186 value of `-1` and set *errno* to indicate the interruption. If the *rmtp* argument is non-NULL, the
 28187 **timespec** structure referenced by it is updated to contain the amount of time remaining in the
 28188 interval (the requested time minus the time actually slept). The *rqtp* and *rmtp* arguments may
 28189 point to the same object. If the *rmtp* argument is NULL, the remaining time is not returned.

28190 If *nanosleep()* fails, it shall return a value of `-1` and set *errno* to indicate the error.

28191 **ERRORS**

28192 The *nanosleep()* function shall fail if:

- | | | |
|-------|----------|--|
| 28193 | [EINTR] | The <i>nanosleep()</i> function was interrupted by a signal. |
| 28194 | [EINVAL] | The <i>rqtp</i> argument specified a nanosecond value less than zero or greater than
28195 or equal to 1 000 million. |

28196 **EXAMPLES**

28197 None.

28198 **APPLICATION USAGE**

28199 None.

28200 **RATIONALE**

28201 It is common to suspend execution of a thread for an interval in order to poll the status of a non-
 28202 interrupting function. A large number of actual needs can be met with a simple extension to
 28203 *sleep()* that provides finer resolution.

28204 In the POSIX.1-1990 standard and SVR4, it is possible to implement such a routine, but the
 28205 frequency of wakeup is limited by the resolution of the *alarm()* and *sleep()* functions. In 4.3 BSD,
 28206 it is possible to write such a routine using no static storage and reserving no system facilities.
 28207 Although it is possible to write a function with similar functionality to *sleep()* using the
 28208 remainder of the *timer_**() functions, such a function requires the use of signals and the
 28209 reservation of some signal number. This volume of IEEE Std 1003.1-200x requires that
 28210 *nanosleep()* be non-intrusive of the signals function.

nanosleep()

28211 The *nanosleep()* function shall return a value of 0 on success and -1 on failure or if interrupted.
28212 This latter case is different from *sleep()*. This was done because the remaining time is returned
28213 via an argument structure pointer, *rmtpt*, instead of as the return value.

FUTURE DIRECTIONS

28214 None.
28215

SEE ALSO

28216 *clock_nanosleep()*, *sleep()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**time.h**>
28217

CHANGE HISTORY

28218 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.
28219

Issue 6

28220 The *nanosleep()* function is marked as part of the Timers option.
28221

28222 The [ENOSYS] error condition has been removed as stubs need not be provided if an
28223 implementation does not support the Timers option.

28224 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/37 is applied, updating the SEE ALSO
28225 section to include the *clock_nanosleep()* function.

28226 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/63 is applied, correcting text in the
28227 RATIONALE section.

Issue 7

28228 SD5-XBD-ERN-33 is applied.
28229

28230 The *nanosleep()* function is moved from the Timers option to the Base.

28231 **NAME**
 28232 nearbyint, nearbyintf, nearbyintl — floating-point rounding functions

28233 **SYNOPSIS**
 28234 #include <math.h>
 28235 double nearbyint(double x);
 28236 float nearbyintf(float x);
 28237 long double nearbyintl(long double x);

28238 **DESCRIPTION**
 28239 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 28240 conflict between the requirements described here and the ISO C standard is unintentional. This
 28241 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

28242 These functions shall round their argument to an integer value in floating-point format, using
 28243 the current rounding direction and without raising the inexact floating-point exception.

28244 An application wishing to check for error situations should set *errno* to zero and call
 28245 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 28246 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 28247 zero, an error has occurred.

28248 **RETURN VALUE**
 28249 Upon successful completion, these functions shall return the rounded integer value.

28250 MX If *x* is NaN, a NaN shall be returned.

28251 If *x* is ± 0 , ± 0 shall be returned.

28252 If *x* is $\pm\text{Inf}$, *x* shall be returned.

28253 XSI If the correct value would cause overflow, a range error shall occur and *nearbyint*(), *nearbyintf*(),
 28254 and *nearbyintl*() shall return the value of the macro $\pm\text{HUGE_VAL}$, $\pm\text{HUGE_VALF}$, and
 28255 $\pm\text{HUGE_VALL}$ (with the same sign as *x*), respectively.

28256 **ERRORS**
 28257 These functions shall fail if:

28258 XSI **Range Error** The result would cause an overflow.

28259 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 28260 then *errno* shall be set to [ERANGE]. If the integer expression
 28261 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
 28262 floating-point exception shall be raised.

28263 **EXAMPLES**
 28264 None.

28265 **APPLICATION USAGE**
 28266 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
 28267 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

28268 **RATIONALE**
 28269 None.

nearbyint()

28270

FUTURE DIRECTIONS

28271

None.

28272

SEE ALSO

28273

feclearexcept(), *fetestexcept()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18,

28274

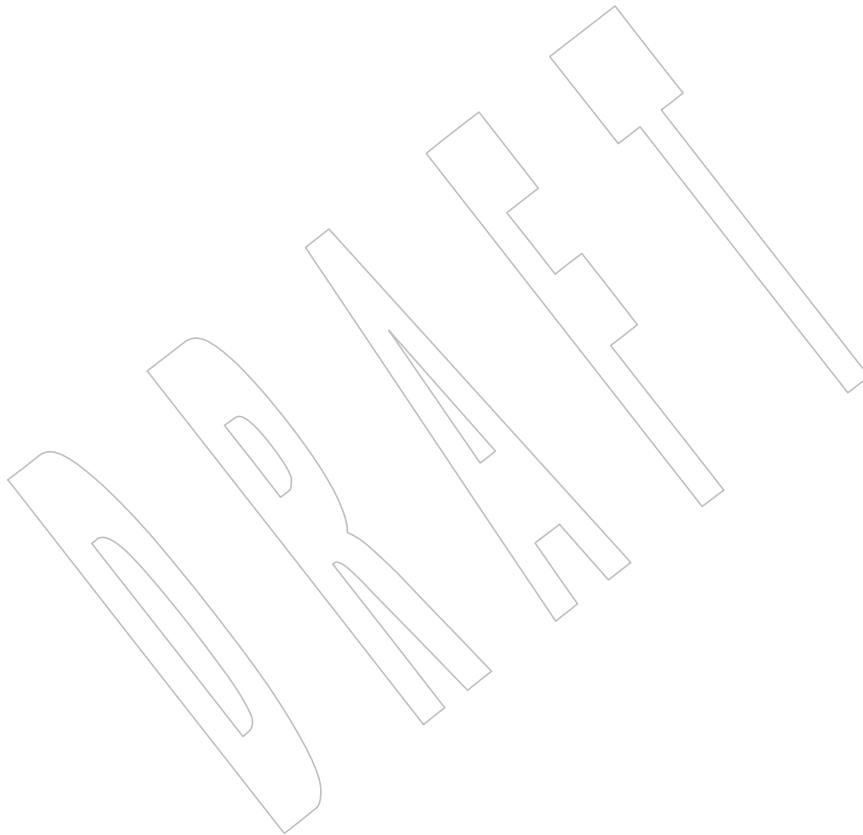
Treatment of Error Conditions for Mathematical Functions, **<math.h>**

28275

CHANGE HISTORY

28276

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.



28277 **NAME**

28278 newlocale — create or modify a locale object

28279 **SYNOPSIS**

```
28280 CX #include <locale.h>
28281 locale_t newlocale(int category_mask, const char *locale,
28282 locale_t base);
```

28283 **DESCRIPTION**

28284 The *newlocale()* function shall create a new locale object or modify an existing one. If the *base*
 28285 argument is (**locale_t**)0, a new locale object shall be created. It is unspecified whether the locale
 28286 object pointed to by *base* shall be modified or freed and a new locale object created.

28287 The *category_mask* argument specifies the locale categories to be set or modified. Values for
 28288 *category_mask* shall be constructed by a bitwise-inclusive OR of the symbolic constants
 28289 *LC_CTYPE_MASK*, *LC_NUMERIC_MASK*, *LC_TIME_MASK*, *LC_COLLATE_MASK*,
 28290 *LC_MONETARY_MASK*, and *LC_MESSAGES_MASK*, or any of the other implementation-
 28291 defined *LC_*_MASK* values defined in **<locale.h>**.

28292 For each category with the corresponding bit set in *category_mask* the data from the locale named
 28293 by *locale* shall be used. In the case of modifying an existing locale object, the data from the locale
 28294 named by *locale* shall replace the existing data within the locale object. If a completely new locale
 28295 object is created, the data for all sections not requested by *category_mask* shall be taken from the
 28296 default locale.

28297 The following preset values of *locale* are defined for all settings of *category_mask*:

28298	"POSIX"	Specifies the minimal environment for C-language translation called the 28299 POSIX locale.
28300	"C"	Equivalent to "POSIX".
28301	" "	Specifies an implementation-defined native environment. This corresponds to 28302 the value of the associated environment variables, <i>LC_*</i> and <i>LANG</i> ; see the 28303 Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale and the 28304 Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment 28305 Variables.

28306 If the *base* argument is not (**locale_t**)0 and the *newlocale()* function call succeeds, the contents of
 28307 *base* are unspecified. Applications shall ensure that they stop using *base* as a locale object before
 28308 calling *newlocale()*. If the function call fails and the *base* argument is not (**locale_t**)0, the contents
 28309 of *base* shall remain valid and unchanged.

28310 The results are undefined if the *base* argument is the special locale object *LC_GLOBAL_LOCALE*.

28311 **RETURN VALUE**

28312 Upon successful completion, the *newlocale()* function shall return a handle which the caller may
 28313 use on subsequent calls to *duplocale()*, *freelocale()*, and other functions taking a **locale_t**
 28314 argument.

28315 Upon failure, the *newlocale()* function shall return (**locale_t**)0 and set *errno* to indicate the error.

28316 **ERRORS**

28317 The *newlocale()* function shall fail if:

- 28318 [ENOMEM] There is not enough memory available to create the locale object or load the
28319 locale data.
- 28320 [EINVAL] The *category_mask* contains a bit that does not correspond to a valid category.
- 28321 [ENOENT] For any of the categories in *category_mask*, the locale data is not available.
- 28322 The *newlocale()* function may fail if:
- 28323 [EINVAL] The *locale* argument is not a valid string pointer.

EXAMPLES**Constructing a Locale Object from Different Locales**

The following example shows the construction of a locale where the *LC_CTYPE* category data comes from a locale *loc1* and the *LC_TIME* category data from a locale *tok2*:

```
28328 #include <locale.h>
28329 ...
28330 locale_t loc, new_loc;
28331 /* Get the "loc1" data. */
28332 loc = newlocale (LC_CTYPE_MASK, "loc1", NULL);
28333 if (loc == (locale_t) 0)
28334     abort ();
28335 /* Get the "loc2" data. */
28336 new_loc = newlocale (LC_TIME_MASK, "loc2", loc);
28337 if (new_loc != (locale_t) 0)
28338     /* We don't abort if this fails. In this case this
28339     simply used to unchanged locale object. */
28340     loc = new_loc;
28341 ...
```

Freeing up a Locale Object

The following example shows a code fragment to free a locale object created by *newlocale()*:

```
28344 #include <locale.h>
28345 ...
28346 /* Every locale object allocated with newlocale() should be
28347 * freed using freelocale():
28348 */
28349 locale_t loc;
28350 /* Get the locale. */
28351 loc = newlocale (LC_CTYPE_MASK | LC_TIME_MASK, "locname", NULL);
28352 /* ... Use the locale object ... */
28353 ...
28354 /* Free the locale object resources. */
28355 freelocale (loc);
```

28356 **APPLICATION USAGE**

28357 Handles for locale objects created by the *newlocale()* function should be released by a
28358 corresponding call to *freelocale()*.

28359 The special locale object *LC_GLOBAL_LOCALE* must not be passed for the *base* argument, even
28360 when returned by the *uselocale()* function.

28361 **RATIONALE**

28362 None.

28363 **FUTURE DIRECTIONS**

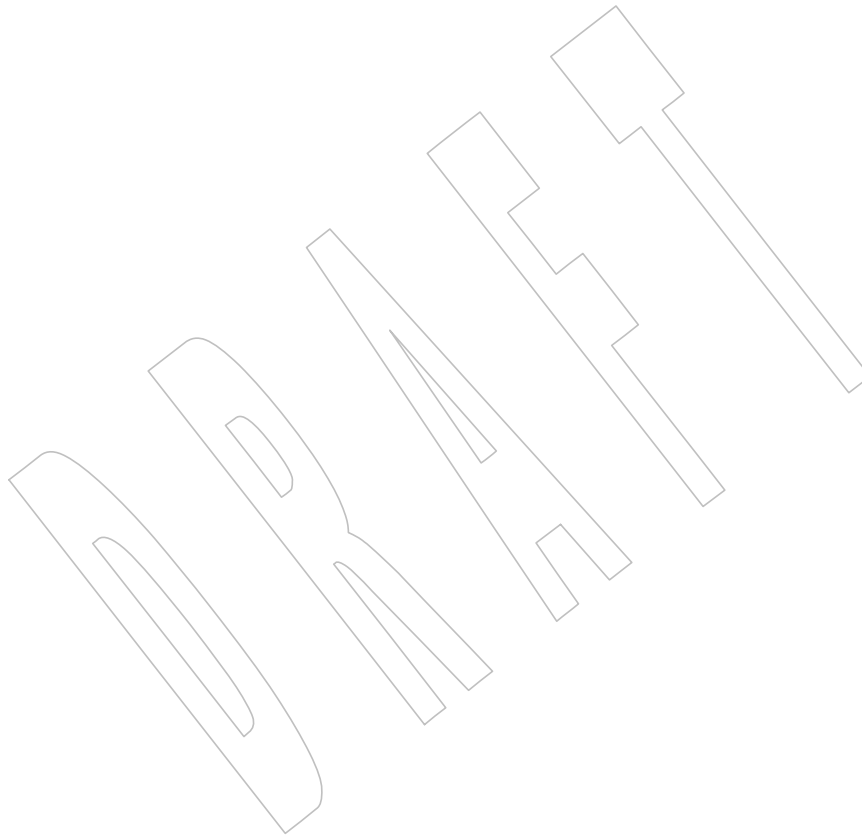
28364 None.

28365 **SEE ALSO**

28366 *duplocale()*, *freelocale()*, *uselocale()*, the Base Definitions volume of IEEE Std 1003.1-200x,
28367 **<locale.h>**

28368 **CHANGE HISTORY**

28369 First released in Issue 7.



28370 **NAME**

28371 nextafter, nextafterf, nextafterl, nexttoward, nexttowardf, nexttowardl — next representable
 28372 floating-point number

28373 **SYNOPSIS**

28374 #include <math.h>

28375 double nextafter(double x, double y);

28376 float nextafterf(float x, float y);

28377 long double nextafterl(long double x, long double y);

28378 double nexttoward(double x, long double y);

28379 float nexttowardf(float x, long double y);

28380 long double nexttowardl(long double x, long double y);

28381 **DESCRIPTION**

28382 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 28383 conflict between the requirements described here and the ISO C standard is unintentional. This
 28384 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

28385 The *nextafter()*, *nextafterf()*, and *nextafterl()* functions shall compute the next representable
 28386 floating-point value following *x* in the direction of *y*. Thus, if *y* is less than *x*, *nextafter()* shall
 28387 return the largest representable floating-point number less than *x*. The *nextafter()*, *nextafterf()*,
 28388 and *nextafterl()* functions shall return *y* if *x* equals *y*.

28389 The *nexttoward()*, *nexttowardf()*, and *nexttowardl()* functions shall be equivalent to the
 28390 corresponding *nextafter()* functions, except that the second parameter shall have type **long**
 28391 **double** and the functions shall return *y* converted to the type of the function if *x* equals *y*.

28392 An application wishing to check for error situations should set *errno* to zero and call
 28393 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 28394 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 28395 zero, an error has occurred.

28396 **RETURN VALUE**

28397 Upon successful completion, these functions shall return the next representable floating-point
 28398 value following *x* in the direction of *y*.

28399 If $x=y$, *y* (of the type *x*) shall be returned.

28400 If *x* is finite and the correct function value would overflow, a range error shall occur and
 28401 \pm HUGE_VAL, \pm HUGE_VALF, and \pm HUGE_VALL (with the same sign as *x*) shall be returned as
 28402 appropriate for the return type of the function.

28403 **MX** If *x* or *y* is NaN, a NaN shall be returned.

28404 If $x \neq y$ and the correct function value is subnormal, zero, or underflows, a range error shall
 28405 occur, and either the correct function value (if representable) or 0.0 shall be returned.

28406 **ERRORS**

28407 These functions shall fail if:

28408 **Range Error** The correct value overflows.

28409 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 28410 then *errno* shall be set to [ERANGE]. If the integer expression
 28411 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
 28412 floating-point exception shall be raised.

28413 MX **Range Error** The correct value is subnormal or underflows.
 28414 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 28415 then *errno* shall be set to [ERANGE]. If the integer expression
 28416 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 28417 floating-point exception shall be raised.

EXAMPLES

28418 None.
 28419

APPLICATION USAGE

28420 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
 28421 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.
 28422

RATIONALE

28423 None.
 28424

FUTURE DIRECTIONS

28425 None.
 28426

SEE ALSO

28427 *feclearexcept()*, *fetestexcept()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18,
 28428 Treatment of Error Conditions for Mathematical Functions, <**math.h**>
 28429

CHANGE HISTORY

28430 First released in Issue 4, Version 2.
 28431

Issue 5

28432 Moved from X/OPEN UNIX extension to BASE.
 28433

Issue 6

28434 The *nextafter()* function is no longer marked as an extension.
 28435

28436 The *nextafterf()*, *nextafterl()*, *nexttoward()*, *nexttowardf()*, and *nexttowardl()* functions are added
 28437 for alignment with the ISO/IEC 9899:1999 standard.

28438 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
 28439 revised to align with the ISO/IEC 9899:1999 standard.

28440 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
 28441 marked.

28442 **NAME**

28443 nftw — walk a file tree

28444 **SYNOPSIS**

```
28445 XSI #include <ftw.h>
28446 int nftw(const char *path, int (*fn)(const char *,
28447      const struct stat *, int, struct FTW *), int fd_limit, int flags);
```

28448 **DESCRIPTION**

28449 The *nftw()* function shall recursively descend the directory hierarchy rooted in *path*. The *nftw()*
 28450 function has a similar effect to *ftw()* except that it takes an additional argument *flags*, which is a
 28451 bitwise-inclusive OR of zero or more of the following flags:

28452 **FTW_CHDIR** If set, *nftw()* shall change the current working directory to each directory as it
 28453 reports files in that directory. If clear, *nftw()* shall not change the current
 28454 working directory.

28455 **FTW_DEPTH** If set, *nftw()* shall report all files in a directory before reporting the directory
 28456 itself. If clear, *nftw()* shall report any directory before reporting the files in that
 28457 directory.

28458 **FTW_MOUNT** If set, *nftw()* shall only report files in the same file system as *path*. If clear,
 28459 *nftw()* shall report all files encountered during the walk.

28460 **FTW_PHYS** If set, *nftw()* shall perform a physical walk and shall not follow symbolic links.

28461 If **FTW_PHYS** is clear and **FTW_DEPTH** is set, *nftw()* shall follow links instead of reporting
 28462 them, but shall not report any directory that would be a descendant of itself. If **FTW_PHYS** is
 28463 clear and **FTW_DEPTH** is clear, *nftw()* shall follow links instead of reporting them, but shall not
 28464 report the contents of any directory that would be a descendant of itself.

28465 At each file it encounters, *nftw()* shall call the user-supplied function *fn* with four arguments:

- 28466 • The first argument is the pathname of the object.
- 28467 • The second argument is a pointer to the **stat** buffer containing information on the object,
 28468 filled in as if *fstatat()*, *stat()*, or *lstat()* had been called to retrieve the information.
- 28469 • The third argument is an integer giving additional information. Its value is one of the
 28470 following:

28471 **FTW_F** The object is a file.

28472 **FTW_D** The object is a directory.

28473 **FTW_DP** The object is a directory and subdirectories have been visited. (This condition
 28474 shall only occur if the **FTW_DEPTH** flag is included in *flags*.)

28475 **FTW_SL** The object is a symbolic link. (This condition shall only occur if the
 28476 **FTW_PHYS** flag is included in *flags*.)

28477 **FTW_SLN** The object is a symbolic link that does not name an existing file. (This
 28478 condition shall only occur if the **FTW_PHYS** flag is not included in *flags*.)

28479 **FTW_DNR** The object is a directory that cannot be read. The *fn* function shall not be
 28480 called for any of its descendants.

28481 FTW_NS The *stat()* function failed on the object because of lack of appropriate
 28482 permission. The **stat** buffer passed to *fn* is undefined. Failure of *stat()* for any
 28483 other reason is considered an error and *nftw()* shall return -1 .

28484 • The fourth argument is a pointer to an **FTW** structure. The value of **base** is the offset of the
 28485 object's filename in the pathname passed as the first argument to *fn*. The value of **level**
 28486 indicates depth relative to the root of the walk, where the root level is 0.

28487 The results are unspecified if the application-supplied *fn* function does not preserve the current
 28488 working directory.

28489 The argument *fd_limit* sets the maximum number of file descriptors that shall be used by *nftw()*
 28490 while traversing the file tree. At most one file descriptor shall be used for each directory level.

28491 The *nftw()* function need not be thread-safe. A function that is not required to be thread-safe is
 28492 not required to be reentrant.

28493 RETURN VALUE

28494 The *nftw()* function shall continue until the first of the following conditions occurs:

- 28495 • An invocation of *fn* shall return a non-zero value, in which case *nftw()* shall return that
 28496 value.
- 28497 • The *nftw()* function detects an error other than [EACCES] (see FTW_DNR and FTW_NS
 28498 above), in which case *nftw()* shall return -1 and set *errno* to indicate the error.
- 28499 • The tree is exhausted, in which case *nftw()* shall return 0.

28500 ERRORS

28501 The *nftw()* function shall fail if:

- 28502 [EACCES] Search permission is denied for any component of *path* or read permission is
 28503 denied for *path*, or *fn* returns -1 and does not reset *errno*.
- 28504 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 28505 argument.
- 28506 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname
 28507 component is longer than {NAME_MAX}.
 28508
- 28509 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.
- 28510 [ENOTDIR] A component of *path* is not a directory.
- 28511 [EOVERFLOW] A field in the **stat** structure cannot be represented correctly in the current
 28512 programming environment for one or more files found in the file hierarchy.

28513 The *nftw()* function may fail if:

- 28514 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 28515 resolution of the *path* argument.
- 28516 [EMFILE] All file descriptors available to the process are currently open.
- 28517 [ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result
 28518 whose length exceeds {PATH_MAX}.
 28519
- 28520 [ENFILE] Too many files are currently open in the system.

28521 In addition, *errno* may be set if the function pointed to by *fn* causes *errno* to be set.

28522 **EXAMPLES**

28523 The following example walks the **/tmp** directory and its subdirectories, calling the *nftw()*
 28524 function for every directory entry, using a maximum of 5 file descriptors.

```
28525 #include <ftw.h>
28526 ...
28527 int nftwfunc(const char *, const struct stat *, int, struct FTW *);
28528
28528 int nftwfunc(const char *filename, const struct stat *statptr,
28529             int fileflags, struct FTW *pftw)
28530 {
28531     return 0;
28532 }
28533 ...
28534 char *startpath = "/tmp";
28535 int fd_limit = 5;
28536 int flags = FTW_CHDIR | FTW_DEPTH | FTW_MOUNT;
28537 int ret;
28538
28538 ret = nftw(startpath, nftwfunc, fd_limit, flags);
```

28539 **APPLICATION USAGE**

28540 None.

28541 **RATIONALE**

28542 None.

28543 **FUTURE DIRECTIONS**

28544 None.

28545 **SEE ALSO**

28546 *fdopendir()*, *fstatat()*, *readdir()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<ftw.h>**

28547 **CHANGE HISTORY**

28548 First released in Issue 4, Version 2.

28549 **Issue 5**

28550 Moved from X/OPEN UNIX extension to BASE.

28551 In the DESCRIPTION, the definition of the *depth* argument is clarified.

28552 **Issue 6**

28553 The Open Group Base Resolution bwg97-003 is applied.

28554 The ERRORS section is updated as follows:

- 28555 • The wording of the mandatory [ELOOP] error condition is updated.
- 28556 • A second optional [ELOOP] error condition is added.
- 28557 • The [Eoverflow] mandatory error condition is added.

28558 Text is added to the DESCRIPTION to say that the *nftw()* function need not be reentrant and that
 28559 the results are unspecified if the application-supplied *fn* function does not preserve the current
 28560 working directory.

28561 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/64 is applied, changing the argument
 28562 *depth* to *fd_limit* throughout and changing “to a maximum of 5 levels deep” to “using a
 28563 maximum of 5 file descriptors” in the EXAMPLES section.

28564

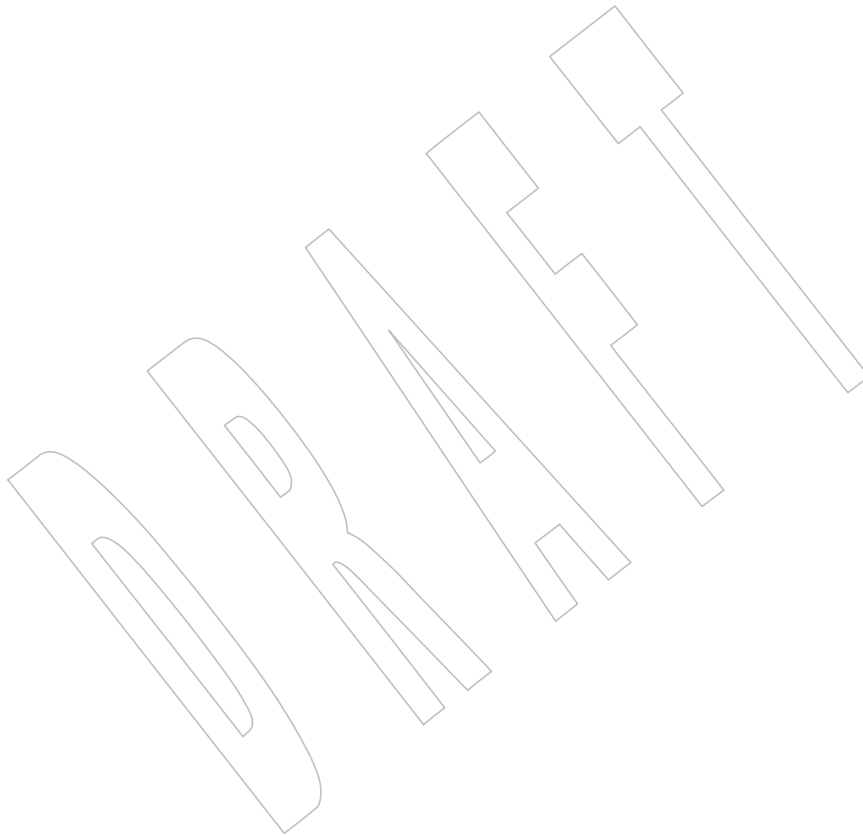
Issue 7

28565

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

28566

SD6-XBD-ERN-61 is applied.



28567 **NAME**28568 `nice` — change the nice value of a process28569 **SYNOPSIS**

```
28570 XSI #include <unistd.h>
28571 int nice(int incr);
```

28572 **DESCRIPTION**

28573 The `nice()` function shall add the value of `incr` to the nice value of the calling process. A nice
 28574 value of a process is a non-negative number for which a more positive value shall result in less
 28575 favorable scheduling.

28576 A maximum nice value of $2*\{NZERO\}-1$ and a minimum nice value of 0 shall be imposed by the
 28577 system. Requests for values above or below these limits shall result in the nice value being set to
 28578 the corresponding limit. Only a process with appropriate privileges can lower the nice value.

28579 PS|TPS Calling the `nice()` function has no effect on the priority of processes or threads with policy
 28580 SCHED_FIFO or SCHED_RR. The effect on processes or threads with other scheduling policies
 28581 is implementation-defined.

28582 The nice value set with `nice()` shall be applied to the process. If the process is multi-threaded, the
 28583 nice value shall affect all system scope threads in the process.

28584 As -1 is a permissible return value in a successful situation, an application wishing to check for
 28585 error situations should set `errno` to 0, then call `nice()`, and if it returns -1 , check to see whether
 28586 `errno` is non-zero.

28587 **RETURN VALUE**

28588 Upon successful completion, `nice()` shall return the new nice value $\{-NZERO\}$. Otherwise, -1
 28589 shall be returned, the nice value of the process shall not be changed, and `errno` shall be set to
 28590 indicate the error.

28591 **ERRORS**

28592 The `nice()` function shall fail if:

28593 [EPERM] The `incr` argument is negative and the calling process does not have
 28594 appropriate privileges.

28595 **EXAMPLES**28596 **Changing the Nice Value**

28597 The following example adds the value of the `incr` argument, -20 , to the nice value of the calling
 28598 process.

```
28599 #include <unistd.h>
28600 ...
28601 int incr = -20;
28602 int ret;
28603 ret = nice(incr);
```

28604 **APPLICATION USAGE**

28605 None.

28606

RATIONALE

28607

None.

28608

FUTURE DIRECTIONS

28609

None.

28610

SEE ALSO

28611

exec, *getpriority()*, *setpriority()*, the Base Definitions volume of IEEE Std 1003.1-200x, <limits.h>,

28612

<unistd.h>

28613

CHANGE HISTORY

28614

First released in Issue 1. Derived from Issue 1 of the SVID.

28615

Issue 5

28616

A statement is added to the description indicating the effects of this function on the different scheduling policies and multi-threaded processes.

28617



28618 **NAME**28619 `nl_langinfo, nl_langinfo_l` — language information28620 **SYNOPSIS**28621 `#include <langinfo.h>`28622 `char *nl_langinfo(nl_item item);`28623 `char *nl_langinfo_l(nl_item item, locale_t locale);`28624 **DESCRIPTION**

28625 The `nl_langinfo()` and `nl_langinfo_l()` functions shall return a pointer to a string containing
 28626 information relevant to the particular language or cultural area defined in the locale of the
 28627 process, or in the locale represented by `locale`, respectively (see **<langinfo.h>**). The manifest
 28628 constant names and values of `item` are defined in **<langinfo.h>**. For example:

28629 `nl_langinfo(ABDAY_1)`

28630 would return a pointer to the string "Dom" if the identified language was Portuguese, and
 28631 "Sun" if the identified language was English.

28632 `nl_langinfo_l(ABDAY_1, loc)`

28633 would return a pointer to the string "Dom" if the identified language of the locale represented by
 28634 `loc` was Portuguese, and "Sun" if the identified language of the locale represented by `loc` was
 28635 English.

28636 Calls to `setlocale()` with a category corresponding to the category of `item` (see **<langinfo.h>**), or to
 28637 the category `LC_ALL`, may overwrite the array pointed to by the return value. Calls to `uselocale()`
 28638 which change the category corresponding to the category of `item` may overwrite the array
 28639 pointed to by the return value.

28640 The `nl_langinfo()` function need not be thread-safe. A function that is not required to be thread-
 28641 safe is not required to be reentrant.

28642 **RETURN VALUE**

28643 In a locale where `langinfo` data is not defined, these functions shall return a pointer to the
 28644 corresponding string in the POSIX locale. In all locales, these functions shall return a pointer to
 28645 an empty string if `item` contains an invalid setting.

28646 This pointer may point to static data that may be overwritten on the next call to either function.

28647 **ERRORS**28648 The `nl_langinfo_l()` function may fail if:28649 [EINVAL] `locale` is not a valid locale object handle.28650 **EXAMPLES**28651 **Getting Date and Time Formatting Information**

28652 The following example returns a pointer to a string containing date and time formatting
 28653 information, as defined in the `LC_TIME` category of the current locale.

28654 `#include <time.h>`28655 `#include <langinfo.h>`28656 `...`28657 `strftime(datestring, sizeof(datestring), nl_langinfo(D_T_FMT), tm);`28658 `...`

28659

APPLICATION USAGE

28660

The array pointed to by the return value should not be modified by the program, but may be modified by further calls to these functions.

28661

28662

RATIONALE

28663

None.

28664

FUTURE DIRECTIONS

28665

None.

28666

SEE ALSO

28667

setlocale(), *uselocale()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale, `<langinfo.h>`, `<locale.h>`, `<nl_types.h>`

28668

28669

CHANGE HISTORY

28670

First released in Issue 2.

28671

Issue 5

28672

The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section.

28673

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

28674

Issue 7

28675

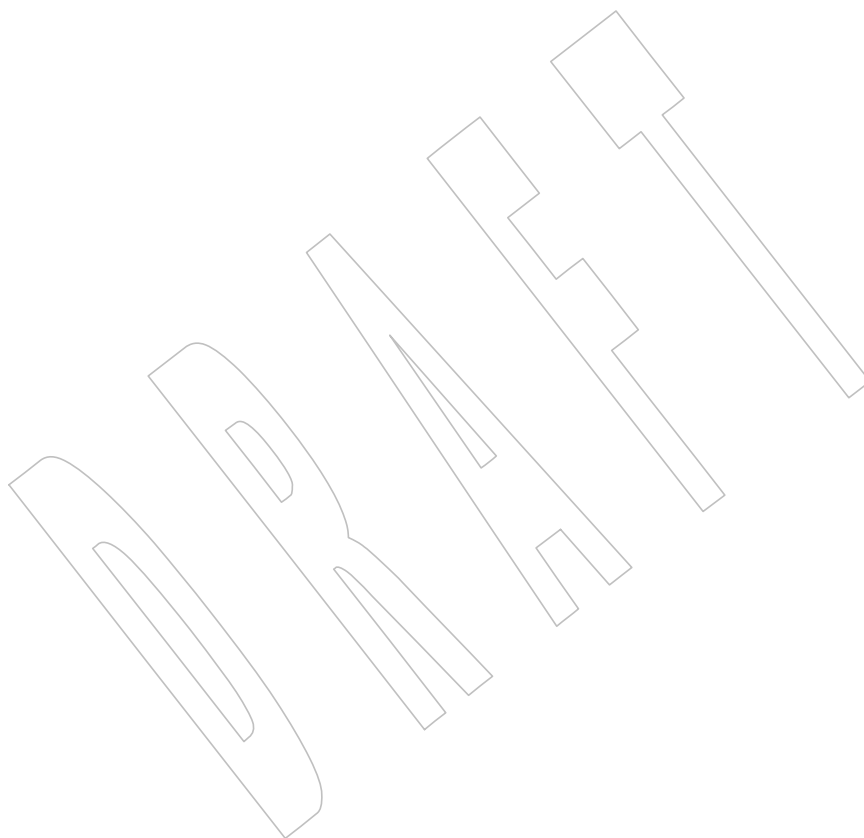
The *nl_langinfo()* function is moved from the XSI option to the Base.

28676

The *nl_langinfo_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

28677

DRAFT



28685 **NAME**
28686 ntohl, ntohs — convert values between host and network byte order

28687 **SYNOPSIS**
28688 #include <arpa/inet.h>
28689 uint32_t ntohl(uint32_t netlong);
28690 uint16_t ntohs(uint16_t netshort);

28691 **DESCRIPTION**
28692 Refer to [htonl\(\)](#).



28693 **NAME**
 28694 open, openat — open file relative to directory file descriptor

28695 **SYNOPSIS**

```
28696 OH #include <sys/stat.h>
28697 #include <fcntl.h>

28698 int open(const char *path, int oflag, ... );
28699 int openat(int fd, const char *path, int oflag, ...);
```

28700 **DESCRIPTION**

28701 The *open()* function shall establish the connection between a file and a file descriptor. It shall
 28702 create an open file description that refers to a file and a file descriptor that refers to that open file
 28703 description. The file descriptor is used by other I/O functions to refer to that file. The *path*
 28704 argument points to a pathname naming the file.

28705 The *open()* function shall return a file descriptor for the named file that is the lowest file
 28706 descriptor not currently open for that process. The open file description is new, and therefore the
 28707 file descriptor shall not share it with any other process in the system. The FD_CLOEXEC file
 28708 descriptor flag associated with the new file descriptor shall be cleared.

28709 The file offset used to mark the current position within the file shall be set to the beginning of
 28710 the file.

28711 The file status flags and file access modes of the open file description shall be set according to
 28712 the value of *oflag*.

28713 Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list, defined
 28714 in **<fcntl.h>**. Applications shall specify exactly one of the first four values (file access modes)
 28715 below in the value of *oflag*:

28716 O_EXEC Open for execute only (non-directory files). Use of this flag on directories
 28717 is currently unspecified.

28718 O_RDONLY Open for reading only.

28719 O_WRONLY Open for writing only.

28720 O_RDWR Open for reading and writing. The result is undefined if this flag is
 28721 applied to a FIFO.

28722 Any combination of the following may be used:

28723 O_APPEND If set, the file offset shall be set to the end of the file prior to each write.

28724 O_CREAT If the file exists, this flag has no effect except as noted under O_EXCL
 28725 below. Otherwise, the file shall be created; the user ID of the file shall be
 28726 set to the effective user ID of the process; the group ID of the file shall be
 28727 set to the group ID of the file's parent directory or to the effective group
 28728 ID of the process; and the access permission bits (see **<sys/stat.h>**) of the
 28729 file mode shall be set to the value of the argument following the *oflag*
 28730 argument taken as type **mode_t** modified as follows: a bitwise AND is
 28731 performed on the file-mode bits and the corresponding bits in the
 28732 complement of the process' file mode creation mask. Thus, all bits in the
 28733 file mode whose corresponding bit in the file mode creation mask is set
 28734 are cleared. When bits other than the file permission bits are set, the effect
 28735 is unspecified. The argument following the *oflag* argument does not affect
 28736 whether the file is open for reading, writing, or for both. Implementations
 28737 shall provide a way to initialize the file's group ID to the group ID of the

28738			parent directory. Implementations may, but need not, provide an implementation-defined way to initialize the file's group ID to the effective group ID of the calling process.
28739			
28740			
28741	SIO	O_DIRECTORY	If <i>path</i> does not name a directory, fail and set <i>errno</i> to [ENOTDIR]. Write I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion.
28742			
28743			
28744		O_EXCL	If O_CREAT and O_EXCL are set, <i>open()</i> shall fail if the file exists. The check for the existence of the file and the creation of the file if it does not exist shall be atomic with respect to other threads executing <i>open()</i> naming the same filename in the same directory with O_EXCL and O_CREAT set. If O_EXCL and O_CREAT are set, and <i>path</i> names a symbolic link, <i>open()</i> shall fail and set <i>errno</i> to [EEXIST], regardless of the contents of the symbolic link. If O_EXCL is set and O_CREAT is not set, the result is undefined.
28745			
28746			
28747			
28748			
28749			
28750			
28751			
28752		O_NOCTTY	If set and <i>path</i> identifies a terminal device, <i>open()</i> shall not cause the terminal device to become the controlling terminal for the process.
28753			
28754		O_NOFOLLOW	If <i>path</i> names a symbolic link, fail and set <i>errno</i> to [ELOOP]. When opening a FIFO with O_RDONLY or O_WRONLY set:
28755			
28756			• If O_NONBLOCK is set, an <i>open()</i> for reading-only shall return without delay. An <i>open()</i> for writing-only shall return an error if no process currently has the file open for reading.
28757			
28758			
28759			• If O_NONBLOCK is clear, an <i>open()</i> for reading-only shall block the calling thread until a thread opens the file for writing. An <i>open()</i> for writing-only shall block the calling thread until a thread opens the file for reading.
28760			
28761			
28762			
28763			When opening a block special or character special file that supports non-blocking opens:
28764			
28765			• If O_NONBLOCK is set, the <i>open()</i> function shall return without blocking for the device to be ready or available. Subsequent behavior of the device is device-specific.
28766			
28767			
28768			• If O_NONBLOCK is clear, the <i>open()</i> function shall block the calling thread until the device is ready or available before returning.
28769			
28770			Otherwise, the behavior of O_NONBLOCK is unspecified.
28771	SIO	O_RSYNC	Read I/O operations on the file descriptor shall complete at the same level of integrity as specified by the O_DSYNC and O_SYNC flags. If both O_DSYNC and O_RSYNC are set in <i>oflag</i> , all I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion. If both O_SYNC and O_RSYNC are set in flags, all I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.
28772			
28773			
28774			
28775			
28776			
28777			
28778	XSI SIO	O_SYNC	Write I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.
28779			
28780	XSI		The O_SYNC flag shall be supported for regular files, even if the Synchronized Input and Output option is not supported.
28781			
28782		O_TRUNC	If the file exists and is a regular file, and the file is successfully opened O_RDWR or O_WRONLY, its length shall be truncated to 0, and the mode and owner shall be unchanged. It shall have no effect on FIFO special files
28783			
28784			

open()

28785 or terminal device files. Its effect on other file types is implementation-
 28786 defined. The result of using O_TRUNC without either O_RDWR or
 28787 O_WRONLY is undefined.

28788 If O_CREAT is set and the file did not previously exist, upon successful completion, *open()* shall
 28789 mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file and the *st_ctime* and
 28790 *st_mtime* fields of the parent directory.

28791 If O_TRUNC is set and the file did previously exist, upon successful completion, *open()* shall
 28792 mark for update the *st_ctime* and *st_mtime* fields of the file.

28793 SIO If both the O_SYNC and O_DSYNC flags are set, the effect is as if only the O_SYNC flag was set.

28794 OB XSR If *path* refers to a STREAMS file, *oflag* may be constructed from O_NONBLOCK OR'ed with
 28795 either O_RDONLY, O_WRONLY, or O_RDWR. Other flag values are not applicable to STREAMS
 28796 devices and shall have no effect on them. The value O_NONBLOCK affects the operation of
 28797 STREAMS drivers and certain functions applied to file descriptors associated with STREAMS
 28798 files. For STREAMS drivers, the implementation of O_NONBLOCK is device-specific.

28799 XSI If *path* names the master side of a pseudo-terminal device, then it is unspecified whether *open()*
 28800 locks the slave side so that it cannot be opened. Conforming applications shall call *unlockpt()*
 28801 before opening the slave side.

28802 The largest value that can be represented correctly in an object of type **off_t** shall be established
 28803 as the offset maximum in the open file description.

28804 The *openat()* function shall be equivalent to the *open()* function except in the case where *path*
 28805 specifies a relative path. In this case the file to be opened is determined relative to the directory
 28806 associated with the file descriptor *fd* instead of the current working directory. It is unspecified
 28807 whether directory searches are permitted based on whether the file was opened with search
 28808 permission or on the current permissions of the directory underlying the file descriptor.

28809 The *oflag* parameter and the optional fourth parameter correspond exactly to the parameters of
 28810 *open()*.

28811 If *openat()* is passed the special value AT_FDCWD in the *fd* parameter, the current working
 28812 directory is used and the behavior shall be identical to a call to *open()*.

RETURN VALUE

28813 Upon successful completion, these functions shall open the file and return a non-negative
 28814 integer representing the lowest numbered unused file descriptor. Otherwise, these functions
 28815 shall return -1 and set *errno* to indicate the error. If - is returned, no files shall be created or
 28816 modified.
 28817

ERRORS

28818 These functions shall fail if:

28819

28820 [EACCES] Search permission is denied on a component of the path prefix, or the file
 28821 exists and the permissions specified by *oflag* are denied, or the file does not
 28822 exist and write permission is denied for the parent directory of the file to be
 28823 created, or O_TRUNC is specified and write permission is denied.

28824 [EEXIST] O_CREAT and O_EXCL are set, and the named file exists.

28825 [EINTR] A signal was caught during *open()*.

28826 SIO [EINVAL] The implementation does not support synchronized I/O for this file.

28827 OB XSR [EIO] The *path* argument names a STREAMS file and a hangup or error occurred
 28828 during the *open()*.

28829		[EISDIR]	The named file is a directory and <i>oflag</i> includes O_WRONLY or O_RDWR.
28830		[ELOOP]	A loop exists in symbolic links encountered during resolution of the <i>path</i> argument, or O_NOFOLLOW was specified and the <i>path</i> argument names a symbolic link.
28831			
28832			
28833		[EMFILE]	All file descriptors available to the process are currently open.
28834		[ENAMETOOLONG]	
28835			The length of the <i>path</i> argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
28836			
28837		[ENFILE]	The maximum allowable number of files is currently open in the system.
28838		[ENOENT]	O_CREAT is not set and the named file does not exist; or O_CREAT is set and either the path prefix does not exist or the <i>path</i> argument points to an empty string.
28839			
28840			
28841	OB XSR	[ENOSR]	The <i>path</i> argument names a STREAMS-based file and the system is unable to allocate a STREAM.
28842			
28843		[ENOSPC]	The directory or file system that would contain the new file cannot be expanded, the file does not exist, and O_CREAT is specified.
28844			
28845		[ENOTDIR]	A component of the path prefix is not a directory, or O_DIRECTORY was specified and the <i>path</i> argument does not name a directory.
28846			
28847		[ENXIO]	O_NONBLOCK is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.
28848			
28849		[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
28850			
28851		[EOVERFLOW]	The named file is a regular file and the size of the file cannot be represented correctly in an object of type off_t .
28852			
28853		[EROFS]	The named file resides on a read-only file system and either O_WRONLY, O_RDWR, O_CREAT (if the file does not exist), or O_TRUNC is set in the <i>oflag</i> argument.
28854			
28855			
28856			The <i>openat()</i> function shall fail if:
28857		[EBADF]	The <i>path</i> argument does not specify an absolute path and the <i>fd</i> argument is neither AT_FDCWD nor a valid file descriptor open for searching.
28858			
28859			These functions may fail if:
28860	XSI	[EAGAIN]	The <i>path</i> argument names the slave side of a pseudo-terminal device that is locked.
28861			
28862		[EINVAL]	The value of the <i>oflag</i> argument is not valid.
28863		[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the <i>path</i> argument.
28864			
28865		[ENAMETOOLONG]	
28866			As a result of encountering a symbolic link in resolution of the <i>path</i> argument, the length of the substituted pathname string exceeded {PATH_MAX}.
28867			
28868	OB XSR	[ENOMEM]	The <i>path</i> argument names a STREAMS file and the system is unable to allocate resources.
28869			
28870		[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and <i>oflag</i> is O_WRONLY or O_RDWR.
28871			

28872 The `openat()` function may fail if:

28873 [ENOTDIR] The `path` argument is not an absolute path and `fd` is neither `AT_FDCWD` nor a
28874 file descriptor associated with a directory.

28875 EXAMPLES

28876 Opening a File for Writing by the Owner

28877 The following example opens the file `/tmp/file`, either by creating it (if it does not already exist),
28878 or by truncating its length to 0 (if it does exist). In the former case, if the call creates a new file,
28879 the access permission bits in the file mode of the file are set to permit reading and writing by the
28880 owner, and to permit reading only by group members and others.

28881 If the call to `open()` is successful, the file is opened for writing.

```
28882 #include <fcntl.h>
28883 ...
28884 int fd;
28885 mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
28886 char *filename = "/tmp/file";
28887 ...
28888 fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, mode);
28889 ...
```

28890 Opening a File Using an Existence Check

28891 The following example uses the `open()` function to try to create the `LOCKFILE` file and open it
28892 for writing. Since the `open()` function specifies the `O_EXCL` flag, the call fails if the file already
28893 exists. In that case, the program assumes that someone else is updating the password file and
28894 exits.

```
28895 #include <fcntl.h>
28896 #include <stdio.h>
28897 #include <stdlib.h>
28898 #define LOCKFILE "/etc/ptmp"
28899 ...
28900 int pfd; /* Integer for file descriptor returned by open() call. */
28901 ...
28902 if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
28903 S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
28904 {
28905     fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
28906     exit(1);
28907 }
28908 ...
```

28909 Opening a File for Writing

28910 The following example opens a file for writing, creating the file if it does not already exist. If the
28911 file does exist, the system truncates the file to zero bytes.

```
28912 #include <fcntl.h>
28913 #include <stdio.h>
28914 #include <stdlib.h>
28915 #define LOCKFILE "/etc/ptmp"
28916 ...
```

```

28917     int pfd;
28918     char filename[PATH_MAX+1];
28919     ...
28920     if ((pfd = open(filename, O_WRONLY | O_CREAT | O_TRUNC,
28921         S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
28922     {
28923         perror("Cannot open output file\n"); exit(1);
28924     }
28925     ...

```

APPLICATION USAGE

None.

RATIONALE

Except as specified in this volume of IEEE Std 1003.1-200x, the flags allowed in *oflag* are not mutually-exclusive and any number of them may be used simultaneously.

Some implementations permit opening FIFOs with O_RDWR. Since FIFOs could be implemented in other ways, and since two file descriptors can be used to the same effect, this possibility is left as undefined.

See *getgroups()* about the group of a newly created file.

The use of *open()* to create a regular file is preferable to the use of *creat()*, because the latter is redundant and included only for historical reasons.

The use of the O_TRUNC flag on FIFOs and directories (pipes cannot be *open()*-ed) must be permissible without unexpected side effects (for example, *creat()* on a FIFO must not remove data). Since terminal special files might have type-ahead data stored in the buffer, O_TRUNC should not affect their content, particularly if a program that normally opens a regular file should open the current controlling terminal instead. Other file types, particularly implementation-defined ones, are left implementation-defined.

IEEE Std 1003.1-200x permits [EACCES] to be returned for conditions other than those explicitly listed.

The O_NOCTTY flag was added to allow applications to avoid unintentionally acquiring a controlling terminal as a side effect of opening a terminal file. This volume of IEEE Std 1003.1-200x does not specify how a controlling terminal is acquired, but it allows an implementation to provide this on *open()* if the O_NOCTTY flag is not set and other conditions specified in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface are met. The O_NOCTTY flag is an effective no-op if the file being opened is not a terminal device.

In historical implementations the value of O_RDONLY is zero. Because of that, it is not possible to detect the presence of O_RDONLY and another option. Future implementations should encode O_RDONLY and O_WRONLY as bit flags so that:

```
O_RDONLY | O_WRONLY == O_RDWR
```

O_EXEC is specified as one of the four file access modes. On implementations where none of O_RDONLY, O_WRONLY, or O_RDWR is zero, applications may open a directory with O_EXEC OR'd in with one of the other three file access modes. On many historical implementations, this cannot be done since O_RDONLY has been defined to be zero.

In general, the *open()* function follows the symbolic link if *path* names a symbolic link. However, the *open()* function, when called with O_CREAT and O_EXCL, is required to fail with [EEXIST] if *path* names an existing symbolic link, even if the symbolic link refers to a nonexistent file. This behavior is required so that privileged applications can create a new file in a known location without the possibility that a symbolic link might cause the file to be created in a different

28965 location.

28966 In addition, the *open()* function refuses to open non-directories if the `O_DIRECTORY` flag is set.
 28967 This avoids race conditions whereby a user might compromise the system by substituting a hard
 28968 link to a sensitive file (e.g., a device or a FIFO) while a privileged application is running, where
 28969 opening a file even for read access might have undesirable side-effects.

28970 In addition, the *open()* function does not follow symbolic links if the `O_NOFOLLOW` flag is set.
 28971 This avoids race conditions whereby a user might compromise the system by substituting a
 28972 symbolic link to a sensitive file (e.g., a device) while a privileged application is running, where
 28973 opening a file even for read access might have undesirable side-effects.

28974 For example, a privileged application that must create a file with a predictable name in a user-
 28975 writable directory, such as the user's home directory, could be compromised if the user creates a
 28976 symbolic link with that name that refers to a nonexistent file in a system directory. If the user can
 28977 influence the contents of a file, the user could compromise the system by creating a new system
 28978 configuration or spool file that would then be interpreted by the system. The test for a symbolic
 28979 link which refers to a nonexistent file must be atomic with the creation of a new file.

28980 The POSIX.1-1990 standard required that the group ID of a newly created file be set to the group
 28981 ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required
 28982 that implementations provide a way to have the group ID be set to the group ID of the
 28983 containing directory, but did not prohibit implementations also supporting a way to set the
 28984 group ID to the effective group ID of the creating process. Conforming applications should not
 28985 assume which group ID will be used. If it matters, an application can use *chown()* to set the
 28986 group ID after the file is created, or determine under what conditions the implementation will
 28987 set the desired group ID.

28988 The purpose of the *openat()* function is to enable opening files in directories other than the
 28989 current working directory without exposure to race conditions. Any part of the path of a file
 28990 could be changed in parallel to a call to *open()*, resulting in unspecified behavior. By opening a
 28991 file descriptor for the target directory and using the *openat()* function it can be guaranteed that
 28992 the opened file is located relative to the desired directory. Some implementations use the
 28993 *openat()* function for other purposes as well. In some cases, if the *oflag* parameter has the
 28994 `O_XATTR` bit set, the returned file descriptor provides access to extended attributes. This
 28995 functionality is not standardized here.

28996 FUTURE DIRECTIONS

28997 The meaning of the `O_EXEC` flag on directories may be specified in a future version.

28998 SEE ALSO

28999 *chmod()*, *close()*, *creat()*, *dirfd()*, *dup()*, *exec*, *fcntl()*, *fdopendir()*, *link()*, *lseek()*, *mkdtemp()*,
 29000 *mknod()*, *read()*, *symlink()*, *umask()*, *unlockpt()*, *write()*, the Base Definitions volume of
 29001 IEEE Std 1003.1-200x, `<fcntl.h>`, `<sys/stat.h>`, `<sys/types.h>`

29002 CHANGE HISTORY

29003 First released in Issue 1. Derived from Issue 1 of the SVID.

29004 Issue 5

29005 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
 29006 Threads Extension.

29007 Large File Summit extensions are added.

29008 Issue 6

29009 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

29010 The following new requirements on POSIX implementations derive from alignment with the
 29011 Single UNIX Specification:

- 29012
- 29013
- 29014
- The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- 29015
- 29016
- 29017
- In the DESCRIPTION, `O_CREAT` is amended to state that the group ID of the file is set to the group ID of the file's parent directory or to the effective group ID of the process. This is a FIPS requirement.
- 29018
- 29019
- In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file description. This change is to support large files.
- 29020
- 29021
- In the ERRORS section, the `[EOVERFLOW]` condition is added. This change is to support large files.
- 29022
- The `[ENXIO]` mandatory error condition is added.
- 29023
- The `[EINVAL]`, `[ENAMETOOLONG]`, and `[ETXTBSY]` optional error conditions are added.
- 29024
- 29025
- The DESCRIPTION and ERRORS sections are updated so that items related to the optional XSI STREAMS Option Group are marked.
- 29026
- The following changes were made to align with the IEEE P1003.1a draft standard:
- 29027
- An explanation is added of the effect of the `O_CREAT` and `O_EXCL` flags when the path refers to a symbolic link.
- 29028
- 29029
- The `[ELOOP]` optional error condition is added.
- 29030
- The normative text is updated to avoid use of the term "must" for application requirements.
- 29031
- The DESCRIPTION of `O_EXCL` is updated in response to IEEE PASC Interpretation 1003.1c #48.
- 29032
- Issue 7**
- 29033
- SD5-XBD-ERN-4 is applied, changing the definition of the `[EMFILE]` error.
- 29034
- 29035
- This page is revised and the `openat()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.
- 29036
- Functionality relating to the XSI STREAMS option is marked obsolescent.
- 29037
- Austin Group Interpretation 1003.1-2001 #113 is applied.

29038 **NAME**29039 `open_memstream, open_wmemstream` — open a dynamic memory buffer stream29040 **SYNOPSIS**

```

29041 CX #include <stdio.h>
29042 FILE *open_memstream(char **bufp, size_t *sizep);
29043 #include <wchar.h>
29044 FILE *open_wmemstream(wchar_t **bufp, size_t *sizep);

```

29045 **DESCRIPTION**

29046 The `open_memstream()` and `open_wmemstream()` functions shall create an I/O stream associated
 29047 with a dynamically allocated memory buffer. The stream shall be opened for writing and shall
 29048 be seekable.

29049 The stream associated with a call to `open_memstream()` shall be byte-oriented.

29050 The stream associated with a call to `open_wmemstream()` shall be wide-oriented.

29051 The stream shall maintain a current position in the allocated buffer and a current buffer length.
 29052 The position shall be initially set to zero (the start of the buffer). Each write to the stream shall
 29053 start at the current position and move this position by the number of successfully written bytes
 29054 for `open_memstream()` or the number of successfully written wide characters for
 29055 `open_wmemstream()`. The length shall be initially set to zero. If a write moves the position to a
 29056 value larger than the current length, the current length shall be set to this position. In this case a
 29057 null character for `open_memstream()` or a null wide character for `open_wmemstream()` shall be
 29058 appended to the current buffer. For both functions the terminating null is not included in the
 29059 calculation of the buffer length.

29060 After a successful `fflush()` or `fclose()`, the pointer referenced by `bufp` shall contain the address of
 29061 the buffer, and the variable pointed to by `sizep` shall contain the number of successfully written
 29062 bytes for `open_memstream()` or the number of successfully written wide characters for
 29063 `open_wmemstream()`. The buffer shall be terminated by a null character for `open_memstream()` or
 29064 a null wide character for `open_wmemstream()`.

29065 After a successful `fflush()` the pointer referenced by `bufp` and the variable referenced by `sizep`
 29066 remain valid only until the next write operation on the stream or a call to `fclose()`.

29067 **RETURN VALUE**

29068 Upon successful completion, these functions shall return a pointer to the object controlling the
 29069 stream. Otherwise, a null pointer shall be returned, and `errno` shall be set to indicate the error.

29070 **ERRORS**

29071 These functions may fail if:

- 29072 [EINVAL] `bufp` or `sizep` are NULL.
- 29073 [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.
- 29074 [ENOMEM] Memory for the stream or the buffer could not be allocated.

29075 **EXAMPLES**

```

29076 #include <stdio.h>
29077 int main (void)
29078 {
29079     FILE *stream;
29080     char *buf;
29081     size_t len;
29082
29083     stream = open_memstream(&buf, &len);
29084
29085     if (stream == NULL)
29086         /* handle error */;
29087
29088     fprintf(stream, "hello my world");
29089     fflush(stream);
29090     printf("buf=%s, len=%zu\n", buf, len);
29091     fseeko(stream, 0, SEEK_SET);
29092     fprintf(stream, "good-bye");
29093     fclose(stream);
29094     printf("buf=%s, len=%zu\n", buf, len);
29095     free(buf);
29096     return 0;
29097 }

```

29095 This program produces the following output:

```

29096 buf=hello my world, len=14
29097 buf=good-bye world, len=14

```

29098 **APPLICATION USAGE**

29099 The buffer created by these functions should be freed by the application after closing the stream,
 29100 by means of a call to *free()*.

29101 **RATIONALE**

29102 These functions are similar to *fmemopen()* except that the memory is always allocated
 29103 dynamically by the function, and the stream is opened only for output.

29104 **FUTURE DIRECTIONS**

29105 None.

29106 **SEE ALSO**

29107 *fclose()*, *fdopen()*, *fflush()*, *fmemopen()*, *fopen()*, *free()*, *freopen()*, the Base Definitions volume of
 29108 IEEE Std 1003.1-200x, **<stdio.h>**

29109 **CHANGE HISTORY**

29110 First released in Issue 7.

29111 **NAME**
29112 open_wmemstream — open a dynamic memory buffer stream

29113 **SYNOPSIS**

```
29114 CX       #include <wchar.h>  
29115       FILE *open_wmemstream(wchar_t **bufp, size_t *sizep);
```

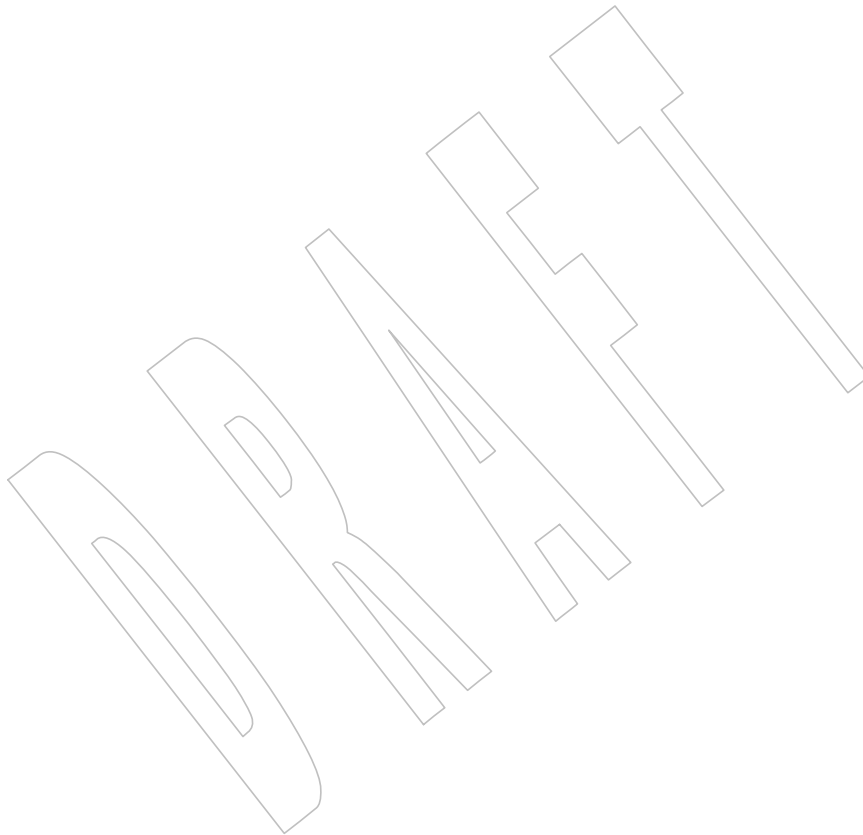
29116 **DESCRIPTION**

29117 Refer to *open_memstream()*.

29118 **NAME**
29119 openat — open file relative to directory file descriptor

29120 **SYNOPSIS**
29121 #include <fcntl.h>
29122 int openat(int *fd*, const char **path*, int *oflag*, ...);

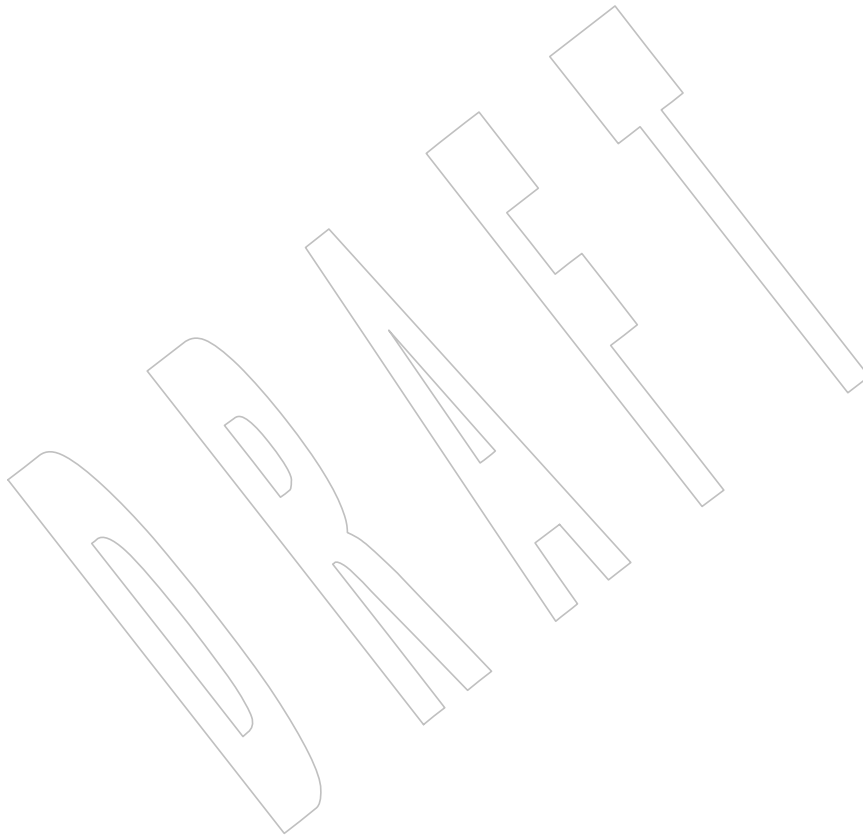
29123 **DESCRIPTION**
29124 Refer to *open()*.



29125 **NAME**
29126 opendir — open directory associated with file descriptor

29127 **SYNOPSIS**
29128 #include <dirent.h>
29129 DIR *opendir(const char *dirname);

29130 **DESCRIPTION**
29131 Refer to *fdopendir()*.



29132 **NAME**
29133 `openlog` — open a connection to the logging facility

29134 **SYNOPSIS**

29135 XSI `#include <syslog.h>`
29136 `void openlog(const char *ident, int logopt, int facility);`

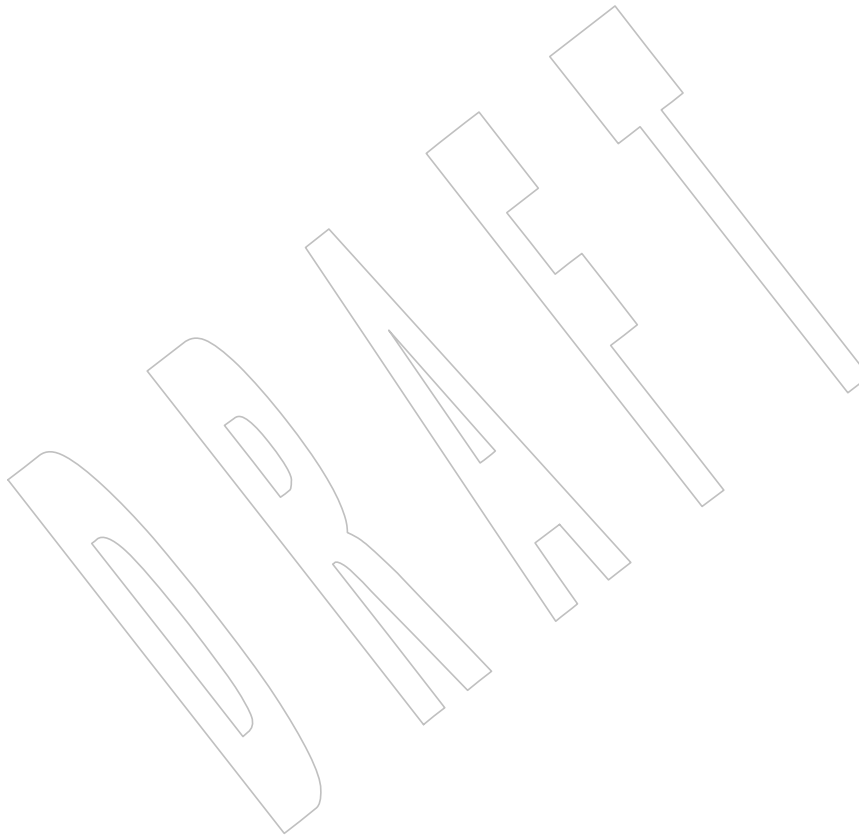
29137 **DESCRIPTION**

29138 Refer to [*closelog\(\)*](#).

29139 **NAME**
29140 `optarg, opterr, optind, optopt` — options parsing variables

29141 **SYNOPSIS**
29142 `#include <unistd.h>`
29143 `extern char *optarg;`
29144 `extern int opterr, optind, optopt;`

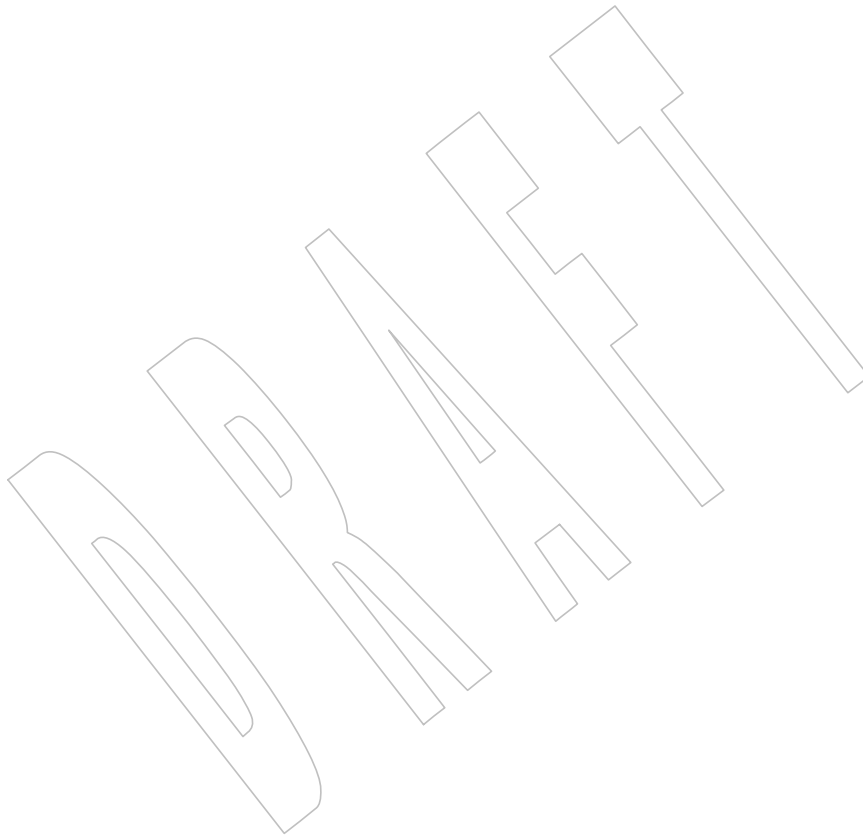
29145 **DESCRIPTION**
29146 Refer to *getopt()*.



29147 **NAME**
29148 pathconf — get configurable pathname variables

29149 **SYNOPSIS**
29150 #include <unistd.h>
29151 long pathconf(const char *path, int name);

29152 **DESCRIPTION**
29153 Refer to *fpathconf()*.



29154 **NAME**
 29155 `pause` — suspend the thread until a signal is received

29156 **SYNOPSIS**
 29157 `#include <unistd.h>`
 29158 `int pause(void);`

29159 **DESCRIPTION**
 29160 The `pause()` function shall suspend the calling thread until delivery of a signal whose action is
 29161 either to execute a signal-catching function or to terminate the process.

29162 If the action is to terminate the process, `pause()` shall not return.

29163 If the action is to execute a signal-catching function, `pause()` shall return after the signal-catching
 29164 function returns.

29165 **RETURN VALUE**
 29166 Since `pause()` suspends thread execution indefinitely unless interrupted by a signal, there is no
 29167 successful completion return value. A value of `-1` shall be returned and `errno` set to indicate the
 29168 error.

29169 **ERRORS**
 29170 The `pause()` function shall fail if:
 29171 [EINTR] A signal is caught by the calling process and control is returned from the
 29172 signal-catching function.

29173 **EXAMPLES**
 29174 None.

29175 **APPLICATION USAGE**
 29176 Many common uses of `pause()` have timing windows. The scenario involves checking a
 29177 condition related to a signal and, if the signal has not occurred, calling `pause()`. When the signal
 29178 occurs between the check and the call to `pause()`, the process often blocks indefinitely. The
 29179 `sigprocmask()` and `sigsuspend()` functions can be used to avoid this type of problem.

29180 **RATIONALE**
 29181 None.

29182 **FUTURE DIRECTIONS**
 29183 None.

29184 **SEE ALSO**
 29185 `sigsuspend()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<unistd.h>`

29186 **CHANGE HISTORY**
 29187 First released in Issue 1. Derived from Issue 1 of the SVID.

29188 **Issue 5**
 29189 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

29190 **Issue 6**
 29191 The APPLICATION USAGE section is added.

29192 **NAME**

29193 pclose — close a pipe stream to or from a process

29194 **SYNOPSIS**

```
29195 CX #include <stdio.h>
29196 int pclose(FILE *stream);
```

29197 **DESCRIPTION**

29198 The *pclose()* function shall close a stream that was opened by *popen()*, wait for the command to
 29199 terminate, and return the termination status of the process that was running the command
 29200 language interpreter. However, if a call caused the termination status to be unavailable to
 29201 *pclose()*, then *pclose()* shall return -1 with *errno* set to [ECHILD] to report this situation. This can
 29202 happen if the application calls one of the following functions:

- 29203 • *wait()*
- 29204 • *waitpid()* with a *pid* argument less than or equal to 0 or equal to the process ID of the
 29205 command line interpreter
- 29206 • Any other function not defined in this volume of IEEE Std 1003.1-200x that could do one of
 29207 the above

29208 In any case, *pclose()* shall not return before the child process created by *popen()* has terminated.

29209 If the command language interpreter cannot be executed, the child termination status returned
 29210 by *pclose()* shall be as if the command language interpreter terminated using *exit(127)* or
 29211 *_exit(127)*.

29212 The *pclose()* function shall not affect the termination status of any child of the calling process
 29213 other than the one created by *popen()* for the associated stream.

29214 If the argument *stream* to *pclose()* is not a pointer to a stream created by *popen()*, the result of
 29215 *pclose()* is undefined.

29216 **RETURN VALUE**

29217 Upon successful return, *pclose()* shall return the termination status of the command language
 29218 interpreter. Otherwise, *pclose()* shall return -1 and set *errno* to indicate the error.

29219 **ERRORS**

29220 The *pclose()* function shall fail if:

29221 [ECHILD] The status of the child process could not be obtained, as described above.

29222 **EXAMPLES**

29223 None.

29224 **APPLICATION USAGE**

29225 None.

29226 **RATIONALE**

29227 There is a requirement that *pclose()* not return before the child process terminates. This is
 29228 intended to disallow implementations that return [EINTR] if a signal is received while waiting.
 29229 If *pclose()* returned before the child terminated, there would be no way for the application to
 29230 discover which child used to be associated with the stream, and it could not do the cleanup
 29231 itself.

29232 If the stream pointed to by *stream* was not created by *popen()*, historical implementations of
 29233 *pclose()* return -1 without setting *errno*. To avoid requiring *pclose()* to set *errno* in this case,

29234 IEEE Std 1003.1-200x makes the behavior unspecified. An application should not use *pclose()* to
 29235 close any stream that was not created by *popen()*.

29236 Some historical implementations of *pclose()* either block or ignore the signals SIGINT, SIGQUIT,
 29237 and SIGHUP while waiting for the child process to terminate. Since this behavior is not
 29238 described for the *pclose()* function in IEEE Std 1003.1-200x, such implementations are not
 29239 conforming. Also, some historical implementations return [EINTR] if a signal is received, even
 29240 though the child process has not terminated. Such implementations are also considered non-
 29241 conforming.

29242 Consider, for example, an application that uses:

```
29243 popen("command", "r")
```

29244 to start *command*, which is part of the same application. The parent writes a prompt to its
 29245 standard output (presumably the terminal) and then reads from the *popen()*ed stream. The child
 29246 reads the response from the user, does some transformation on the response (pathname
 29247 expansion, perhaps) and writes the result to its standard output. The parent process reads the
 29248 result from the pipe, does something with it, and prints another prompt. The cycle repeats.
 29249 Assuming that both processes do appropriate buffer flushing, this would be expected to work.

29250 To conform to IEEE Std 1003.1-200x, *pclose()* must use *waitpid()*, or some similar function,
 29251 instead of *wait()*.

29252 The code sample below illustrates how the *pclose()* function might be implemented on a system
 29253 conforming to IEEE Std 1003.1-200x.

```
29254 int pclose(FILE *stream)
29255 {
29256     int stat;
29257     pid_t pid;
29258     pid = <pid for process created for stream by popen(>
29259     (void) fclose(stream);
29260     while (waitpid(pid, &stat, 0) == -1) {
29261         if (errno != EINTR){
29262             stat = -1;
29263             break;
29264         }
29265     }
29266     return(stat);
29267 }
```

29268 FUTURE DIRECTIONS

29269 None.

29270 SEE ALSO

29271 *fork()*, *popen()*, *waitpid()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>

29272 CHANGE HISTORY

29273 First released in Issue 1. Derived from Issue 1 of the SVID.

29274 **NAME**
 29275 `perror` — write error messages to standard error

29276 **SYNOPSIS**
 29277 `#include <stdio.h>`

29278 `void perror(const char *s);`

29279 **DESCRIPTION**

29280 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 29281 conflict between the requirements described here and the ISO C standard is unintentional. This
 29282 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

29283 The `perror()` function shall map the error number accessed through the symbol `errno` to a
 29284 language-dependent error message, which shall be written to the standard error stream as
 29285 follows:

- 29286 • First (if `s` is not a null pointer and the character pointed to by `s` is not the null byte), the
 29287 string pointed to by `s` followed by a colon and a `<space>`.
- 29288 • Then an error message string followed by a `<newline>`.

29289 The contents of the error message strings shall be the same as those returned by `strerror()` with
 29290 argument `errno`.

29291 CX The `perror()` function shall mark the file associated with the standard error stream as having
 29292 been written (`st_ctime`, `st_mtime` marked for update) at some time between its successful
 29293 completion and `exit()`, `abort()`, or the completion of `fflush()` or `fclose()` on `stderr`.

29294 The `perror()` function shall not change the orientation of the standard error stream.

29295 **RETURN VALUE**

29296 The `perror()` function shall not return a value.

29297 **ERRORS**

29298 No errors are defined.

29299 **EXAMPLES**

29300 **Printing an Error Message for a Function**

29301 The following example replaces `bufptr` with a buffer that is the necessary size. If an error occurs,
 29302 the `perror()` function prints a message and the program exits.

```

29303 #include <stdio.h>
29304 #include <stdlib.h>
29305 ...
29306 char *bufptr;
29307 size_t szbuf;
29308 ...
29309 if ((bufptr = malloc(szbuf)) == NULL) {
29310     perror("malloc"); exit(2);
29311 }
29312 ...
  
```


perror()29313
29314
29315
29316
29317
29318
29319
29320
29321
29322
29323
29324
29325
29326
29327**APPLICATION USAGE**

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO*psiginfo()*, *strerror()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>**CHANGE HISTORY**

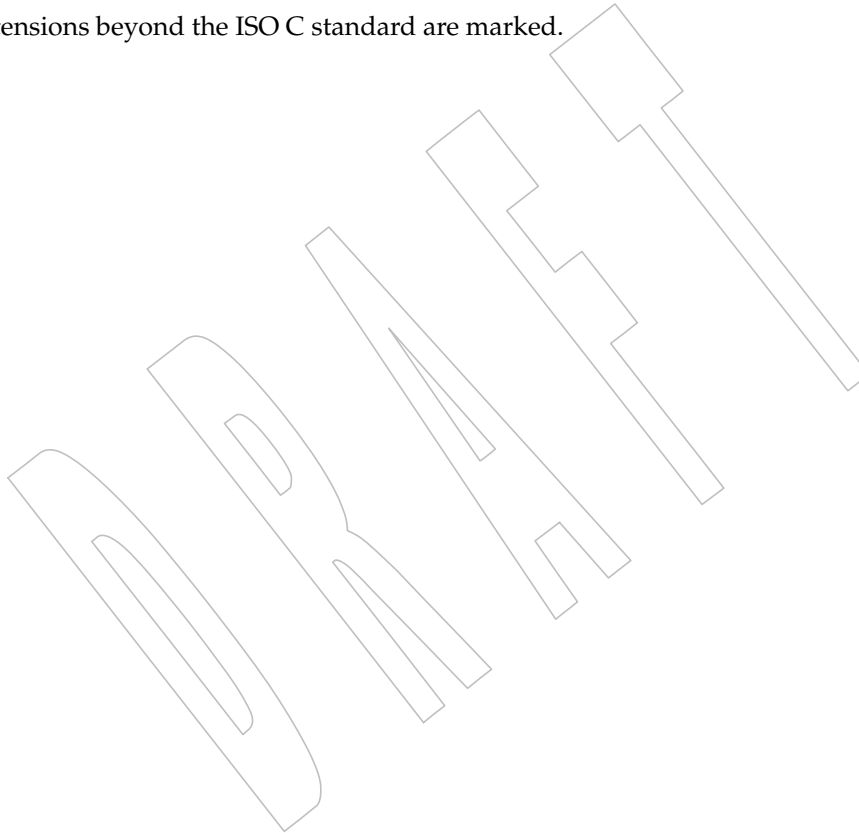
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

A paragraph is added to the DESCRIPTION indicating that *perror()* does not change the orientation of the standard error stream.

Issue 6

Extensions beyond the ISO C standard are marked.



29328 **NAME**
 29329 pipe — create an interprocess channel

29330 **SYNOPSIS**
 29331 #include <unistd.h>
 29332 int pipe(int *fildes*[2]);

29333 **DESCRIPTION**
 29334 The *pipe()* function shall create a pipe and place two file descriptors, one each into the
 29335 arguments *fildes*[0] and *fildes*[1], that refer to the open file descriptions for the read and write
 29336 ends of the pipe. Their integer values shall be the two lowest available at the time of the *pipe()*
 29337 call. The *O_NONBLOCK* and *FD_CLOEXEC* flags shall be clear on both file descriptors. (The
 29338 *fcntl()* function can be used to set both these flags.)

29339 Data can be written to the file descriptor *fildes*[1] and read from the file descriptor *fildes*[0]. A
 29340 read on the file descriptor *fildes*[0] shall access data written to the file descriptor *fildes*[1] on a
 29341 first-in-first-out basis. It is unspecified whether *fildes*[0] is also open for writing and whether
 29342 *fildes*[1] is also open for reading.

29343 A process has the pipe open for reading (correspondingly writing) if it has a file descriptor open
 29344 that refers to the read end, *fildes*[0] (write end, *fildes*[1]).

29345 The pipe's user ID shall be set to the effective user ID of the calling process.

29346 The pipe's group ID shall be set to the effective group ID of the calling process.

29347 Upon successful completion, *pipe()* shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
 29348 fields of the pipe.

29349 **RETURN VALUE**
 29350 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and *errno* set to
 29351 indicate the error.

29352 **ERRORS**
 29353 The *pipe()* function shall fail if:
 29354 [EMFILE] All, or all but one, of the file descriptors available to the process are currently
 29355 open.
 29356 [ENFILE] The number of simultaneously open files in the system would exceed a
 29357 system-imposed limit.

29358 EXAMPLES

29359 Using a Pipe to Pass Data Between a Parent Process and a Child Process

29360 The following example demonstrates the use of a pipe to transfer data between a parent process
 29361 and a child process. Error handling is excluded, but otherwise this code demonstrates good
 29362 practice when using pipes: after the *fork()* the two processes close the unused ends of the pipe
 29363 before they commence transferring data.

```
29364 #include <stdlib.h>
29365 #include <unistd.h>
29366 ...
29367 int fildes[2];
29368 const int BSIZE = 100;
29369 char buf[BSIZE];
29370 ssize_t nbytes;
```

```

29371     int status;
29372
29372     status = pipe(fildes);
29373     if (status == -1 ) {
29374         /* an error occurred */
29375         ...
29376     }
29377
29377     switch (fork()) {
29378     case -1: /* Handle error */
29379         break;
29380
29380     case 0: /* Child - reads from pipe */
29381         close(fildes[1]); /* Write end is unused */
29382         nbytes = read(fildes[0], buf, BSIZE); /* Get data from pipe */
29383         /* At this point, a further read would see end of file ... */
29384         close(fildes[0]); /* Finished with pipe */
29385         exit(EXIT_SUCCESS);
29386
29386     default: /* Parent - writes to pipe */
29387         close(fildes[0]); /* Read end is unused */
29388         write(fildes[1], "Hello world\n", 12); /* Write data on pipe */
29389         close(fildes[1]); /* Child will see EOF */
29390         exit(EXIT_SUCCESS);
29391     }

```

APPLICATION USAGE

None.

RATIONALE

The wording carefully avoids using the verb “to open” in order to avoid any implication of use of *open()*; see also *write()*.

FUTURE DIRECTIONS

None.

SEE ALSO

fcntl(), *read()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-200x, *<fcntl.h>*, *<unistd.h>*

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to indicate that certain dispositions of *fildes[0]* and *fildes[1]* are unspecified.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/65 is applied, adding the example to the EXAMPLES section.

Issue 7

SD5-XSH-ERN-156 is applied, updating the DESCRIPTION to state the setting of the pipe’s user ID and group ID.

29414 **NAME**

29415 poll — input/output multiplexing

29416 **SYNOPSIS**

29417 #include <poll.h>

29418 int poll(struct pollfd *fds*[], nfds_t *nfds*, int *timeout*);29419 **DESCRIPTION**

29420 The *poll()* function provides applications with a mechanism for multiplexing input/output over
 29421 a set of file descriptors. For each member of the array pointed to by *fds*, *poll()* shall examine the
 29422 given file descriptor for the event(s) specified in *events*. The number of **pollfd** structures in the
 29423 *fds* array is specified by *nfds*. The *poll()* function shall identify those file descriptors on which an
 29424 application can read or write data, or on which certain events have occurred.

29425 The *fds* argument specifies the file descriptors to be examined and the events of interest for each
 29426 file descriptor. It is a pointer to an array with one member for each open file descriptor of
 29427 interest. The array's members are **pollfd** structures within which *fd* specifies an open file
 29428 descriptor and *events* and *revents* are bitmasks constructed by OR'ing a combination of the
 29429 following event flags:

29430	POLLIN	Data other than high-priority data may be read without blocking.
29431	OB XSR	For STREAMS, this flag is set in <i>revents</i> even if the message is of zero length.
29432		This flag shall be equivalent to POLLRDNORM POLLRDBAND.
29433	POLLRDNORM	Normal data may be read without blocking.
29434	OB XSR	For STREAMS, data on priority band 0 may be read without blocking. This
29435		flag is set in <i>revents</i> even if the message is of zero length.
29436	POLLRDBAND	Priority data may be read without blocking.
29437	OB XSR	For STREAMS, data on priority bands greater than 0 may be read without
29438		blocking. This flag is set in <i>revents</i> even if the message is of zero length.
29439	POLLPRI	High-priority data may be read without blocking.
29440	OB XSR	For STREAMS, this flag is set in <i>revents</i> even if the message is of zero length.
29441	POLLOUT	Normal data may be written without blocking.
29442	OB XSR	For STREAMS, data on priority band 0 may be written without blocking.
29443	POLLWRNORM	Equivalent to POLLOUT.
29444	POLLWRBAND	Priority data may be written.
29445	OB XSR	For STREAMS, data on priority bands greater than 0 may be written without
29446		blocking. If any priority band has been written to on this STREAM, this event
29447		only examines bands that have been written to at least once.
29448	POLLERR	An error has occurred on the device or stream. This flag is only valid in the
29449		<i>revents</i> bitmask; it shall be ignored in the <i>events</i> member.
29450	POLLHUP	The device has been disconnected. This event and POLLOUT are mutually-
29451		exclusive; a stream can never be writable if a hangup has occurred. However,
29452		this event and POLLIN, POLLRDNORM, POLLRDBAND, or POLLPRI are
29453		not mutually-exclusive. This flag is only valid in the <i>revents</i> bitmask; it shall be
29454		ignored in the <i>events</i> member.

poll()

29455 POLLNVAL The specified *fd* value is invalid. This flag is only valid in the *revents* member;
 29456 it shall ignored in the *events* member.

29457 The significance and semantics of normal, priority, and high-priority data are file and device-
 29458 specific.

29459 If the value of *fd* is less than 0, *events* shall be ignored, and *revents* shall be set to 0 in that entry on
 29460 return from *poll()*.

29461 In each **pollfd** structure, *poll()* shall clear the *revents* member, except that where the application
 29462 requested a report on a condition by setting one of the bits of *events* listed above, *poll()* shall set
 29463 the corresponding bit in *revents* if the requested condition is true. In addition, *poll()* shall set the
 29464 POLLHUP, POLLERR, and POLLNVAL flag in *revents* if the condition is true, even if the
 29465 application did not set the corresponding bit in *events*.

29466 If none of the defined events have occurred on any selected file descriptor, *poll()* shall wait at
 29467 least *timeout* milliseconds for an event to occur on any of the selected file descriptors. If the value
 29468 of *timeout* is 0, *poll()* shall return immediately. If the value of *timeout* is -1, *poll()* shall block until
 29469 a requested event occurs or until the call is interrupted.

29470 Implementations may place limitations on the granularity of timeout intervals. If the requested
 29471 timeout interval requires a finer granularity than the implementation supports, the actual
 29472 timeout interval shall be rounded up to the next supported value.

29473 The *poll()* function shall not be affected by the O_NONBLOCK flag.

29474 The *poll()* function shall support regular files, terminal and pseudo-terminal devices, FIFOs,
 29475 pipes, sockets and STREAMS-based files. The behavior of *poll()* on elements of *fds* that refer to
 29476 other types of file is unspecified.

OB XSR

29477 Regular files shall always poll TRUE for reading and writing.

29478 A file descriptor for a socket that is listening for connections shall indicate that it is ready for
 29479 reading, once connections are available. A file descriptor for a socket that is connecting
 29480 asynchronously shall indicate that it is ready for writing, once a connection has been established.

RETURN VALUE

29481 Upon successful completion, *poll()* shall return a non-negative value. A positive value indicates
 29482 the total number of file descriptors that have been selected (that is, file descriptors for which the
 29483 *revents* member is non-zero). A value of 0 indicates that the call timed out and no file descriptors
 29484 have been selected. Upon failure, *poll()* shall return -1 and set *errno* to indicate the error.

ERRORS

29486 The *poll()* function shall fail if:

29487

29488 [EAGAIN] The allocation of internal data structures failed but a subsequent request may
 29489 succeed.

29490 [EINTR] A signal was caught during *poll()*.

29491 OB XSR [EINVAL] The *nfds* argument is greater than {OPEN_MAX}, or one of the *fd* members
 29492 refers to a STREAM or multiplexer that is linked (directly or indirectly)
 29493 downstream from a multiplexer.

EXAMPLES

Checking for Events on a Stream

The following example opens a pair of STREAMS devices and then waits for either one to become writable. This example proceeds as follows:

1. Sets the *timeout* parameter to 500 milliseconds.
2. Opens the STREAMS devices */dev/dev0* and */dev/dev1*, and then polls them, specifying POLLOUT and POLLWRBAND as the events of interest.
 The STREAMS device names */dev/dev0* and */dev/dev1* are only examples of how STREAMS devices can be named; STREAMS naming conventions may vary among systems conforming to the IEEE Std 1003.1-200x.
3. Uses the *ret* variable to determine whether an event has occurred on either of the two STREAMS. The *poll()* function is given 500 milliseconds to wait for an event to occur (if it has not occurred prior to the *poll()* call).
4. Checks the returned value of *ret*. If a positive value is returned, one of the following can be done:
 - a. Priority data can be written to the open STREAM on priority bands greater than 0, because the POLLWRBAND event occurred on the open STREAM (*fds[0]* or *fds[1]*).
 - b. Data can be written to the open STREAM on priority-band 0, because the POLLOUT event occurred on the open STREAM (*fds[0]* or *fds[1]*).
5. If the returned value is not a positive value, permission to write data to the open STREAM (on any priority band) is denied.
6. If the POLLHUP event occurs on the open STREAM (*fds[0]* or *fds[1]*), the device on the open STREAM has disconnected.

```

29517 #include <stropts.h>
29518 #include <poll.h>
29519 ...
29520 struct pollfd fds[2];
29521 int timeout_msecs = 500;
29522 int ret;
29523 int i;
29524 /* Open STREAMS device. */
29525 fds[0].fd = open("/dev/dev0", ...);
29526 fds[1].fd = open("/dev/dev1", ...);
29527 fds[0].events = POLLOUT | POLLWRBAND;
29528 fds[1].events = POLLOUT | POLLWRBAND;
29529 ret = poll(fds, 2, timeout_msecs);
29530 if (ret > 0) {
29531     /* An event on one of the fds has occurred. */
29532     for (i=0; i<2; i++) {
29533         if (fds[i].revents & POLLWRBAND) {
29534             /* Priority data may be written on device number i. */
29535             ...
29536         }
29537         if (fds[i].revents & POLLOUT) {
29538             /* Data may be written on device number i. */
29539             ...

```

```

29540         }
29541         if (fds[i].revents & POLLHUP) {
29542             /* A hangup has occurred on device number i. */
29543             ...
29544         }
29545     }
29546 }

```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.6 (on page 38), *getmsg()*, *putmsg()*, *read()*, *select()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<poll.h>**, **<stropts.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The description of POLLWRBAND is updated.

Issue 6

Text referring to sockets is added to the DESCRIPTION.

Functionality relating to the XSI STREAMS Option Group is marked.

The Open Group Corrigendum U055/3 is applied, updating the DESCRIPTION of POLLWRBAND.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/66 is applied, correcting the spacing in the EXAMPLES section.

Issue 7The *poll()* function is moved from the XSI option to the Base.

Functionality relating to the XSI STREAMS option is marked obsolescent.

29571 **NAME**
 29572 popen — initiate pipe streams to or from a process

29573 **SYNOPSIS**

29574 CX `#include <stdio.h>`
 29575 `FILE *popen(const char *command, const char *mode);`

29576 **DESCRIPTION**

29577 The *popen()* function shall execute the command specified by the string *command*. It shall create
 29578 a pipe between the calling program and the executed command, and shall return a pointer to a
 29579 stream that can be used to either read from or write to the pipe.

29580 The environment of the executed command shall be as if a child process were created within the
 29581 *popen()* call using the *fork()* function, and the child invoked the *sh* utility using the call:

29582 `execl(shell_path, "sh", "-c", command, (char *)0);`

29583 where *shell_path* is an unspecified pathname for the *sh* utility.

29584 The *popen()* function shall ensure that any streams from previous *popen()* calls that remain open
 29585 in the parent process are closed in the new child process.

29586 The *mode* argument to *popen()* is a string that specifies I/O mode:

- 29587 1. If *mode* is *r*, when the child process is started, its file descriptor `STDOUT_FILENO` shall be
 29588 the writable end of the pipe, and the file descriptor *fileno(stream)* in the calling process,
 29589 where *stream* is the stream pointer returned by *popen()*, shall be the readable end of the
 29590 pipe.
- 29591 2. If *mode* is *w*, when the child process is started its file descriptor `STDIN_FILENO` shall be
 29592 the readable end of the pipe, and the file descriptor *fileno(stream)* in the calling process,
 29593 where *stream* is the stream pointer returned by *popen()*, shall be the writable end of the
 29594 pipe.
- 29595 3. If *mode* is any other value, the result is unspecified.

29596 After *popen()*, both the parent and the child process shall be capable of executing independently
 29597 before either terminates.

29598 Pipe streams are byte-oriented.

29599 **RETURN VALUE**

29600 Upon successful completion, *popen()* shall return a pointer to an open stream that can be used to
 29601 read or write to the pipe. Otherwise, it shall return a null pointer and may set *errno* to indicate
 29602 the error.

29603 **ERRORS**

29604 The *popen()* function may fail if:

29605 [EMFILE] {FOPEN_MAX} or {STREAM_MAX} streams are currently open in the calling
 29606 process.

29607 [EINVAL] The *mode* argument is invalid.

29608 The *popen()* function may also set *errno* values as described by *fork()* or *pipe()*.

EXAMPLES**Using popen() to Obtain a List of Files from the ls Utility**

The following example demonstrates the use of *popen()* and *pclose()* to execute the command *ls** in order to obtain a list of files in the current directory:

```

29613 #include <stdio.h>
29614 ...
29615 FILE *fp;
29616 int status;
29617 char path[PATH_MAX];
29618 fp = popen("ls *", "r");
29619 if (fp == NULL)
29620     /* Handle error */;
29621 while (fgets(path, PATH_MAX, fp) != NULL)
29622     printf("%s", path);
29623 status = pclose(fp);
29624 if (status == -1) {
29625     /* Error reported by pclose() */
29626     ...
29627 } else {
29628     /* Use macros described under wait() to inspect 'status' in order
29629     to determine success/failure of command executed by popen() */
29630     ...
29631 }

```

APPLICATION USAGE

Since open files are shared, a mode *r* command can be used as an input filter and a mode *w* command as an output filter.

Buffered reading before opening an input filter may leave the standard input of that filter mispositioned. Similar problems with an output filter may be prevented by careful buffer flushing; for example, with *fflush()*.

A stream opened by *popen()* should be closed by *pclose()*.

The behavior of *popen()* is specified for values of *mode* of *r* and *w*. Other modes such as *rb* and *wb* might be supported by specific implementations, but these would not be portable features. Note that historical implementations of *popen()* only check to see if the first character of *mode* is *r*. Thus, a *mode* of *robert the robot* would be treated as *mode r*, and a *mode* of *anything else* would be treated as *mode w*.

If the application calls *waitpid()* or *waitid()* with a *pid* argument greater than 0, and it still has a stream that was called with *popen()* open, it must ensure that *pid* does not refer to the process started by *popen()*.

To determine whether or not the environment specified in the Shell and Utilities volume of IEEE Std 1003.1-200x is present, use the function call:

```
sysconf(_SC_2_VERSION)
```

(See *sysconf()*).

29651
29652
29653
29654
29655

29656
29657
29658

29659
29660

29661
29662
29663

29664
29665

29666
29667

29668
29669
29670

29671
29672
29673

29674
29675
29676

RATIONALE

The *popen()* function should not be used by programs that have set user (or group) ID privileges. The *fork()* and *exec* family of functions (except *execlp()* and *execvp()*), should be used instead. This prevents any unforeseen manipulation of the environment of the user that could cause execution of commands not anticipated by the calling program.

If the original and *popen()*ed processes both intend to read or write or read and write a common file, and either will be using FILE-type C functions (*fread()*, *fwrite()*, and so on), the rules for sharing file handles must be observed (see [Section 2.5.1](#) (on page 35)).

FUTURE DIRECTIONS

None.

SEE ALSO

pclose(), *pipe()*, *sysconf()*, *system()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdio.h>`, the Shell and Utilities volume of IEEE Std 1003.1-200x, *sh*

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

A statement is added to the DESCRIPTION indicating that pipe streams are byte-oriented.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The optional [EMFILE] error condition is added.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/67 is applied, adding the example to the EXAMPLES section.

Issue 7

Austin Group Interpretation 1003.1-2001 #029 is applied, clarifying the values for *mode* in the DESCRIPTION.

29677 **NAME**29678 posix_fadvise — file advisory information (**ADVANCED REALTIME**)29679 **SYNOPSIS**

```
29680 ADV #include <fcntl.h>
29681 int posix_fadvise(int fd, off_t offset, off_t len, int advice);
```

29682 **DESCRIPTION**

29683 The *posix_fadvise()* function shall advise the implementation on the expected behavior of the
 29684 application with respect to the data in the file associated with the open file descriptor, *fd*, starting
 29685 at *offset* and continuing for *len* bytes. The specified range need not currently exist in the file. If *len*
 29686 is zero, all data following *offset* is specified. The implementation may use this information to
 29687 optimize handling of the specified data. The *posix_fadvise()* function shall have no effect on the
 29688 semantics of other operations on the specified data, although it may affect the performance of
 29689 other operations.

29690 The advice to be applied to the data is specified by the *advice* parameter and may be one of the
 29691 following values:

29692 **POSIX_FADV_NORMAL**

29693 Specifies that the application has no advice to give on its behavior with respect to the
 29694 specified data. It is the default characteristic if no advice is given for an open file.

29695 **POSIX_FADV_SEQUENTIAL**

29696 Specifies that the application expects to access the specified data sequentially from lower
 29697 offsets to higher offsets.

29698 **POSIX_FADV_RANDOM**

29699 Specifies that the application expects to access the specified data in a random order.

29700 **POSIX_FADV_WILLNEED**

29701 Specifies that the application expects to access the specified data in the near future.

29702 **POSIX_FADV_DONTNEED**

29703 Specifies that the application expects that it will not access the specified data in the near
 29704 future.

29705 **POSIX_FADV_NOREUSE**

29706 Specifies that the application expects to access the specified data once and then not reuse it
 29707 thereafter.

29708 These values are defined in **<fcntl.h>**.

29709 **RETURN VALUE**

29710 Upon successful completion, *posix_fadvise()* shall return zero; otherwise, an error number shall
 29711 be returned to indicate the error.

29712 **ERRORS**

29713 The *posix_fadvise()* function shall fail if:

- | | | |
|-------|----------|--|
| 29714 | [EBADF] | The <i>fd</i> argument is not a valid file descriptor. |
| 29715 | [EINVAL] | The value of <i>advice</i> is invalid, or the value of <i>len</i> is less than zero. |
| 29716 | [ESPIPE] | The <i>fd</i> argument is associated with a pipe or FIFO. |

29717
29718

29719
29720
29721

29722
29723

29724
29725

29726
29727

29728
29729

29730

29731
29732
29733
29734

29735
29736
29737

EXAMPLES

None.

APPLICATION USAGE

The *posix_fadvise()* function is part of the Advisory Information option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_madvise(), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/68 is applied, changing the function prototype in the SYNOPSIS section. The previous prototype was not large file-aware, and the standard developers felt it acceptable to make this change before implementations of this function become widespread.

Issue 7

Austin Group Interpretation 1003.1-2001 #024 is applied, changing the definition of the [EINVAL] error.

29738 **NAME**29739 posix_fallocate — file space control (**ADVANCED REALTIME**)29740 **SYNOPSIS**

```
29741 ADV #include <fcntl.h>
29742 int posix_fallocate(int fd, off_t offset, off_t len);
```

29743 **DESCRIPTION**

29744 The *posix_fallocate()* function shall ensure that any required storage for regular file data starting
 29745 at *offset* and continuing for *len* bytes is allocated on the file system storage media. If
 29746 *posix_fallocate()* returns successfully, subsequent writes to the specified file data shall not fail due
 29747 to the lack of free space on the file system storage media.

29748 If the *offset+len* is beyond the current file size, then *posix_fallocate()* shall adjust the file size to
 29749 *offset+len*. Otherwise, the file size shall not be changed.

29750 It is implementation-defined whether a previous *posix_fadvise()* call influences allocation
 29751 strategy.

29752 Space allocated via *posix_fallocate()* shall be freed by a successful call to *creat()* or *open()* that
 29753 truncates the size of the file. Space allocated via *posix_fallocate()* may be freed by a successful call
 29754 to *truncate()* that reduces the file size to a size smaller than *offset+len*.

29755 **RETURN VALUE**

29756 Upon successful completion, *posix_fallocate()* shall return zero; otherwise, an error number shall
 29757 be returned to indicate the error.

29758 **ERRORS**

29759 The *posix_fallocate()* function shall fail if:

29760	[EBADF]	The <i>fd</i> argument is not a valid file descriptor.
29761	[EBADF]	The <i>fd</i> argument references a file that was opened without write permission.
29762	[EFBIG]	The value of <i>offset+len</i> is greater than the maximum file size.
29763	[EINTR]	A signal was caught during execution.
29764	[EINVAL]	The <i>len</i> argument is less than or equal to zero, or the <i>offset</i> argument is less 29765 than zero, or the underlying file system does not support this operation.
29766	[EIO]	An I/O error occurred while reading from or writing to a file system.
29767	[ENODEV]	The <i>fd</i> argument does not refer to a regular file.
29768	[ENOSPC]	There is insufficient free space remaining on the file system storage media.
29769	[ESPIPE]	The <i>fd</i> argument is associated with a pipe or FIFO.

29770 **EXAMPLES**

29771 None.

29772 **APPLICATION USAGE**

29773 The *posix_fallocate()* function is part of the Advisory Information option and need not be
 29774 provided on all implementations.

29775
29776
29777
29778
29779
29780
29781
29782
29783
29784
29785
29786
29787
29788
29789
29790
29791

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

creat(), *ftruncate()*, *open()*, *unlink()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<fcntl.h>`

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/69 is applied, changing the function prototype in the SYNOPSIS section. The previous prototype was not large file-aware, and the standard developers felt it acceptable to make this change before implementations of this function become widespread.

Issue 7

Austin Group Interpretations 1003.1-2001 #022 and #024 are applied, changing the definition of the [EINVAL] error.

DRAFT

29792 **NAME**

29793 `posix_madvise` — memory advisory information and alignment control (**ADVANCED**
 29794 **REALTIME**)

29795 **SYNOPSIS**

```
29796 ADV #include <sys/mman.h>
29797 int posix_madvise(void *addr, size_t len, int advice);
```

29798 **DESCRIPTION**

29799 The `posix_madvise()` function shall advise the implementation on the expected behavior of the
 29800 application with respect to the data in the memory starting at address `addr`, and continuing for
 29801 `len` bytes. The implementation may use this information to optimize handling of the specified
 29802 data. The `posix_madvise()` function shall have no effect on the semantics of access to memory in
 29803 the specified range, although it may affect the performance of access.

29804 The implementation may require that `addr` be a multiple of the page size, which is the value
 29805 returned by `sysconf()` when the name value `_SC_PAGESIZE` is used.

29806 The advice to be applied to the memory range is specified by the `advice` parameter and may be
 29807 one of the following values:

29808 **POSIX_MADV_NORMAL**

29809 Specifies that the application has no advice to give on its behavior with respect to the
 29810 specified range. It is the default characteristic if no advice is given for a range of memory.

29811 **POSIX_MADV_SEQUENTIAL**

29812 Specifies that the application expects to access the specified range sequentially from lower
 29813 addresses to higher addresses.

29814 **POSIX_MADV_RANDOM**

29815 Specifies that the application expects to access the specified range in a random order.

29816 **POSIX_MADV_WILLNEED**

29817 Specifies that the application expects to access the specified range in the near future.

29818 **POSIX_MADV_DONTNEED**

29819 Specifies that the application expects that it will not access the specified range in the near
 29820 future.

29821 These values are defined in the `<sys/mman.h>` header.

29822 **RETURN VALUE**

29823 Upon successful completion, `posix_madvise()` shall return zero; otherwise, an error number shall
 29824 be returned to indicate the error.

29825 **ERRORS**

29826 The `posix_madvise()` function shall fail if:

29827 [EINVAL] The value of `advice` is invalid.

29828 [ENOMEM] Addresses in the range starting at `addr` and continuing for `len` bytes are partly
 29829 or completely outside the range allowed for the address space of the calling
 29830 process.

29831 The *posix_madvise()* function may fail if:

29832 [EINVAL] The value of *addr* is not a multiple of the value returned by *sysconf()* when the
29833 name value *_SC_PAGESIZE* is used.

29834 [EINVAL] The value of *len* is zero.

29835 **EXAMPLES**

29836 None.

29837 **APPLICATION USAGE**

29838 The *posix_madvise()* function is part of the Advisory Information option and need not be
29839 provided on all implementations.

29840 **RATIONALE**

29841 None.

29842 **FUTURE DIRECTIONS**

29843 None.

29844 **SEE ALSO**

29845 *mmap()*, *posix_fadvise()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-200x,
29846 <sys/mman.h>

29847 **CHANGE HISTORY**

29848 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29849 IEEE PASC Interpretation 1003.1 #102 is applied.

29850 **NAME**

29851 `posix_mem_offset` — find offset and length of a mapped typed memory block (**ADVANCED**
 29852 **REALTIME**)

29853 **SYNOPSIS**

```
29854 TYM #include <sys/mman.h>
29855
29856 int posix_mem_offset(const void *restrict addr, size_t len,
29857                    off_t *restrict off, size_t *restrict contig_len,
29858                    int *restrict fildes);
```

29858 **DESCRIPTION**

29859 The `posix_mem_offset()` function shall return in the variable pointed to by `off` a value that
 29860 identifies the offset (or location), within a memory object, of the memory block currently
 29861 mapped at `addr`. The function shall return in the variable pointed to by `fildes`, the descriptor used
 29862 (via `mmap()`) to establish the mapping which contains `addr`. If that descriptor was closed since
 29863 the mapping was established, the returned value of `fildes` shall be `-1`. The `len` argument specifies
 29864 the length of the block of the memory object the user wishes the offset for; upon return, the
 29865 value pointed to by `contig_len` shall equal either `len`, or the length of the largest contiguous block
 29866 of the memory object that is currently mapped to the calling process starting at `addr`, whichever
 29867 is smaller.

29868 If the memory object mapped at `addr` is a typed memory object, then if the `off` and `contig_len`
 29869 values obtained by calling `posix_mem_offset()` are used in a call to `mmap()` with a file descriptor
 29870 that refers to the same memory pool as `fildes` (either through the same port or through a different
 29871 port), and that was opened with neither the `POSIX_TYPED_MEM_ALLOCATE` nor the
 29872 `POSIX_TYPED_MEM_ALLOCATE_CONTIG` flag, the typed memory area that is mapped shall
 29873 be exactly the same area that was mapped at `addr` in the address space of the process that called
 29874 `posix_mem_offset()`.

29875 If the memory object specified by `fildes` is not a typed memory object, then the behavior of this
 29876 function is implementation-defined.

29877 **RETURN VALUE**

29878 Upon successful completion, the `posix_mem_offset()` function shall return zero; otherwise, the
 29879 corresponding error status value shall be returned.

29880 **ERRORS**

29881 The `posix_mem_offset()` function shall fail if:

29882 [EACCES] The process has not mapped a memory object supported by this function at
 29883 the given address `addr`.

29884 This function shall not return an error code of [EINTR].

29885 **EXAMPLES**

29886 None.

29887 **APPLICATION USAGE**

29888 None.

29889 **RATIONALE**

29890 None.

29891
29892
29893
29894
29895
29896
29897**FUTURE DIRECTIONS**

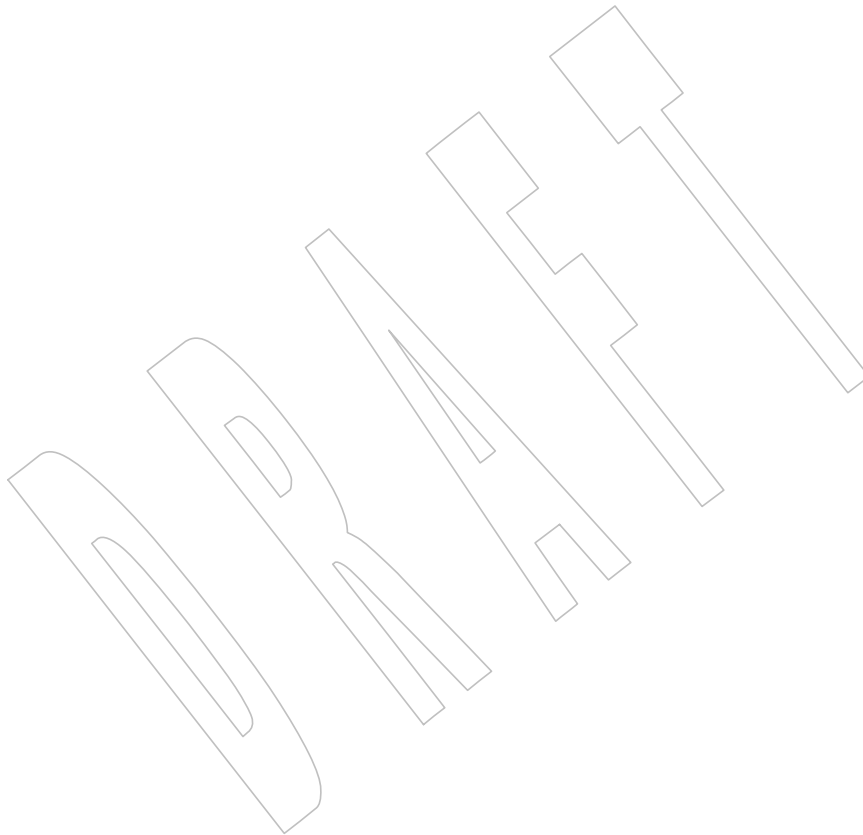
None.

SEE ALSO

mmap(), *posix_typed_mem_open()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<sys/mman.h>`

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.



29898 **NAME**29899 `posix_memalign` — aligned memory allocation (**ADVANCED REALTIME**)29900 **SYNOPSIS**

```
29901 ADV #include <stdlib.h>
29902 int posix_memalign(void **memptr, size_t alignment, size_t size);
```

29903 **DESCRIPTION**

29904 The `posix_memalign()` function shall allocate *size* bytes aligned on a boundary specified by
 29905 *alignment*, and shall return a pointer to the allocated memory in *memptr*. The value of *alignment*
 29906 shall be a power of two multiple of `sizeof(void *)`. Upon successful completion, the value
 29907 pointed to by *memptr* shall be a multiple of *alignment*.

29908 CX The `free()` function shall deallocate memory that has previously been allocated by
 29909 `posix_memalign()`.

29910 **RETURN VALUE**

29911 Upon successful completion, `posix_memalign()` shall return zero; otherwise, an error number
 29912 shall be returned to indicate the error.

29913 **ERRORS**

29914 The `posix_memalign()` function shall fail if:

- | | | |
|-------|----------|--|
| 29915 | [EINVAL] | The value of the alignment parameter is not a power of two multiple of |
| 29916 | | <code>sizeof(void *)</code> . |
| 29917 | [ENOMEM] | There is insufficient memory available with the requested alignment. |

29918 **EXAMPLES**

29919 None.

29920 **APPLICATION USAGE**

29921 The `posix_memalign()` function is part of the Advisory Information option and need not be
 29922 provided on all implementations.

29923 **RATIONALE**

29924 None.

29925 **FUTURE DIRECTIONS**

29926 None.

29927 **SEE ALSO**

29928 `free()`, `malloc()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdlib.h>`

29929 **CHANGE HISTORY**

29930 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29931 In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.

29932 **Issue 7**

29933 Austin Group Interpretation 1003.1-2001 #058 is applied, clarifying the value of the *alignment*
 29934 argument in the DESCRIPTION.

29935 **NAME**
 29936 posix_openpt — open a pseudo-terminal device

29937 **SYNOPSIS**

```
29938 XSI #include <stdlib.h>
29939 #include <fcntl.h>
29940 int posix_openpt(int oflag);
```

29941 **DESCRIPTION**

29942 The *posix_openpt()* function shall establish a connection between a master device for a pseudo-
 29943 terminal and a file descriptor. The file descriptor is used by other I/O functions that refer to that
 29944 pseudo-terminal.

29945 The file status flags and file access modes of the open file description shall be set according to
 29946 the value of *oflag*.

29947 Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list, defined
 29948 in *<fcntl.h>*:

29949 O_RDWR Open for reading and writing.
 29950 O_NOCTTY If set *posix_openpt()* shall not cause the terminal device to become the
 29951 controlling terminal for the process.

29952 The behavior of other values for the *oflag* argument is unspecified.

29953 **RETURN VALUE**

29954 Upon successful completion, the *posix_openpt()* function shall open a master pseudo-terminal
 29955 device and return a non-negative integer representing the lowest numbered unused file
 29956 descriptor. Otherwise, *-1* shall be returned and *errno* set to indicate the error.

29957 **ERRORS**

29958 The *posix_openpt()* function shall fail if:

29959 [EMFILE] All file descriptors available to the process are currently open.
 29960 [ENFILE] The maximum allowable number of files is currently open in the system.

29961 The *posix_openpt()* function may fail if:

29962 [EINVAL] The value of *oflag* is not valid.
 29963 [EAGAIN] Out of pseudo-terminal resources.

29964 OB XSR [ENOSR] Out of STREAMS resources.

29965 **EXAMPLES**

29966 **Opening a Pseudo-Terminal and Returning the Name of the Slave Device and a File**
 29967 **Descriptor**

```
29968 #include <fcntl.h>
29969 #include <stdio.h>
29970 int masterfd, slavefd;
29971 char *slavedevice;
29972 masterfd = posix_openpt(O_RDWR|O_NOCTTY);
29973 if (masterfd == -1
```

```

29974     || grantpt (masterfd) == -1
29975     || unlockpt (masterfd) == -1
29976     || (slavedevice = ptsname (masterfd)) == NULL)
29977     return -1;

29978     printf("slave device is: %s\n", slavedevice);

29979     slavefd = open(slavedevice, O_RDWR|O_NOCTTY);
29980     if (slavefd < 0)
29981         return -1;

```

APPLICATION USAGE

This function is a method for portably obtaining a file descriptor of a master terminal device for a pseudo-terminal. The *grantpt()* and *ptsname()* functions can be used to manipulate mode and ownership permissions, and to obtain the name of the slave device, respectively.

RATIONALE

The standard developers considered the matter of adding a special device for cloning master pseudo-terminals: the */dev/ptmx* device. However, consensus could not be reached, and it was felt that adding a new function would permit other implementations. The *posix_openpt()* function is designed to complement the *grantpt()*, *ptsname()*, and *unlockpt()* functions.

On implementations supporting the */dev/ptmx* clone device, opening the master device of a pseudo-terminal is simply:

```

29993     mfdp = open("/dev/ptmx", oflag );
29994     if (mfdp < 0)
29995         return -1;

```

FUTURE DIRECTIONS

None.

SEE ALSO

grantpt(), *open()*, *ptsname()*, *unlockpt()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**

CHANGE HISTORY

First released in Issue 6.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

SD5-XSH-ERN-51 is applied, correcting an error in the EXAMPLES section.

30006 **NAME**30007 posix_spawn, posix_spawnnp — spawn a process (**ADVANCED REALTIME**)30008 **SYNOPSIS**

```

30009 SPN    #include <spawn.h>
30010
30011 int posix_spawn(pid_t *restrict pid, const char *restrict path,
30012               const posix_spawn_file_actions_t *file_actions,
30013               const posix_spawnattr_t *restrict attrp,
30014               char *const argv[restrict], char *const envp[restrict]);
30015 int posix_spawnnp(pid_t *restrict pid, const char *restrict file,
30016                 const posix_spawn_file_actions_t *file_actions,
30017                 const posix_spawnattr_t *restrict attrp,
30018                 char *const argv[restrict], char * const envp[restrict]);

```

30018 **DESCRIPTION**

30019 The *posix_spawn()* and *posix_spawnnp()* functions shall create a new process (child process) from
 30020 the specified process image. The new process image shall be constructed from a regular
 30021 executable file called the new process image file.

30022 When a C program is executed as the result of this call, it shall be entered as a C-language
 30023 function call as follows:

```

30024 int main(int argc, char *argv[]);

```

30025 where *argc* is the argument count and *argv* is an array of character pointers to the arguments
 30026 themselves. In addition, the following variable:

```

30027 extern char **environ;

```

30028 shall be initialized as a pointer to an array of character pointers to the environment strings.

30029 The argument *argv* is an array of character pointers to null-terminated strings. The last member
 30030 of this array shall be a null pointer and is not counted in *argc*. These strings constitute the
 30031 argument list available to the new process image. The value in *argv*[0] should point to a filename
 30032 that is associated with the process image being started by the *posix_spawn()* or *posix_spawnnp()*
 30033 function.

30034 The argument *envp* is an array of character pointers to null-terminated strings. These strings
 30035 constitute the environment for the new process image. The environment array is terminated by a
 30036 null pointer.

30037 The number of bytes available for the combined argument and environment lists of the child
 30038 process is {ARG_MAX}. The implementation shall specify in the system documentation (see the
 30039 Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance) whether any list
 30040 overhead, such as length words, null terminators, pointers, or alignment bytes, is included in
 30041 this total.

30042 The *path* argument to *posix_spawn()* is a pathname that identifies the new process image file to
 30043 execute.

30044 The *file* parameter to *posix_spawnnp()* shall be used to construct a pathname that identifies the
 30045 new process image file. If the *file* parameter contains a slash character, the *file* parameter shall be
 30046 used as the pathname for the new process image file. Otherwise, the path prefix for this file shall
 30047 be obtained by a search of the directories passed as the environment variable *PATH* (see the Base
 30048 Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables). If this
 30049 environment variable is not defined, the results of the search are implementation-defined.

30050 If *file_actions* is a null pointer, then file descriptors open in the calling process shall remain open
 30051 in the child process, except for those whose close-on-exec flag FD_CLOEXEC is set (see *fcntl()*).
 30052 For those file descriptors that remain open, all attributes of the corresponding open file
 30053 descriptions, including file locks (see *fcntl()*), shall remain unchanged.

30054 If *file_actions* is not NULL, then the file descriptors open in the child process shall be those open
 30055 in the calling process as modified by the spawn file actions object pointed to by *file_actions* and
 30056 the FD_CLOEXEC flag of each remaining open file descriptor after the spawn file actions have
 30057 been processed. The effective order of processing the spawn file actions shall be:

- 30058 1. The set of open file descriptors for the child process shall initially be the same set as is
 30059 open for the calling process. All attributes of the corresponding open file descriptions,
 30060 including file locks (see *fcntl()*), shall remain unchanged.
- 30061 2. The signal mask, signal default actions, and the effective user and group IDs for the child
 30062 process shall be changed as specified in the attributes object referenced by *attrp*.
- 30063 3. The file actions specified by the spawn file actions object shall be performed in the order
 30064 in which they were added to the spawn file actions object.
- 30065 4. Any file descriptor that has its FD_CLOEXEC flag set (see *fcntl()*) shall be closed.

30066 The **posix_spawnattr_t** spawn attributes object type is defined in **<spawn.h>**. It shall contain at
 30067 least the attributes defined below.

30068 If the POSIX_SPAWN_SETPGROUP flag is set in the *spawn-flags* attribute of the object referenced
 30069 by *attrp*, and the *spawn-pgroup* attribute of the same object is non-zero, then the child's process
 30070 group shall be as specified in the *spawn-pgroup* attribute of the object referenced by *attrp*.

30071 As a special case, if the POSIX_SPAWN_SETPGROUP flag is set in the *spawn-flags* attribute of
 30072 the object referenced by *attrp*, and the *spawn-pgroup* attribute of the same object is set to zero,
 30073 then the child shall be in a new process group with a process group ID equal to its process ID.

30074 If the POSIX_SPAWN_SETPGROUP flag is not set in the *spawn-flags* attribute of the object
 30075 referenced by *attrp*, the new child process shall inherit the parent's process group.

30076 PS If the POSIX_SPAWN_SETSCHEDPARAM flag is set in the *spawn-flags* attribute of the object
 30077 referenced by *attrp*, but POSIX_SPAWN_SETSCHEDULER is not set, the new process image
 30078 shall initially have the scheduling policy of the calling process with the scheduling parameters
 30079 specified in the *spawn-schedparam* attribute of the object referenced by *attrp*.

30080 If the POSIX_SPAWN_SETSCHEDULER flag is set in the *spawn-flags* attribute of the object
 30081 referenced by *attrp* (regardless of the setting of the POSIX_SPAWN_SETSCHEDPARAM flag),
 30082 the new process image shall initially have the scheduling policy specified in the *spawn-*
 30083 *schedpolicy* attribute of the object referenced by *attrp* and the scheduling parameters specified in
 30084 the *spawn-schedparam* attribute of the same object.

30085 The POSIX_SPAWN_RESETPID flag in the *spawn-flags* attribute of the object referenced by *attrp*
 30086 governs the effective user ID of the child process. If this flag is not set, the child process shall
 30087 inherit the effective user ID of the parent process. If this flag is set, the effective user ID of the
 30088 child process shall be reset to the parent's real user ID. In either case, if the set-user-ID mode bit
 30089 of the new process image file is set, the effective user ID of the child process shall become that
 30090 file's owner ID before the new process image begins execution.

30091 The POSIX_SPAWN_RESETPID flag in the *spawn-flags* attribute of the object referenced by *attrp*
 30092 also governs the effective group ID of the child process. If this flag is not set, the child process
 30093 shall inherit the effective group ID of the parent process. If this flag is set, the effective group ID
 30094 of the child process shall be reset to the parent's real group ID. In either case, if the set-group-ID
 30095 mode bit of the new process image file is set, the effective group ID of the child process shall
 30096 become that file's group ID before the new process image begins execution.

30097 If the POSIX_SPAWN_SETSIGMASK flag is set in the *spawn_flags* attribute of the object
 30098 referenced by *attrp*, the child process shall initially have the signal mask specified in the *spawn-*
 30099 *sigmask* attribute of the object referenced by *attrp*.

30100 If the POSIX_SPAWN_SETSIGDEF flag is set in the *spawn_flags* attribute of the object referenced
 30101 by *attrp*, the signals specified in the *spawn-sigdefault* attribute of the same object shall be set to
 30102 their default actions in the child process. Signals set to the default action in the parent process
 30103 shall be set to the default action in the child process.

30104 Signals set to be caught by the calling process shall be set to the default action in the child
 30105 process.

30106 Except for SIGCHLD, signals set to be ignored by the calling process image shall be set to be
 30107 ignored by the child process, unless otherwise specified by the POSIX_SPAWN_SETSIGDEF flag
 30108 being set in the *spawn_flags* attribute of the object referenced by *attrp* and the signals being
 30109 indicated in the *spawn-sigdefault* attribute of the object referenced by *attrp*.

30110 If the SIGCHLD signal is set to be ignored by the calling process, it is unspecified whether the
 30111 SIGCHLD signal is set to be ignored or to the default action in the child process, unless
 30112 otherwise specified by the POSIX_SPAWN_SETSIGDEF flag being set in the *spawn_flags*
 30113 attribute of the object referenced by *attrp* and the SIGCHLD signal being indicated in the
 30114 *spawn-sigdefault* attribute of the object referenced by *attrp*.

30115 If the value of the *attrp* pointer is NULL, then the default values are used.

30116 All process attributes, other than those influenced by the attributes set in the object referenced
 30117 by *attrp* as specified above or by the file descriptor manipulations specified in *file_actions*, shall
 30118 appear in the new process image as though *fork()* had been called to create a child process and
 30119 then a member of the *exec* family of functions had been called by the child process to execute the
 30120 new process image.

30121 It is implementation-defined whether the fork handlers are run when *posix_spawn()* or
 30122 *posix_spawnnp()* is called.

30123 RETURN VALUE

30124 Upon successful completion, *posix_spawn()* and *posix_spawnnp()* shall return the process ID of the
 30125 child process to the parent process, in the variable pointed to by a non-NULL *pid* argument, and
 30126 shall return zero as the function return value. Otherwise, no child process shall be created, the
 30127 value stored into the variable pointed to by a non-NULL *pid* is unspecified, and an error number
 30128 shall be returned as the function return value to indicate the error. If the *pid* argument is a null
 30129 pointer, the process ID of the child is not returned to the caller.

30130 ERRORS

30131 The *posix_spawn()* and *posix_spawnnp()* functions may fail if:

30132 [EINVAL] The value specified by *file_actions* or *attrp* is invalid.

30133 If this error occurs after the calling process successfully returns from the *posix_spawn()* or
 30134 *posix_spawnnp()* function, the child process may exit with exit status 127.

30135 If *posix_spawn()* or *posix_spawnnp()* fail for any of the reasons that would cause *fork()* or one of
 30136 the *exec* family of functions to fail, an error value shall be returned as described by *fork()* and
 30137 *exec*, respectively (or, if the error occurs after the calling process successfully returns, the child
 30138 process shall exit with exit status 127).

30139 If POSIX_SPAWN_SETPGROUP is set in the *spawn_flags* attribute of the object referenced by
 30140 *attrp*, and *posix_spawn()* or *posix_spawnnp()* fails while changing the child's process group, an
 30141 error value shall be returned as described by *setpgid()* (or, if the error occurs after the calling
 30142 process successfully returns, the child process shall exit with exit status 127).

30143 PS If `POSIX_SPAWN_SETSCHEDPARAM` is set and `POSIX_SPAWN_SETSCHEDULER` is not set in
 30144 the *spawn-flags* attribute of the object referenced by *attrp*, then if *posix_spawn()* or *posix_spawnp()*
 30145 fails for any of the reasons that would cause *sched_setparam()* to fail, an error value shall be
 30146 returned as described by *sched_setparam()* (or, if the error occurs after the calling process
 30147 successfully returns, the child process shall exit with exit status 127).

30148 If `POSIX_SPAWN_SETSCHEDULER` is set in the *spawn-flags* attribute of the object referenced by
 30149 *attrp*, and if *posix_spawn()* or *posix_spawnp()* fails for any of the reasons that would cause
 30150 *sched_setscheduler()* to fail, an error value shall be returned as described by *sched_setscheduler()*
 30151 (or, if the error occurs after the calling process successfully returns, the child process shall exit
 30152 with exit status 127).

30153 If the *file_actions* argument is not `NULL`, and specifies any *close*, *dup2*, or *open* actions to be
 30154 performed, and if *posix_spawn()* or *posix_spawnp()* fails for any of the reasons that would cause
 30155 *close()*, *dup2()*, or *open()* to fail, an error value shall be returned as described by *close()*,
 30156 *dup2()*, and *open()*, respectively (or, if the error occurs after the calling process successfully returns, the
 30157 child process shall exit with exit status 127). An open file action may, by itself, result in any of
 30158 the errors described by *close()* or *dup2()*, in addition to those described by *open()*.

EXAMPLES

30159 None.
 30160

APPLICATION USAGE

30161 These functions are part of the Spawn option and need not be provided on all implementations.
 30162

RATIONALE

30163 The *posix_spawn()* function and its close relation *posix_spawnp()* have been introduced to
 30164 overcome the following perceived difficulties with *fork()*: the *fork()* function is difficult or
 30165 impossible to implement without swapping or dynamic address translation.
 30166

- 30167 • Swapping is generally too slow for a realtime environment.
- 30168 • Dynamic address translation is not available everywhere that POSIX might be useful.
- 30169 • Processes are too useful to simply option out of POSIX whenever it must run without
 30170 address translation or other MMU services.

30171 Thus, POSIX needs process creation and file execution primitives that can be efficiently
 30172 implemented without address translation or other MMU services.

30173 The *posix_spawn()* function is implementable as a library routine, but both *posix_spawn()* and
 30174 *posix_spawnp()* are designed as kernel operations. Also, although they may be an efficient
 30175 replacement for many *fork()/exec* pairs, their goal is to provide useful process creation
 30176 primitives for systems that have difficulty with *fork()*, not to provide drop-in replacements for
 30177 *fork()/exec*.

30178 This view of the role of *posix_spawn()* and *posix_spawnp()* influenced the design of their API. It
 30179 does not attempt to provide the full functionality of *fork()/exec* in which arbitrary user-specified
 30180 operations of any sort are permitted between the creation of the child process and the execution
 30181 of the new process image; any attempt to reach that level would need to provide a programming
 30182 language as parameters. Instead, *posix_spawn()* and *posix_spawnp()* are process creation
 30183 primitives like the *Start_Process* and *Start_Process_Search* Ada language bindings package
 30184 *POSIX_Process_Primitives* and also like those in many operating systems that are not UNIX
 30185 systems, but with some POSIX-specific additions.

30186 To achieve its coverage goals, *posix_spawn()* and *posix_spawnp()* have control of six types of
 30187 inheritance: file descriptors, process group ID, user and group ID, signal mask, scheduling, and
 30188 whether each signal ignored in the parent will remain ignored in the child, or be reset to its
 30189 default action in the child.

30190 Control of file descriptors is required to allow an independently written child process image to

30191 access data streams opened by and even generated or read by the parent process without being
 30192 specifically coded to know which parent files and file descriptors are to be used. Control of the
 30193 process group ID is required to control how the job control of the child process relates to that of
 30194 the parent.

30195 Control of the signal mask and signal defaulting is sufficient to support the implementation of
 30196 *system()*. Although support for *system()* is not explicitly one of the goals for *posix_spawn()* and
 30197 *posix_spawnnp()*, it is covered under the “at least 50%” coverage goal.

30198 The intention is that the normal file descriptor inheritance across *fork()*, the subsequent effect of
 30199 the specified spawn file actions, and the normal file descriptor inheritance across one of the *exec*
 30200 family of functions should fully specify open file inheritance. The implementation need make no
 30201 decisions regarding the set of open file descriptors when the child process image begins
 30202 execution, those decisions having already been made by the caller and expressed as the set of
 30203 open file descriptors and their FD_CLOEXEC flags at the time of the call and the spawn file
 30204 actions object specified in the call. We have been assured that in cases where the POSIX
 30205 *Start_Process* Ada primitives have been implemented in a library, this method of controlling file
 30206 descriptor inheritance may be implemented very easily.

30207 We can identify several problems with *posix_spawn()* and *posix_spawnnp()*, but there does not
 30208 appear to be a solution that introduces fewer problems. Environment modification for child
 30209 process attributes not specifiable via the *attrp* or *file_actions* arguments must be done in the
 30210 parent process, and since the parent generally wants to save its context, it is more costly than
 30211 similar functionality with *fork()/exec*. It is also complicated to modify the environment of a
 30212 multi-threaded process temporarily, since all threads must agree when it is safe for the
 30213 environment to be changed. However, this cost is only borne by those invocations of
 30214 *posix_spawn()* and *posix_spawnnp()* that use the additional functionality. Since extensive
 30215 modifications are not the usual case, and are particularly unlikely in time-critical code, keeping
 30216 much of the environment control out of *posix_spawn()* and *posix_spawnnp()* is appropriate design.

30217 The *posix_spawn()* and *posix_spawnnp()* functions do not have all the power of *fork()/exec*. This is
 30218 to be expected. The *fork()* function is a wonderfully powerful operation. We do not expect to
 30219 duplicate its functionality in a simple, fast function with no special hardware requirements. It is
 30220 worth noting that *posix_spawn()* and *posix_spawnnp()* are very similar to the process creation
 30221 operations on many operating systems that are not UNIX systems.

30222 Requirements

30223 The requirements for *posix_spawn()* and *posix_spawnnp()* are:

- 30224 • They must be implementable without an MMU or unusual hardware.
- 30225 • They must be compatible with existing POSIX standards.

30226 Additional goals are:

- 30227 • They should be efficiently implementable.
- 30228 • They should be able to replace at least 50% of typical executions of *fork()*.
- 30229 • A system with *posix_spawn()* and *posix_spawnnp()* and without *fork()* should be useful, at
 30230 least for realtime applications.
- 30231 • A system with *fork()* and the *exec* family should be able to implement *posix_spawn()* and
 30232 *posix_spawnnp()* as library routines.

30233

Two-Syntax30234
30235
30236
30237

POSIX *exec* has several calling sequences with approximately the same functionality. These appear to be required for compatibility with existing practice. Since the existing practice for the *posix_spawn**() functions is otherwise substantially unlike POSIX, we feel that simplicity outweighs compatibility. There are, therefore, only two names for the *posix_spawn**() functions.

30238
30239

The parameter list does not differ between *posix_spawn*() and *posix_spawnp*(); *posix_spawnp*() interprets the second parameter more elaborately than *posix_spawn*().

30240

Compatibility with POSIX.5 (Ada)30241
30242
30243
30244
30245
30246
30247
30248
30249
30250
30251
30252
30253
30254
30255

The *Start_Process* and *Start_Process_Search* procedures from the *POSIX_Process_Primitives* package from the Ada language binding to POSIX.1 encapsulate *fork*() and *exec* functionality in a manner similar to that of *posix_spawn*() and *posix_spawnp*(). Originally, in keeping with our simplicity goal, the standard developers had limited the capabilities of *posix_spawn*() and *posix_spawnp*() to a subset of the capabilities of *Start_Process* and *Start_Process_Search*; certain non-default capabilities were not supported. However, based on suggestions by the ballot group to improve file descriptor mapping or drop it, and on the advice of an Ada Language Bindings working group member, the standard developers decided that *posix_spawn*() and *posix_spawnp*() should be sufficiently powerful to implement *Start_Process* and *Start_Process_Search*. The rationale is that if the Ada language binding to such a primitive had already been approved as an IEEE standard, there can be little justification for not approving the functionally-equivalent parts of a C binding. The only three capabilities provided by *posix_spawn*() and *posix_spawnp*() that are not provided by *Start_Process* and *Start_Process_Search* are optionally specifying the child's process group ID, the set of signals to be reset to default signal handling in the child process, and the child's scheduling policy and parameters.

30256
30257
30258
30259
30260
30261

For the Ada language binding for *Start_Process* to be implemented with *posix_spawn*(), that binding would need to explicitly pass an empty signal mask and the parent's environment to *posix_spawn*() whenever the caller of *Start_Process* allowed these arguments to default, since *posix_spawn*() does not provide such defaults. The ability of *Start_Process* to mask user-specified signals during its execution is functionally unique to the Ada language binding and must be dealt with in the binding separately from the call to *posix_spawn*().

30262

Process Group30263
30264
30265

The process group inheritance field can be used to join the child process with an existing process group. By assigning a value of zero to the *spawn-pgroup* attribute of the object referenced by *attrp*, the *setpgid*() mechanism will place the child process in a new process group.

30266

Threads30267
30268
30269
30270
30271
30272
30273
30274
30275

Without the *posix_spawn*() and *posix_spawnp*() functions, systems without address translation can still use threads to give an abstraction of concurrency. In many cases, thread creation suffices, but it is not always a good substitute. The *posix_spawn*() and *posix_spawnp*() functions are considerably "heavier" than thread creation. Processes have several important attributes that threads do not. Even without address translation, a process may have base-and-bound memory protection. Each process has a process environment including security attributes and file capabilities, and powerful scheduling attributes. Processes abstract the behavior of non-uniform-memory-architecture multi-processors better than threads, and they are more convenient to use for activities that are not closely linked.

30276
30277
30278
30279
30280

The *posix_spawn*() and *posix_spawnp*() functions may not bring support for multiple processes to every configuration. Process creation is not the only piece of operating system support required to support multiple processes. The total cost of support for multiple processes may be quite high in some circumstances. Existing practice shows that support for multiple processes is uncommon and threads are common among "tiny kernels". There should, therefore, probably

30281 continue to be AEPs for operating systems with only one process.

30282 Asynchronous Error Notification

30283 A library implementation of *posix_spawn()* or *posix_spawnp()* may not be able to detect all
30284 possible errors before it forks the child process. IEEE Std 1003.1-200x provides for an error
30285 indication returned from a child process which could not successfully complete the spawn
30286 operation via a special exit status which may be detected using the status value returned by
30287 *wait()* and *waitpid()*.

30288 The *stat_val* interface and the macros used to interpret it are not well suited to the purpose of
30289 returning API errors, but they are the only path available to a library implementation. Thus, an
30290 implementation may cause the child process to exit with exit status 127 for any error detected
30291 during the spawn process after the *posix_spawn()* or *posix_spawnp()* function has successfully
30292 returned.

30293 The standard developers had proposed using two additional macros to interpret *stat_val*. The
30294 first, WIFSPAWNFAIL, would have detected a status that indicated that the child exited because
30295 of an error detected during the *posix_spawn()* or *posix_spawnp()* operations rather than during
30296 actual execution of the child process image; the second, WSPAWNERRNO, would have
30297 extracted the error value if WIFSPAWNFAIL indicated a failure. Unfortunately, the ballot group
30298 strongly opposed this because it would make a library implementation of *posix_spawn()* or
30299 *posix_spawnp()* dependent on kernel modifications to *waitpid()* to be able to embed special
30300 information in *stat_val* to indicate a spawn failure.

30301 The 8 bits of child process exit status that are guaranteed by IEEE Std 1003.1-200x to be
30302 accessible to the waiting parent process are insufficient to disambiguate a spawn error from any
30303 other kind of error that may be returned by an arbitrary process image. No other bits of the exit
30304 status are required to be visible in *stat_val*, so these macros could not be strictly implemented at
30305 the library level. Reserving an exit status of 127 for such spawn errors is consistent with the use
30306 of this value by *system()* and *popen()* to signal failures in these operations that occur after the
30307 function has returned but before a shell is able to execute. The exit status of 127 does not
30308 uniquely identify this class of error, nor does it provide any detailed information on the nature
30309 of the failure. Note that a kernel implementation of *posix_spawn()* or *posix_spawnp()* is permitted
30310 (and encouraged) to return any possible error as the function value, thus providing more
30311 detailed failure information to the parent process.

30312 Thus, no special macros are available to isolate asynchronous *posix_spawn()* or *posix_spawnp()*
30313 errors. Instead, errors detected by the *posix_spawn()* or *posix_spawnp()* operations in the context
30314 of the child process before the new process image executes are reported by setting the child's exit
30315 status to 127. The calling process may use the WIFEXITED and WEXITSTATUS macros on the
30316 *stat_val* stored by the *wait()* or *waitpid()* functions to detect spawn failures to the extent that
30317 other status values with which the child process image may exit (before the parent can
30318 conclusively determine that the child process image has begun execution) are distinct from exit
30319 status 127.

30320 FUTURE DIRECTIONS

30321 None.

30322 SEE ALSO

30323 *alarm()*, *chmod()*, *close()*, *dup()*, *exec*, *exit()*, *fcntl()*, *fork()*, *fstatat()*, *kill()*, *open()*,
30324 *posix_spawn_file_actions_addclose()*, *posix_spawn_file_actions_adddup2()*,
30325 *posix_spawn_file_actions_addopen()*, *posix_spawn_file_actions_destroy()*, *posix_spawnattr_destroy()*,
30326 *posix_spawnattr_init()*, *posix_spawnattr_getsigdefault()*, *posix_spawnattr_getflags()*,
30327 *posix_spawnattr_getpgroup()*, *posix_spawnattr_getschedparam()*, *posix_spawnattr_getschedpolicy()*,
30328 *posix_spawnattr_getsigmask()*, *posix_spawnattr_setsigdefault()*, *posix_spawnattr_setflags()*,
30329 *posix_spawnattr_setpgroup()*, *posix_spawnattr_setschedparam()*, *posix_spawnattr_setschedpolicy()*,
30330 *posix_spawnattr_setsigmask()*, *sched_setparam()*, *sched_setscheduler()*, *setpgid()*, *setuid()*, *times()*,

30331 *wait()*, the Base Definitions volume of IEEE Std 1003.1-200x, <spawn.h>

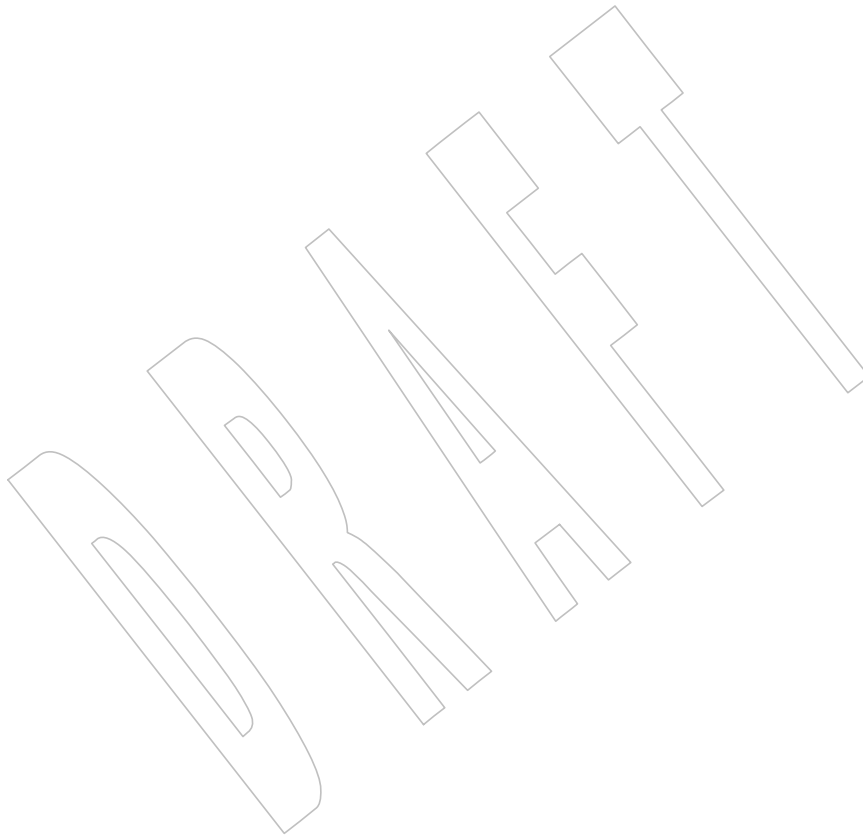
30332 **CHANGE HISTORY**

30333 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

30334 IEEE PASC Interpretation 1003.1 #103 is applied, noting that the signal default actions are
30335 changed as well as the signal mask in step 2.

30336 IEEE PASC Interpretation 1003.1 #132 is applied.

30337 Functionality relating to the Threads option is moved to the Base.



30338 **NAME**

30339 `posix_spawn_file_actions_addclose`, `posix_spawn_file_actions_addopen` — add close or open
 30340 action to spawn file actions object (**ADVANCED REALTIME**)

30341 **SYNOPSIS**

```
30342 SPN      #include <spawn.h>
30343
30344      int posix_spawn_file_actions_addclose(posix_spawn_file_actions_t *
30345      file_actions, int fildes);
30346      int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *
30347      restrict file_actions, int fildes,
30348      const char *restrict path, int oflag, mode_t mode);
```

30348 **DESCRIPTION**

30349 These functions shall add or delete a close or open action to a spawn file actions object.

30350 A spawn file actions object is of type **posix_spawn_file_actions_t** (defined in `<spawn.h>`) and is
 30351 used to specify a series of actions to be performed by a `posix_spawn()` or `posix_spawnp()`
 30352 operation in order to arrive at the set of open file descriptors for the child process given the set
 30353 of open file descriptors of the parent. IEEE Std 1003.1-200x does not define comparison or
 30354 assignment operators for the type **posix_spawn_file_actions_t**.

30355 A spawn file actions object, when passed to `posix_spawn()` or `posix_spawnp()`, shall specify how
 30356 the set of open file descriptors in the calling process is transformed into a set of potentially open
 30357 file descriptors for the spawned process. This transformation shall be as if the specified sequence
 30358 of actions was performed exactly once, in the context of the spawned process (prior to execution
 30359 of the new process image), in the order in which the actions were added to the object;
 30360 additionally, when the new process image is executed, any file descriptor (from this new set)
 30361 which has its `FD_CLOEXEC` flag set shall be closed (see `posix_spawn()`).

30362 The `posix_spawn_file_actions_addclose()` function shall add a *close* action to the object referenced
 30363 by `file_actions` that shall cause the file descriptor `fildes` to be closed (as if `close(fildes)` had been
 30364 called) when a new process is spawned using this file actions object.

30365 The `posix_spawn_file_actions_addopen()` function shall add an *open* action to the object referenced
 30366 by `file_actions` that shall cause the file named by `path` to be opened (as if `open(path, oflag, mode)`
 30367 had been called, and the returned file descriptor, if not `fildes`, had been changed to `fildes`) when a
 30368 new process is spawned using this file actions object. If `fildes` was already an open file descriptor,
 30369 it shall be closed before the new file is opened.

30370 The string described by `path` shall be copied by the `posix_spawn_file_actions_addopen()` function.

30371 **RETURN VALUE**

30372 Upon successful completion, these functions shall return zero; otherwise, an error number shall
 30373 be returned to indicate the error.

30374 **ERRORS**

30375 These functions shall fail if:

30376 [EBADF] The value specified by `fildes` is negative or greater than or equal to
 30377 {OPEN_MAX}.

30378 These functions may fail if:

30379 [EINVAL] The value specified by `file_actions` is invalid.

30380 [ENOMEM] Insufficient memory exists to add to the spawn file actions object.

30381 It shall not be considered an error for the *filides* argument passed to these functions to specify a
 30382 file descriptor for which the specified operation could not be performed at the time of the call.
 30383 Any such error will be detected when the associated file actions object is later used during a
 30384 *posix_spawn()* or *posix_spawnnp()* operation.

30385 EXAMPLES

30386 None.

30387 APPLICATION USAGE

30388 These functions are part of the Spawn option and need not be provided on all implementations.

30389 RATIONALE

30390 A spawn file actions object may be initialized to contain an ordered sequence of *close()*, *dup2()*,
 30391 and *open()* operations to be used by *posix_spawn()* or *posix_spawnnp()* to arrive at the set of open
 30392 file descriptors inherited by the spawned process from the set of open file descriptors in the
 30393 parent at the time of the *posix_spawn()* or *posix_spawnnp()* call. It had been suggested that the
 30394 *close()* and *dup2()* operations alone are sufficient to rearrange file descriptors, and that files
 30395 which need to be opened for use by the spawned process can be handled either by having the
 30396 calling process open them before the *posix_spawn()* or *posix_spawnnp()* call (and close them after),
 30397 or by passing filenames to the spawned process (in *argv*) so that it may open them itself. The
 30398 standard developers recommend that applications use one of these two methods when practical,
 30399 since detailed error status on a failed open operation is always available to the application this
 30400 way. However, the standard developers feel that allowing a spawn file actions object to specify
 30401 open operations is still appropriate because:

- 30402 1. It is consistent with equivalent POSIX.5 (Ada) functionality.
- 30403 2. It supports the I/O redirection paradigm commonly employed by POSIX programs
 30404 designed to be invoked from a shell. When such a program is the child process, it may not
 30405 be designed to open files on its own.
- 30406 3. It allows file opens that might otherwise fail or violate file ownership/access rights if
 30407 executed by the parent process.

30408 Regarding 2. above, note that the spawn open file action provides to *posix_spawn()* and
 30409 *posix_spawnnp()* the same capability that the shell redirection operators provide to *system()*, only
 30410 without the intervening execution of a shell; for example:

```
30411 system ("myprog <file1 3<file2");
```

30412 Regarding 3. above, note that if the calling process needs to open one or more files for access by
 30413 the spawned process, but has insufficient spare file descriptors, then the open action is necessary
 30414 to allow the *open()* to occur in the context of the child process after other file descriptors have
 30415 been closed (that must remain open in the parent).

30416 Additionally, if a parent is executed from a file having a "set-user-id" mode bit set and the
 30417 POSIX_SPAWN_RESETIDS flag is set in the spawn attributes, a file created within the parent
 30418 process will (possibly incorrectly) have the parent's effective user ID as its owner, whereas a file
 30419 created via an *open()* action during *posix_spawn()* or *posix_spawnnp()* will have the parent's real
 30420 ID as its owner; and an open by the parent process may successfully open a file to which the real
 30421 user should not have access or fail to open a file to which the real user should have access.

30422

File Descriptor Mapping

30423

30424

30425

30426

30427

30428

30429

The standard developers had originally proposed using an array which specified the mapping of child file descriptors back to those of the parent. It was pointed out by the ballot group that it is not possible to reshuffle file descriptors arbitrarily in a library implementation of *posix_spawn()* or *posix_spawnnp()* without provision for one or more spare file descriptor entries (which simply may not be available). Such an array requires that an implementation develop a complex strategy to achieve the desired mapping without inadvertently closing the wrong file descriptor at the wrong time.

30430

30431

30432

30433

30434

30435

30436

It was noted by a member of the Ada Language Bindings working group that the approved Ada Language *Start_Process* family of POSIX process primitives use a caller-specified set of file actions to alter the normal *fork()/exec* semantics for inheritance of file descriptors in a very flexible way, yet no such problems exist because the burden of determining how to achieve the final file descriptor mapping is completely on the application. Furthermore, although the file actions interface appears frightening at first glance, it is actually quite simple to implement in either a library or the kernel.

30437

FUTURE DIRECTIONS

30438

None.

30439

SEE ALSO

30440

30441

30442

close(), *dup()*, *open()*, *posix_spawn()*, *posix_spawn_file_actions_adddup2()*, *posix_spawn_file_actions_destroy()*, *posix_spawnnp()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**spawn.h**>

30443

CHANGE HISTORY

30444

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

30445

30446

IEEE PASC Interpretation 1003.1 #105 is applied, adding a note to the DESCRIPTION that the string pointed to by *path* is copied by the *posix_spawn_file_actions_addopen()* function.

30447 **NAME**

30448 `posix_spawn_file_actions_adddup2` — add `dup2` action to spawn file actions object
 30449 (**ADVANCED REALTIME**)

30450 **SYNOPSIS**

```
30451 SPN #include <spawn.h>
30452
30452 int posix_spawn_file_actions_adddup2(posix_spawn_file_actions_t *
30453     file_actions, int fildes, int newfildes);
```

30454 **DESCRIPTION**

30455 The `posix_spawn_file_actions_adddup2()` function shall add a `dup2()` action to the object
 30456 referenced by `file_actions` that shall cause the file descriptor `fildes` to be duplicated as `newfildes` (as
 30457 if `dup2(fildes, newfildes)` had been called) when a new process is spawned using this file actions
 30458 object.

30459 A spawn file actions object is as defined in [`posix_spawn_file_actions_addclose\(\)`](#).

30460 **RETURN VALUE**

30461 Upon successful completion, the `posix_spawn_file_actions_adddup2()` function shall return zero;
 30462 otherwise, an error number shall be returned to indicate the error.

30463 **ERRORS**

30464 The `posix_spawn_file_actions_adddup2()` function shall fail if:

30465 [EBADF] The value specified by `fildes` or `newfildes` is negative or greater than or equal to
 30466 {OPEN_MAX}.

30467 [ENOMEM] Insufficient memory exists to add to the spawn file actions object.

30468 The `posix_spawn_file_actions_adddup2()` function may fail if:

30469 [EINVAL] The value specified by `file_actions` is invalid.

30470 It shall not be considered an error for the `fildes` argument passed to the
 30471 `posix_spawn_file_actions_adddup2()` function to specify a file descriptor for which the specified
 30472 operation could not be performed at the time of the call. Any such error will be detected when
 30473 the associated file actions object is later used during a `posix_spawn()` or `posix_spawnnp()`
 30474 operation.

30475 **EXAMPLES**

30476 None.

30477 **APPLICATION USAGE**

30478 The `posix_spawn_file_actions_adddup2()` function is part of the Spawn option and need not be
 30479 provided on all implementations.

30480 **RATIONALE**

30481 Refer to the RATIONALE in [`posix_spawn_file_actions_addclose\(\)`](#).

30482 **FUTURE DIRECTIONS**

30483 None.

30484 **SEE ALSO**

30485 [`dup\(\)`](#), [`posix_spawn\(\)`](#), [`posix_spawn_file_actions_addclose\(\)`](#), [`posix_spawn_file_actions_destroy\(\)`](#),
 30486 [`posix_spawnnp\(\)`](#), the Base Definitions volume of IEEE Std 1003.1-200x, [<spawn.h>](#)

30487

CHANGE HISTORY

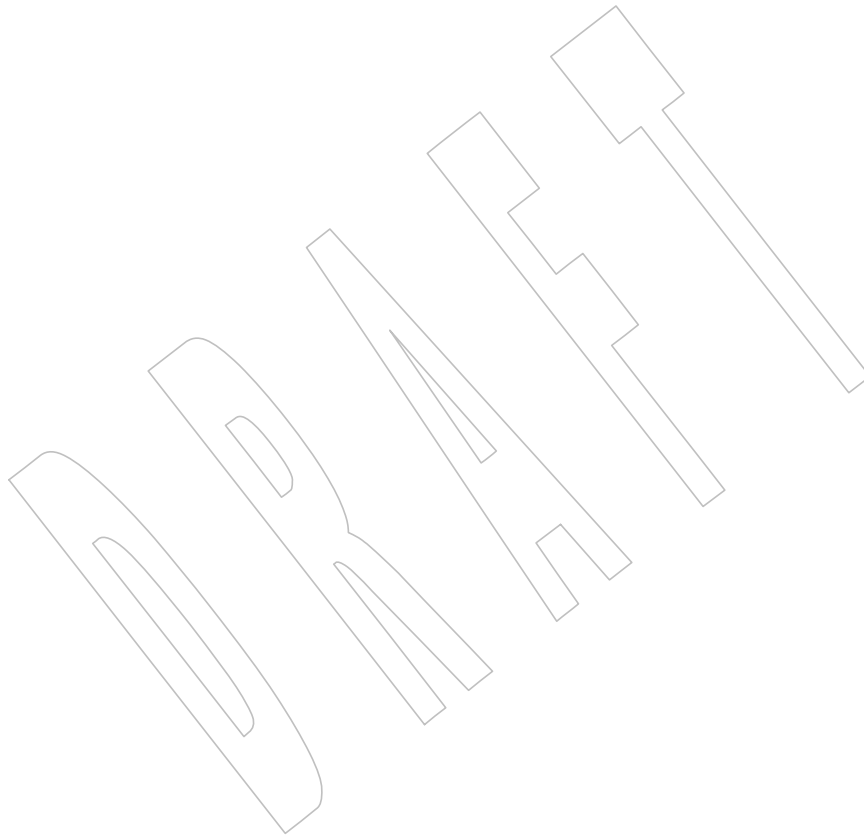
30488

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

30489

IEEE PASC Interpretation 1003.1 #104 is applied, noting that the [EBADF] error can apply to the *newfildes* argument in addition to *fildes*.

30490



posix_spawn_file_actions_addopen()*System Interfaces*30491 **NAME**

30492 `posix_spawn_file_actions_addopen` — add open action to spawn file actions object
30493 (ADVANCED REALTIME)

30494 **SYNOPSIS**

```
30495 SPN #include <spawn.h>  
30496 int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *  
30497     restrict file_actions, int fildes,  
30498     const char *restrict path, int oflag, mode_t mode);
```

30499 **DESCRIPTION**

30500 Refer to [posix_spawn_file_actions_addclose\(\)](#).

30501 **NAME**

30502 `posix_spawn_file_actions_destroy`, `posix_spawn_file_actions_init` — destroy and initialize
 30503 spawn file actions object (**ADVANCED REALTIME**)

30504 **SYNOPSIS**

```
30505 SPN #include <spawn.h>
30506
30507 int posix_spawn_file_actions_destroy(posix_spawn_file_actions_t *
30508     file_actions);
30509 int posix_spawn_file_actions_init(posix_spawn_file_actions_t *
30510     file_actions);
```

30510 **DESCRIPTION**

30511 The `posix_spawn_file_actions_destroy()` function shall destroy the object referenced by `file_actions`;
 30512 the object becomes, in effect, uninitialized. An implementation may cause
 30513 `posix_spawn_file_actions_destroy()` to set the object referenced by `file_actions` to an invalid value. A
 30514 destroyed spawn file actions object can be reinitialized using `posix_spawn_file_actions_init()`; the
 30515 results of otherwise referencing the object after it has been destroyed are undefined.

30516 The `posix_spawn_file_actions_init()` function shall initialize the object referenced by `file_actions` to
 30517 contain no file actions for `posix_spawn()` or `posix_spawnnp()` to perform.

30518 A spawn file actions object is as defined in [`posix_spawn_file_actions_addclose\(\)`](#).

30519 The effect of initializing an already initialized spawn file actions object is undefined.

30520 **RETURN VALUE**

30521 Upon successful completion, these functions shall return zero; otherwise, an error number shall
 30522 be returned to indicate the error.

30523 **ERRORS**

30524 The `posix_spawn_file_actions_init()` function shall fail if:

30525 [ENOMEM] Insufficient memory exists to initialize the spawn file actions object.

30526 The `posix_spawn_file_actions_destroy()` function may fail if:

30527 [EINVAL] The value specified by `file_actions` is invalid.

30528 **EXAMPLES**

30529 None.

30530 **APPLICATION USAGE**

30531 These functions are part of the Spawn option and need not be provided on all implementations.

30532 **RATIONALE**

30533 Refer to the RATIONALE in [`posix_spawn_file_actions_addclose\(\)`](#).

30534 **FUTURE DIRECTIONS**

30535 None.

30536 **SEE ALSO**

30537 [`posix_spawn\(\)`](#), [`posix_spawnnp\(\)`](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<spawn.h>`

30538 **CHANGE HISTORY**

30539 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

30540 In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.

30541 **NAME**

30542 `posix_spawnattr_destroy`, `posix_spawnattr_init` — destroy and initialize spawn attributes object
 30543 (ADVANCED REALTIME)

30544 **SYNOPSIS**

```
30545 SPN #include <spawn.h>
30546
30546 int posix_spawnattr_destroy(posix_spawnattr_t *attr);
30547 int posix_spawnattr_init(posix_spawnattr_t *attr);
```

30548 **DESCRIPTION**

30549 The `posix_spawnattr_destroy()` function shall destroy a spawn attributes object. A destroyed `attr`
 30550 attributes object can be reinitialized using `posix_spawnattr_init()`; the results of otherwise
 30551 referencing the object after it has been destroyed are undefined. An implementation may cause
 30552 `posix_spawnattr_destroy()` to set the object referenced by `attr` to an invalid value.

30553 The `posix_spawnattr_init()` function shall initialize a spawn attributes object `attr` with the default
 30554 value for all of the individual attributes used by the implementation. Results are undefined if
 30555 `posix_spawnattr_init()` is called specifying an already initialized `attr` attributes object.

30556 A spawn attributes object is of type **posix_spawnattr_t** (defined in **<spawn.h>**) and is used to
 30557 specify the inheritance of process attributes across a spawn operation. IEEE Std 1003.1-200x does
 30558 not define comparison or assignment operators for the type **posix_spawnattr_t**.

30559 Each implementation shall document the individual attributes it uses and their default values
 30560 unless these values are defined by IEEE Std 1003.1-200x. Attributes not defined by
 30561 IEEE Std 1003.1-200x, their default values, and the names of the associated functions to get and
 30562 set those attribute values are implementation-defined.

30563 The resulting spawn attributes object (possibly modified by setting individual attribute values),
 30564 is used to modify the behavior of `posix_spawn()` or `posix_spawnnp()`. After a spawn attributes
 30565 object has been used to spawn a process by a call to a `posix_spawn()` or `posix_spawnnp()`, any
 30566 function affecting the attributes object (including destruction) shall not affect any process that
 30567 has been spawned in this way.

30568 **RETURN VALUE**

30569 Upon successful completion, `posix_spawnattr_destroy()` and `posix_spawnattr_init()` shall return
 30570 zero; otherwise, an error number shall be returned to indicate the error.

30571 **ERRORS**

30572 The `posix_spawnattr_init()` function shall fail if:

30573 [ENOMEM] Insufficient memory exists to initialize the spawn attributes object.

30574 The `posix_spawnattr_destroy()` function may fail if:

30575 [EINVAL] The value specified by `attr` is invalid.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE

The original spawn interface proposed in IEEE Std 1003.1-200x defined the attributes that specify the inheritance of process attributes across a spawn operation as a structure. In order to be able to separate optional individual attributes under their appropriate options (that is, the *spawn-schedparam* and *spawn-schedpolicy* attributes depending upon the Process Scheduling option), and also for extensibility and consistency with the newer POSIX interfaces, the attributes interface has been changed to an opaque data type. This interface now consists of the type **posix_spawnattr_t**, representing a spawn attributes object, together with associated functions to initialize or destroy the attributes object, and to set or get each individual attribute. Although the new object-oriented interface is more verbose than the original structure, it is simple to use, more extensible, and easy to implement.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_spawn(), *posix_spawnattr_getsigdefault()*, *posix_spawnattr_getflags()*,
posix_spawnattr_getpgroup(), *posix_spawnattr_getschedparam()*, *posix_spawnattr_getschedpolicy()*,
posix_spawnattr_getsigmask(), *posix_spawnattr_setsigdefault()*, *posix_spawnattr_setflags()*,
posix_spawnattr_setpgroup(), *posix_spawnattr_setsigmask()*, *posix_spawnattr_setschedpolicy()*,
posix_spawnattr_setschedparam(), *posix_spawnnp()*, the Base Definitions volume of
 IEEE Std 1003.1-200x, <**spawn.h**>

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1 #106 is applied, noting that the effect of initializing an already initialized spawn attributes option is undefined.

30604 **NAME**

30605 `posix_spawnattr_getflags`, `posix_spawnattr_setflags` — get and set the spawn-flags attribute of a
 30606 spawn attributes object (**ADVANCED REALTIME**)

30607 **SYNOPSIS**

```
30608 SPN #include <spawn.h>
30609
30609 int posix_spawnattr_getflags(const posix_spawnattr_t *restrict attr,
30610 short *restrict flags);
30611 int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
```

30612 **DESCRIPTION**

30613 The `posix_spawnattr_getflags()` function shall obtain the value of the *spawn-flags* attribute from the
 30614 attributes object referenced by *attr*.

30615 The `posix_spawnattr_setflags()` function shall set the *spawn-flags* attribute in an initialized
 30616 attributes object referenced by *attr*.

30617 The *spawn-flags* attribute is used to indicate which process attributes are to be changed in the
 30618 new process image when invoking `posix_spawn()` or `posix_spawnnp()`. It is the bitwise-inclusive
 30619 OR of zero or more of the following flags:

```
30620 POSIX_SPAWN_RESETIDS
30621 POSIX_SPAWN_SETPGROUP
30622 POSIX_SPAWN_SETSIGDEF
30623 POSIX_SPAWN_SETSIGMASK
30624 PS POSIX_SPAWN_SETSCHEDPARAM
30625 POSIX_SPAWN_SETSCHEDULER
```

30626 These flags are defined in `<spawn.h>`. The default value of this attribute shall be as if no flags
 30627 were set.

30628 **RETURN VALUE**

30629 Upon successful completion, `posix_spawnattr_getflags()` shall return zero and store the value of
 30630 the *spawn-flags* attribute of *attr* into the object referenced by the *flags* parameter; otherwise, an
 30631 error number shall be returned to indicate the error.

30632 Upon successful completion, `posix_spawnattr_setflags()` shall return zero; otherwise, an error
 30633 number shall be returned to indicate the error.

30634 **ERRORS**

30635 These functions may fail if:

30636 [EINVAL] The value specified by *attr* is invalid.

30637 The `posix_spawnattr_setflags()` function may fail if:

30638 [EINVAL] The value of the attribute being set is not valid.

30639
30640
30641
30642
30643
30644
30645
30646
30647
30648
30649
30650
30651
30652
30653
30654

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_spawn(), *posix_spawnattr_destroy()*, *posix_spawnattr_init()*, *posix_spawnattr_getsigdefault()*,
posix_spawnattr_getpgroup(), *posix_spawnattr_getschedparam()*, *posix_spawnattr_getschedpolicy()*,
posix_spawnattr_getsigmask(), *posix_spawnattr_setsigdefault()*, *posix_spawnattr_setpgroup()*,
posix_spawnattr_setschedparam(), *posix_spawnattr_setschedpolicy()*, *posix_spawnattr_setsigmask()*,
posix_spawnnp(), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

DRAFT

posix_spawnattr_getpgroup()

System Interfaces

30655 **NAME**

30656 `posix_spawnattr_getpgroup`, `posix_spawnattr_setpgroup` — get and set the spawn-pgroup
 30657 attribute of a spawn attributes object (**ADVANCED REALTIME**)

30658 **SYNOPSIS**

```
30659 SPN #include <spawn.h>
30660
30661 int posix_spawnattr_getpgroup(const posix_spawnattr_t *restrict attr,
30662                               pid_t *restrict pgroup);
30662 int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
```

30663 **DESCRIPTION**

30664 The `posix_spawnattr_getpgroup()` function shall obtain the value of the `spawn-pgroup` attribute
 30665 from the attributes object referenced by `attr`.

30666 The `posix_spawnattr_setpgroup()` function shall set the `spawn-pgroup` attribute in an initialized
 30667 attributes object referenced by `attr`.

30668 The `spawn-pgroup` attribute represents the process group to be joined by the new process image
 30669 in a spawn operation (if `POSIX_SPAWN_SETPGROUP` is set in the `spawn-flags` attribute). The
 30670 default value of this attribute shall be zero.

30671 **RETURN VALUE**

30672 Upon successful completion, `posix_spawnattr_getpgroup()` shall return zero and store the value of
 30673 the `spawn-pgroup` attribute of `attr` into the object referenced by the `pgroup` parameter; otherwise,
 30674 an error number shall be returned to indicate the error.

30675 Upon successful completion, `posix_spawnattr_setpgroup()` shall return zero; otherwise, an error
 30676 number shall be returned to indicate the error.

30677 **ERRORS**

30678 These functions may fail if:

30679 [EINVAL] The value specified by `attr` is invalid.

30680 The `posix_spawnattr_setpgroup()` function may fail if:

30681 [EINVAL] The value of the attribute being set is not valid.

30682 **EXAMPLES**

30683 None.

30684 **APPLICATION USAGE**

30685 These functions are part of the Spawn option and need not be provided on all implementations.

30686 **RATIONALE**

30687 None.

30688 **FUTURE DIRECTIONS**

30689 None.

30690 **SEE ALSO**

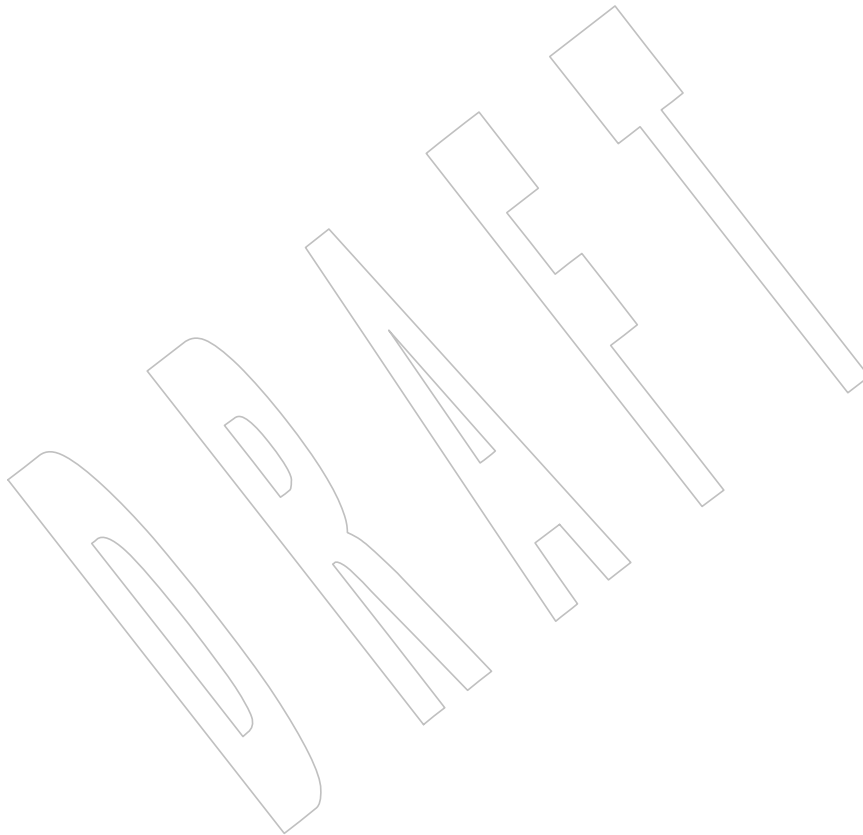
30691 [*posix_spawn\(\)*](#), [*posix_spawnattr_destroy\(\)*](#), [*posix_spawnattr_init\(\)*](#), [*posix_spawnattr_getsigdefault\(\)*](#),
 30692 [*posix_spawnattr_getflags\(\)*](#), [*posix_spawnattr_getschedparam\(\)*](#), [*posix_spawnattr_getschedpolicy\(\)*](#),
 30693 [*posix_spawnattr_getsigmask\(\)*](#), [*posix_spawnattr_setsigdefault\(\)*](#), [*posix_spawnattr_setflags\(\)*](#),
 30694 [*posix_spawnattr_setschedparam\(\)*](#), [*posix_spawnattr_setschedpolicy\(\)*](#), [*posix_spawnattr_setsigmask\(\)*](#),
 30695 [*posix_spawnnp\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

30696

30697

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.



30698 **NAME**

30699 `posix_spawnattr_getschedparam`, `posix_spawnattr_setschedparam` — get and set the spawn-
 30700 `schedparam` attribute of a spawn attributes object (**ADVANCED REALTIME**)

30701 **SYNOPSIS**

```
30702 SPN PS #include <spawn.h>
30703 #include <sched.h>
30704
30705 int posix_spawnattr_getschedparam(const posix_spawnattr_t *
30706     restrict attr, struct sched_param *restrict schedparam);
30707 int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,
30708     const struct sched_param *restrict schedparam);
```

30708 **DESCRIPTION**

30709 The `posix_spawnattr_getschedparam()` function shall obtain the value of the `spawn-schedparam`
 30710 attribute from the attributes object referenced by `attr`.

30711 The `posix_spawnattr_setschedparam()` function shall set the `spawn-schedparam` attribute in an
 30712 initialized attributes object referenced by `attr`.

30713 The `spawn-schedparam` attribute represents the scheduling parameters to be assigned to the new
 30714 process image in a spawn operation (if `POSIX_SPAWN_SETSCHEDULER` or
 30715 `POSIX_SPAWN_SETSCHEDPARAM` is set in the `spawn-flags` attribute). The default value of this
 30716 attribute is unspecified.

30717 **RETURN VALUE**

30718 Upon successful completion, `posix_spawnattr_getschedparam()` shall return zero and store the
 30719 value of the `spawn-schedparam` attribute of `attr` into the object referenced by the `schedparam`
 30720 parameter; otherwise, an error number shall be returned to indicate the error.

30721 Upon successful completion, `posix_spawnattr_setschedparam()` shall return zero; otherwise, an
 30722 error number shall be returned to indicate the error.

30723 **ERRORS**

30724 These functions may fail if:

30725 [EINVAL] The value specified by `attr` is invalid.

30726 The `posix_spawnattr_setschedparam()` function may fail if:

30727 [EINVAL] The value of the attribute being set is not valid.

30728 **EXAMPLES**

30729 None.

30730 **APPLICATION USAGE**

30731 These functions are part of the Spawn and Process Scheduling options and need not be provided
 30732 on all implementations.

30733 **RATIONALE**

30734 None.

30735 **FUTURE DIRECTIONS**

30736 None.

30737

SEE ALSO

30738

30739

30740

30741

30742

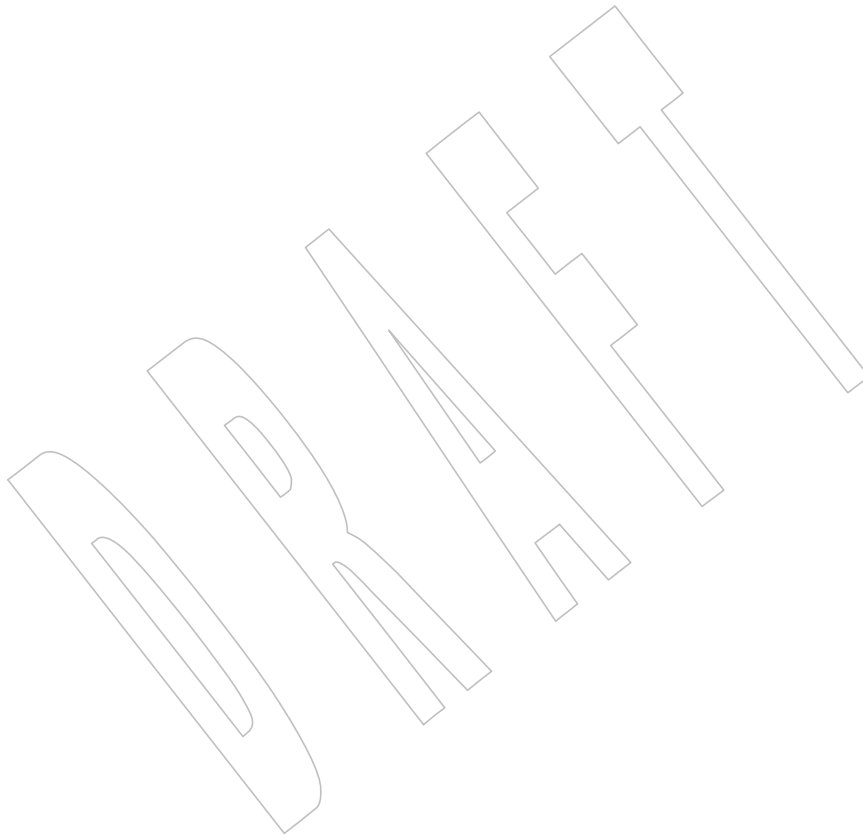
posix_spawn(), *posix_spawnattr_destroy()*, *posix_spawnattr_init()*, *posix_spawnattr_getsigdefault()*,
posix_spawnattr_getflags(), *posix_spawnattr_getpgroup()*, *posix_spawnattr_getschedpolicy()*,
posix_spawnattr_getsigmask(), *posix_spawnattr_setsigdefault()*, *posix_spawnattr_setflags()*,
posix_spawnattr_setpgroup(), *posix_spawnattr_setschedpolicy()*, *posix_spawnattr_setsigmask()*,
posix_spawnnp(), the Base Definitions volume of IEEE Std 1003.1-200x, `<sched.h>`, `<spawn.h>`

30743

CHANGE HISTORY

30744

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.



30745 **NAME**

30746 `posix_spawnattr_getschedpolicy`, `posix_spawnattr_setschedpolicy` — get and set the spawn-
 30747 `schedpolicy` attribute of a spawn attributes object (**ADVANCED REALTIME**)

30748 **SYNOPSIS**

```
30749 SPN PS #include <spawn.h>
30750 #include <sched.h>
30751
30752 int posix_spawnattr_getschedpolicy(const posix_spawnattr_t *
30753     restrict attr, int *restrict schedpolicy);
30754 int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
30755     int schedpolicy);
```

30755 **DESCRIPTION**

30756 The `posix_spawnattr_getschedpolicy()` function shall obtain the value of the *spawn-schedpolicy*
 30757 attribute from the attributes object referenced by *attr*.

30758 The `posix_spawnattr_setschedpolicy()` function shall set the *spawn-schedpolicy* attribute in an
 30759 initialized attributes object referenced by *attr*.

30760 The *spawn-schedpolicy* attribute represents the scheduling policy to be assigned to the new
 30761 process image in a spawn operation (if `POSIX_SPAWN_SETSCHEDULER` is set in the *spawn-*
 30762 *flags* attribute). The default value of this attribute is unspecified.

30763 **RETURN VALUE**

30764 Upon successful completion, `posix_spawnattr_getschedpolicy()` shall return zero and store the
 30765 value of the *spawn-schedpolicy* attribute of *attr* into the object referenced by the *schedpolicy*
 30766 parameter; otherwise, an error number shall be returned to indicate the error.

30767 Upon successful completion, `posix_spawnattr_setschedpolicy()` shall return zero; otherwise, an
 30768 error number shall be returned to indicate the error.

30769 **ERRORS**

30770 These functions may fail if:

30771 [EINVAL] The value specified by *attr* is invalid.

30772 The `posix_spawnattr_setschedpolicy()` function may fail if:

30773 [EINVAL] The value of the attribute being set is not valid.

30774 **EXAMPLES**

30775 None.

30776 **APPLICATION USAGE**

30777 These functions are part of the Spawn and Process Scheduling options and need not be provided
 30778 on all implementations.

30779 **RATIONALE**

30780 None.

30781 **FUTURE DIRECTIONS**

30782 None.

30783 **SEE ALSO**

30784 [posix_spawn\(\)](#), [posix_spawnattr_destroy\(\)](#), [posix_spawnattr_init\(\)](#), [posix_spawnattr_getsigdefault\(\)](#),
 30785 [posix_spawnattr_getflags\(\)](#), [posix_spawnattr_getpgroup\(\)](#), [posix_spawnattr_getschedparam\(\)](#),
 30786 [posix_spawnattr_getsigmask\(\)](#), [posix_spawnattr_setsigdefault\(\)](#), [posix_spawnattr_setflags\(\)](#),
 30787 [posix_spawnattr_setpgroup\(\)](#), [posix_spawnattr_setschedparam\(\)](#), [posix_spawnattr_setsigmask\(\)](#),

30788

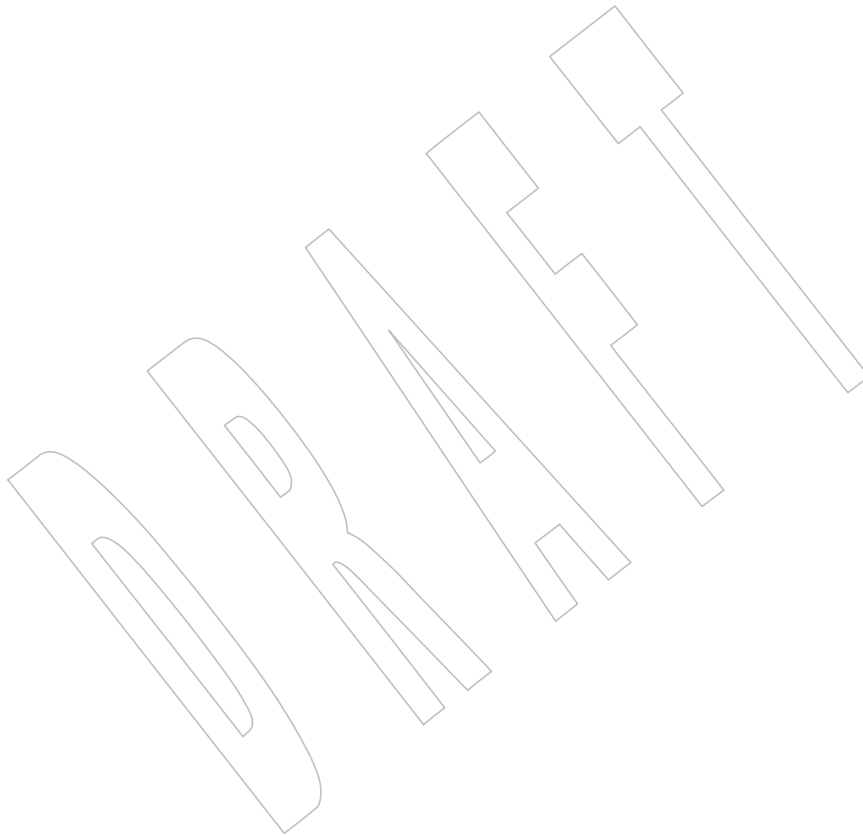
posix_spawnp(), the Base Definitions volume of IEEE Std 1003.1-200x, <sched.h>, <spawn.h>

30789

CHANGE HISTORY

30790

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.



30791 **NAME**

30792 `posix_spawnattr_getsigdefault`, `posix_spawnattr_setsigdefault` — get and set the spawn-
 30793 `sigdefault` attribute of a spawn attributes object (**ADVANCED REALTIME**)

30794 **SYNOPSIS**

```
30795 SPN #include <signal.h>
30796 #include <spawn.h>
30797
30797 int posix_spawnattr_getsigdefault(const posix_spawnattr_t *
30798     restrict attr, sigset_t *restrict sigdefault);
30799 int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr,
30800     const sigset_t *restrict sigdefault);
```

30801 **DESCRIPTION**

30802 The `posix_spawnattr_getsigdefault()` function shall obtain the value of the `spawn-sigdefault`
 30803 attribute from the attributes object referenced by `attr`.

30804 The `posix_spawnattr_setsigdefault()` function shall set the `spawn-sigdefault` attribute in an
 30805 initialized attributes object referenced by `attr`.

30806 The `spawn-sigdefault` attribute represents the set of signals to be forced to default signal handling
 30807 in the new process image (if `POSIX_SPAWN_SETSIGDEF` is set in the `spawn-flags` attribute) by a
 30808 spawn operation. The default value of this attribute shall be an empty signal set.

30809 **RETURN VALUE**

30810 Upon successful completion, `posix_spawnattr_getsigdefault()` shall return zero and store the value
 30811 of the `spawn-sigdefault` attribute of `attr` into the object referenced by the `sigdefault` parameter;
 30812 otherwise, an error number shall be returned to indicate the error.

30813 Upon successful completion, `posix_spawnattr_setsigdefault()` shall return zero; otherwise, an error
 30814 number shall be returned to indicate the error.

30815 **ERRORS**

30816 These functions may fail if:

30817 [EINVAL] The value specified by `attr` is invalid.

30818 The `posix_spawnattr_setsigdefault()` function may fail if:

30819 [EINVAL] The value of the attribute being set is not valid.

30820 **EXAMPLES**

30821 None.

30822 **APPLICATION USAGE**

30823 These functions are part of the Spawn option and need not be provided on all implementations.

30824 **RATIONALE**

30825 None.

30826 **FUTURE DIRECTIONS**

30827 None.

30828 **SEE ALSO**

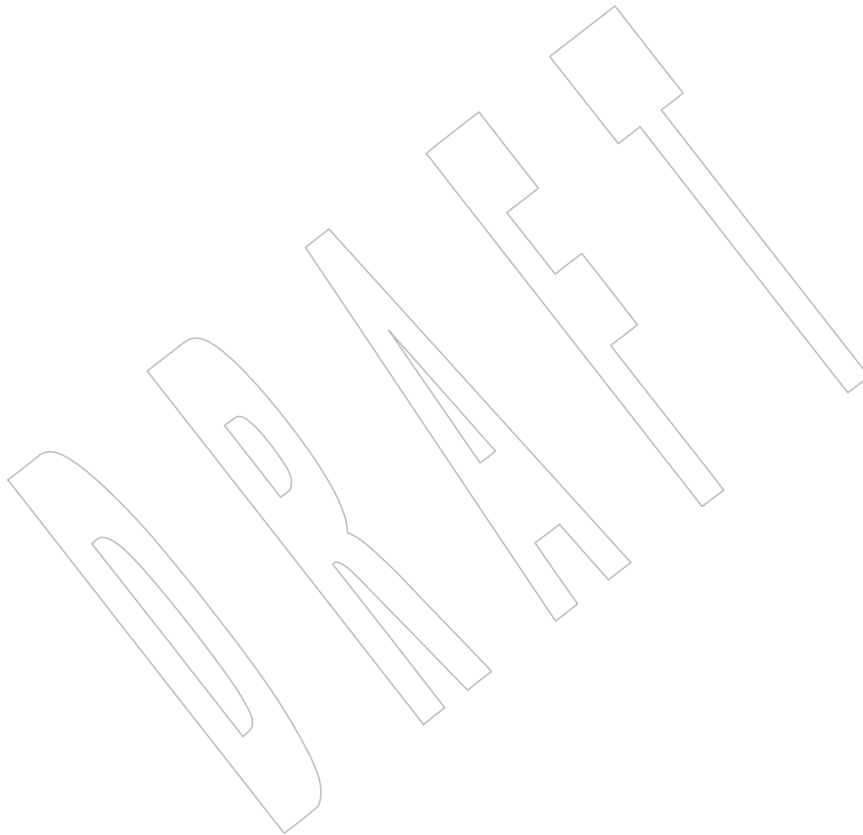
30829 [posix_spawn\(\)](#), [posix_spawnattr_destroy\(\)](#), [posix_spawnattr_init\(\)](#), [posix_spawnattr_getflags\(\)](#),
 30830 [posix_spawnattr_getpgroup\(\)](#), [posix_spawnattr_getschedparam\(\)](#), [posix_spawnattr_getschedpolicy\(\)](#),
 30831 [posix_spawnattr_getsigmask\(\)](#), [posix_spawnattr_setflags\(\)](#), [posix_spawnattr_setpgroup\(\)](#),
 30832 [posix_spawnattr_setschedparam\(\)](#), [posix_spawnattr_setschedpolicy\(\)](#), [posix_spawnattr_setsigmask\(\)](#),
 30833 [posix_spawnnp\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, [<signal.h>](#), [<spawn.h>](#)

30834

30835

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.



30836 **NAME**

30837 `posix_spawnattr_getsigmask`, `posix_spawnattr_setsigmask` — get and set the spawn-sigmask
 30838 attribute of a spawn attributes object (**ADVANCED REALTIME**)

30839 **SYNOPSIS**

```
30840 SPN #include <signal.h>
30841 #include <spawn.h>
30842
30842 int posix_spawnattr_getsigmask(const posix_spawnattr_t *restrict attr,
30843 sigset_t *restrict sigmask);
30844 int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr,
30845 const sigset_t *restrict sigmask);
```

30846 **DESCRIPTION**

30847 The `posix_spawnattr_getsigmask()` function shall obtain the value of the *spawn-sigmask* attribute
 30848 from the attributes object referenced by *attr*.

30849 The `posix_spawnattr_setsigmask()` function shall set the *spawn-sigmask* attribute in an initialized
 30850 attributes object referenced by *attr*.

30851 The *spawn-sigmask* attribute represents the signal mask in effect in the new process image of a
 30852 spawn operation (if `POSIX_SPAWN_SETSIGMASK` is set in the *spawn-flags* attribute). The
 30853 default value of this attribute is unspecified.

30854 **RETURN VALUE**

30855 Upon successful completion, `posix_spawnattr_getsigmask()` shall return zero and store the value
 30856 of the *spawn-sigmask* attribute of *attr* into the object referenced by the *sigmask* parameter;
 30857 otherwise, an error number shall be returned to indicate the error.

30858 Upon successful completion, `posix_spawnattr_setsigmask()` shall return zero; otherwise, an error
 30859 number shall be returned to indicate the error.

30860 **ERRORS**

30861 These functions may fail if:

30862 [EINVAL] The value specified by *attr* is invalid.

30863 The `posix_spawnattr_setsigmask()` function may fail if:

30864 [EINVAL] The value of the attribute being set is not valid.

30865 **EXAMPLES**

30866 None.

30867 **APPLICATION USAGE**

30868 These functions are part of the Spawn option and need not be provided on all implementations.

30869 **RATIONALE**

30870 None.

30871 **FUTURE DIRECTIONS**

30872 None.

30873 **SEE ALSO**

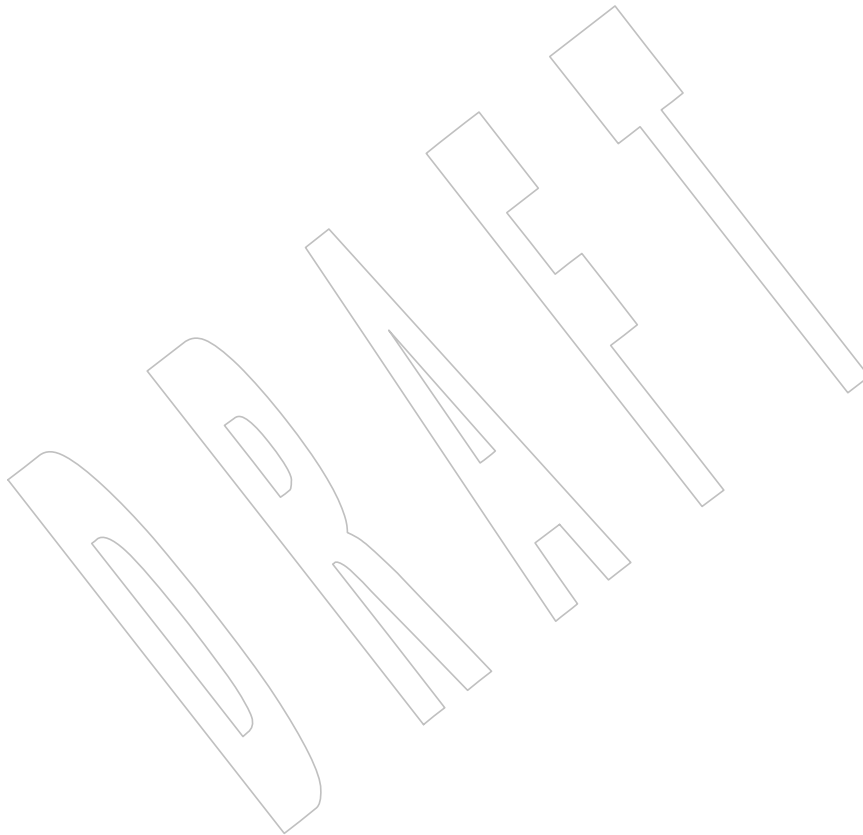
30874 [posix_spawn\(\)](#), [posix_spawnattr_destroy\(\)](#), [posix_spawnattr_init\(\)](#), [posix_spawnattr_getsigdefault\(\)](#),
 30875 [posix_spawnattr_getflags\(\)](#), [posix_spawnattr_getpgroup\(\)](#), [posix_spawnattr_getschedparam\(\)](#),
 30876 [posix_spawnattr_getschedpolicy\(\)](#), [posix_spawnattr_setsigdefault\(\)](#), [posix_spawnattr_setflags\(\)](#),
 30877 [posix_spawnattr_setpgroup\(\)](#), [posix_spawnattr_setschedparam\(\)](#), [posix_spawnattr_setschedpolicy\(\)](#),
 30878 [posix_spawnnp\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, [<signal.h>](#), [<spawn.h>](#)

30879

30880

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.



30881 **NAME**
30882 `posix_spawnattr_init` — initialize the spawn attributes object (**ADVANCED REALTIME**)

30883 **SYNOPSIS**

30884 SPN `#include <spawn.h>`
30885 `int posix_spawnattr_init(posix_spawnattr_t *attr);`

30886 **DESCRIPTION**

30887 Refer to [*posix_spawnattr_destroy\(\)*](#).

30888 **NAME**
30889 `posix_spawnattr_setflags` — set the spawn-flags attribute of a spawn attributes object
30890 (**ADVANCED REALTIME**)

30891 **SYNOPSIS**

```
30892 SPN #include <spawn.h>  
30893 int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
```

30894 **DESCRIPTION**

30895 Refer to [posix_spawnattr_getflags\(\)](#).

posix_spawnattr_setpgroup()30896 **NAME**

30897 `posix_spawnattr_setpgroup` — set the spawn-pgroup attribute of a spawn attributes object
30898 (**ADVANCED REALTIME**)

30899 **SYNOPSIS**

```
30900 SPN #include <spawn.h>  
30901 int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
```

30902 **DESCRIPTION**

30903 Refer to *posix_spawnattr_getpgroup()*.

30904 **NAME**

30905 `posix_spawnattr_setschedparam` — set the spawn-schedparam attribute of a spawn attributes
30906 object (**ADVANCED REALTIME**)

30907 **SYNOPSIS**

```
30908 SPN PS #include <sched.h>  
30909 #include <spawn.h>  
  
30910 int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,  
30911     const struct sched_param *restrict schedparam);
```

30912 **DESCRIPTION**

30913 Refer to [posix_spawnattr_getschedparam\(\)](#).

30914 **NAME**

30915 `posix_spawnattr_setschedpolicy` — set the spawn-schedpolicy attribute of a spawn attributes
30916 object (**ADVANCED REALTIME**)

30917 **SYNOPSIS**

```
30918 SPN PS #include <sched.h>  
30919 #include <spawn.h>  
  
30920 int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,  
30921 int schedpolicy);
```

30922 **DESCRIPTION**

30923 Refer to [posix_spawnattr_getschedpolicy\(\)](#).

30924 **NAME**

30925 `posix_spawnattr_setsigdefault` — set the spawn-sigdefault attribute of a spawn attributes object
30926 (**ADVANCED REALTIME**)

30927 **SYNOPSIS**

```
30928 SPN #include <signal.h>  
30929 #include <spawn.h>  
  
30930 int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr,  
30931 const sigset_t *restrict sigdefault);
```

30932 **DESCRIPTION**

30933 Refer to [posix_spawnattr_getsigdefault\(\)](#).

30934 **NAME**

30935 `posix_spawnattr_setsigmask` — set the spawn-sigmask attribute of a spawn attributes object
30936 (**ADVANCED REALTIME**)

30937 **SYNOPSIS**

```
30938 SPN #include <signal.h>  
30939 #include <spawn.h>  
  
30940 int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr,  
30941 const sigset_t *restrict sigmask);
```

30942 **DESCRIPTION**

30943 Refer to [posix_spawnattr_getsigmask\(\)](#).

30944 **NAME**
30945 `posix_spawn` — spawn a process (**ADVANCED REALTIME**)

30946 **SYNOPSIS**

```
30947 SPN #include <spawn.h>  
30948 int posix_spawn(pid_t *restrict pid, const char *restrict file,  
30949 const posix_spawn_file_actions_t *file_actions,  
30950 const posix_spawnattr_t *restrict attrp,  
30951 char *const argv[restrict], char *const envp[restrict]);
```

30952 **DESCRIPTION**

30953 Refer to *posix_spawn()*.

30954 **NAME**

30955 `posix_trace_attr_destroy`, `posix_trace_attr_init` — destroy and initialize the trace stream
 30956 attributes object (**TRACING**)

30957 **SYNOPSIS**

```
30958 OB TRC #include <trace.h>
30959
30959 int posix_trace_attr_destroy(trace_attr_t *attr);
30960 int posix_trace_attr_init(trace_attr_t *attr);
```

30961 **DESCRIPTION**

30962 The `posix_trace_attr_destroy()` function shall destroy an initialized trace attributes object. A
 30963 destroyed `attr` attributes object can be reinitialized using `posix_trace_attr_init()`; the results of
 30964 otherwise referencing the object after it has been destroyed are undefined.

30965 The `posix_trace_attr_init()` function shall initialize a trace attributes object `attr` with the default
 30966 value for all of the individual attributes used by a given implementation. The read-only
 30967 *generation-version* and *clock-resolution* attributes of the newly initialized trace attributes object
 30968 shall be set to their appropriate values (see [Section 2.11.1.2](#) (on page 77)).

30969 Results are undefined if `posix_trace_attr_init()` is called specifying an already initialized `attr`
 30970 attributes object.

30971 Implementations may add extensions to the trace attributes object structure as permitted in the
 30972 Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance.

30973 The resulting attributes object (possibly modified by setting individual attributes values), when
 30974 used by `posix_trace_create()`, defines the attributes of the trace stream created. A single attributes
 30975 object can be used in multiple calls to `posix_trace_create()`. After one or more trace streams have
 30976 been created using an attributes object, any function affecting that attributes object, including
 30977 destruction, shall not affect any trace stream previously created. An initialized attributes object
 30978 also serves to receive the attributes of an existing trace stream or trace log when calling the
 30979 `posix_trace_get_attr()` function.

30980 **RETURN VALUE**

30981 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 30982 return the corresponding error number.

30983 **ERRORS**

30984 The `posix_trace_attr_destroy()` function may fail if:

30985 [EINVAL] The value of `attr` is invalid.

30986 The `posix_trace_attr_init()` function shall fail if:

30987 [ENOMEM] Insufficient memory exists to initialize the trace attributes object.

30988 **EXAMPLES**

30989 None.

30990 **APPLICATION USAGE**

30991 None.

30992 **RATIONALE**

30993 None.

30994
30995
30996
30997
30998
30999
31000
31001
31002
31003
31004

FUTURE DIRECTIONS

The *posix_trace_attr_destroy()* and *posix_trace_attr_init()* functions may be removed in a future version.

SEE ALSO

posix_trace_create(), *posix_trace_get_attr()*, *uname()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

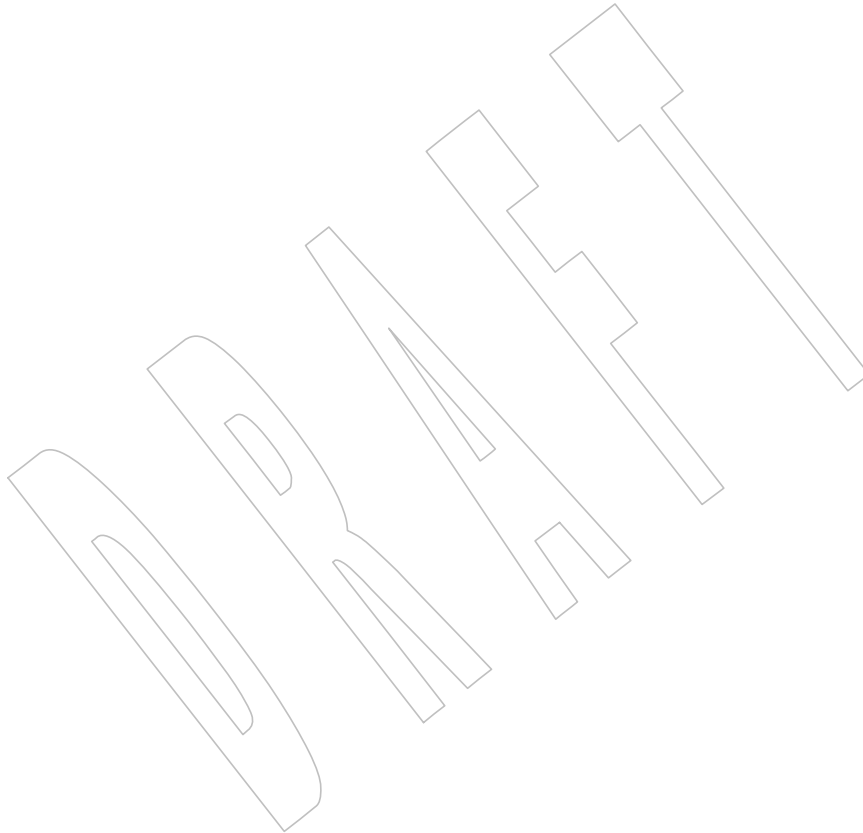
CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE PASC Interpretation 1003.1 #123 is applied.

Issue 7

The *posix_trace_attr_destroy()* and *posix_trace_attr_init()* functions are marked obsolescent.



31005 **NAME**

31006 `posix_trace_attr_getclockres`, `posix_trace_attr_getcreatetime`, `posix_trace_attr_getgenversion`,
 31007 `posix_trace_attr_getname`, `posix_trace_attr_setname` — retrieve and set information about a
 31008 trace stream (**TRACING**)

31009 **SYNOPSIS**

```
31010 OB TRC #include <time.h>
31011 #include <trace.h>
31012
31012 int posix_trace_attr_getclockres(const trace_attr_t *attr,
31013 struct timespec *resolution);
31014 int posix_trace_attr_getcreatetime(const trace_attr_t *attr,
31015 struct timespec *createtime);
31016
31016 #include <trace.h>
31017
31017 int posix_trace_attr_getgenversion(const trace_attr_t *attr,
31018 char *genversion);
31019 int posix_trace_attr_getname(const trace_attr_t *attr,
31020 char *tracename);
31021 int posix_trace_attr_setname(trace_attr_t *attr,
31022 const char *tracename);
```

31023 **DESCRIPTION**

31024 The `posix_trace_attr_getclockres()` function shall copy the clock resolution of the clock used to
 31025 generate timestamps from the *clock-resolution* attribute of the attributes object pointed to by the
 31026 *attr* argument into the structure pointed to by the *resolution* argument.

31027 The `posix_trace_attr_getcreatetime()` function shall copy the trace stream creation time from the
 31028 *creation-time* attribute of the attributes object pointed to by the *attr* argument into the structure
 31029 pointed to by the *createtime* argument. The *creation-time* attribute shall represent the time of
 31030 creation of the trace stream.

31031 The `posix_trace_attr_getgenversion()` function shall copy the string containing version information
 31032 from the *generation-version* attribute of the attributes object pointed to by the *attr* argument into
 31033 the string pointed to by the *genversion* argument. The *genversion* argument shall be the address of
 31034 a character array which can store at least {TRACE_NAME_MAX} characters.

31035 The `posix_trace_attr_getname()` function shall copy the string containing the trace name from the
 31036 *trace-name* attribute of the attributes object pointed to by the *attr* argument into the string
 31037 pointed to by the *tracename* argument. The *tracename* argument shall be the address of a character
 31038 array which can store at least {TRACE_NAME_MAX} characters.

31039 The `posix_trace_attr_setname()` function shall set the name in the *trace-name* attribute of the
 31040 attributes object pointed to by the *attr* argument, using the trace name string supplied by the
 31041 *tracename* argument. If the supplied string contains more than {TRACE_NAME_MAX}
 31042 characters, the name copied into the *trace-name* attribute may be truncated to one less than the
 31043 length of {TRACE_NAME_MAX} characters. The default value is a null string.

31044 **RETURN VALUE**

31045 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 31046 return the corresponding error number.

31047 If successful, the `posix_trace_attr_getclockres()` function stores the *clock-resolution* attribute value in
 31048 the object pointed to by *resolution*. Otherwise, the content of this object is unspecified.

31049 If successful, the `posix_trace_attr_getcreatetime()` function stores the trace stream creation time in

31050 the object pointed to by *createtime*. Otherwise, the content of this object is unspecified.

31051 If successful, the *posix_trace_attr_getgenversion()* function stores the trace version information in
31052 the string pointed to by *genversion*. Otherwise, the content of this string is unspecified.

31053 If successful, the *posix_trace_attr_getname()* function stores the trace name in the string pointed
31054 to by *tracename*. Otherwise, the content of this string is unspecified.

31055 **ERRORS**

31056 The *posix_trace_attr_getclockres()*, *posix_trace_attr_getcreatetime()*, *posix_trace_attr_getgenversion()*,
31057 and *posix_trace_attr_getname()* functions may fail if:

31058 [EINVAL] The value specified by one of the arguments is invalid.

31059 **EXAMPLES**

31060 None.

31061 **APPLICATION USAGE**

31062 None.

31063 **RATIONALE**

31064 None.

31065 **FUTURE DIRECTIONS**

31066 The *posix_trace_attr_getclockres()*, *posix_trace_attr_getcreatetime()*, *posix_trace_attr_getgenversion()*,
31067 *posix_trace_attr_getname()*, and *posix_trace_attr_setname()* functions may be removed in a future
31068 version.

31069 **SEE ALSO**

31070 *posix_trace_attr_init()*, *posix_trace_create()*, *posix_trace_get_attr()*, *uname()*, the Base Definitions
31071 volume of IEEE Std 1003.1-200x, [<time.h>](#), [<trace.h>](#)

31072 **CHANGE HISTORY**

31073 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

31074 **Issue 7**

31075 The *posix_trace_attr_getclockres()*, *posix_trace_attr_getcreatetime()*, *posix_trace_attr_getgenversion()*,
31076 *posix_trace_attr_getname()*, and *posix_trace_attr_setname()* functions are marked obsolescent.

31077 **NAME**

31078 posix_trace_attr_getinherited, posix_trace_attr_getlogfullpolicy,
 31079 posix_trace_attr_getstreamfullpolicy, posix_trace_attr_setinherited,
 31080 posix_trace_attr_setlogfullpolicy, posix_trace_attr_setstreamfullpolicy — retrieve and set the
 31081 behavior of a trace stream (**TRACING**)

31082 **SYNOPSIS**

```
31083 OB TRC #include <trace.h>
31084 TRI int posix_trace_attr_getinherited(const trace_attr_t *restrict attr,
31085 int *restrict inheritancepolicy);
31086 TRI int posix_trace_attr_getlogfullpolicy(const trace_attr_t *restrict attr,
31087 int *restrict logpolicy);
31088 int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *restrict
31089 attr, int *restrict streampolicy);
31090 TRI int posix_trace_attr_setinherited(trace_attr_t *attr,
31091 int inheritancepolicy);
31092 TRI int posix_trace_attr_setlogfullpolicy(trace_attr_t *attr,
31093 int logpolicy);
31094 int posix_trace_attr_setstreamfullpolicy(trace_attr_t *attr,
31095 int streampolicy);
```

31096 **DESCRIPTION**

31097 TRI The *posix_trace_attr_getinherited()* and *posix_trace_attr_setinherited()* functions, respectively, shall
 31098 get and set the inheritance policy stored in the *inheritance* attribute for traced processes across the
 31099 *fork()* and *spawn()* operations. The *inheritance* attribute of the attributes object pointed to by the
 31100 *attr* argument shall be set to one of the following values defined by manifest constants in the
 31101 **<trace.h>** header:

31102 **POSIX_TRACE_CLOSE_FOR_CHILD**

31103 After a *fork()* or *spawn()* operation, the child shall not be traced, and tracing of the parent
 31104 shall continue.

31105 **POSIX_TRACE_INHERITED**

31106 After a *fork()* or *spawn()* operation, if the parent is being traced, its child shall be
 31107 concurrently traced using the same trace stream.

31108 The default value for the *inheritance* attribute is **POSIX_TRACE_CLOSE_FOR_CHILD**.

31109 TRI The *posix_trace_attr_getlogfullpolicy()* and *posix_trace_attr_setlogfullpolicy()* functions,
 31110 respectively, shall get and set the trace log full policy stored in the *log-full-policy* attribute of the
 31111 attributes object pointed to by the *attr* argument.

31112 The *log-full-policy* attribute shall be set to one of the following values defined by manifest
 31113 constants in the **<trace.h>** header:

31114 **POSIX_TRACE_LOOP**

31115 The trace log shall loop until the associated trace stream is stopped. This policy means that
 31116 when the trace log gets full, the file system shall reuse the resources allocated to the oldest
 31117 trace events that were recorded. In this way, the trace log will always contain the most
 31118 recent trace events flushed.

31119 **POSIX_TRACE_UNTIL_FULL**

31120 The trace stream shall be flushed to the trace log until the trace log is full. This condition can
 31121 be deduced from the *posix_log_full_status* member status (see the **posix_trace_status_info**
 31122 structure defined in **<trace.h>**). The last recorded trace event shall be the
 31123 **POSIX_TRACE_STOP** trace event.

- 31124 **POSIX_TRACE_APPEND**
- 31125 The associated trace stream shall be flushed to the trace log without log size limitation. If
- 31126 the application specifies `POSIX_TRACE_APPEND`, the implementation shall ignore the *log-*
- 31127 *max-size* attribute.
- 31128 The default value for the *log-full-policy* attribute is `POSIX_TRACE_LOOP`.
- 31129 The *posix_trace_attr_getstreamfullpolicy()* and *posix_trace_attr_setstreamfullpolicy()* functions,
- 31130 respectively, shall get and set the trace stream full policy stored in the *stream-full-policy* attribute
- 31131 of the attributes object pointed to by the *attr* argument.
- 31132 The *stream-full-policy* attribute shall be set to one of the following values defined by manifest
- 31133 constants in the `<trace.h>` header:
- 31134 **POSIX_TRACE_LOOP**
- 31135 The trace stream shall loop until explicitly stopped by the *posix_trace_stop()* function. This
- 31136 policy means that when the trace stream is full, the trace system shall reuse the resources
- 31137 allocated to the oldest trace events recorded. In this way, the trace stream will always
- 31138 contain the most recent trace events recorded.
- 31139 **POSIX_TRACE_UNTIL_FULL**
- 31140 The trace stream will run until the trace stream resources are exhausted. Then the trace
- 31141 stream will stop. This condition can be deduced from *posix_stream_status* and
- 31142 *posix_stream_full_status* (see the **posix_trace_status_info** structure defined in `<trace.h>`).
- 31143 When this trace stream is read, a `POSIX_TRACE_STOP` trace event shall be reported after
- 31144 reporting the last recorded trace event. The trace system shall reuse the resources allocated
- 31145 to any trace events already reported—see the *posix_trace_getnext_event()*,
- 31146 *posix_trace_trygetnext_event()*, and *posix_trace_timedgetnext_event()* functions—or already
- 31147 flushed for an active trace stream with log if the Trace Log option is supported; see the
- 31148 *posix_trace_flush()* function. The trace system shall restart the trace stream when it is empty
- 31149 and may restart it sooner. A `POSIX_TRACE_START` trace event shall be reported before
- 31150 reporting the next recorded trace event.
- 31151 **POSIX_TRACE_FLUSH**
- 31152 If the Trace Log option is supported, this policy is identical to the
- 31153 `POSIX_TRACE_UNTIL_FULL` trace stream full policy except that the trace stream shall be
- 31154 flushed regularly as if *posix_trace_flush()* had been explicitly called. Defining this policy for
- 31155 an active trace stream without log shall be invalid.
- 31156 The default value for the *stream-full-policy* attribute shall be `POSIX_TRACE_LOOP` for an active
- 31157 trace stream without log.
- 31158 If the Trace Log option is supported, the default value for the *stream-full-policy* attribute shall be
- 31159 `POSIX_TRACE_FLUSH` for an active trace stream with log.
- 31160 **RETURN VALUE**
- 31161 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
- 31162 return the corresponding error number.
- 31163 If successful, the *posix_trace_attr_getinherited()* function shall store the *inheritance* attribute value
- 31164 in the object pointed to by *inheritancepolicy*. Otherwise, the content of this object is undefined.
- 31165 If successful, the *posix_trace_attr_getlogfullpolicy()* function shall store the *log-full-policy* attribute
- 31166 value in the object pointed to by *logpolicy*. Otherwise, the content of this object is undefined.
- 31167 If successful, the *posix_trace_attr_getstreamfullpolicy()* function shall store the *stream-full-policy*
- 31168 attribute value in the object pointed to by *streampolicy*. Otherwise, the content of this object is
- 31169 undefined.

posix_trace_attr_getinherited()31170 **ERRORS**

31171 These functions may fail if:

31172 [EINVAL] The value specified by at least one of the arguments is invalid.

31173 **EXAMPLES**

31174 None.

31175 **APPLICATION USAGE**

31176 None.

31177 **RATIONALE**

31178 None.

31179 **FUTURE DIRECTIONS**

31180 The *posix_trace_attr_getinherited()*, *posix_trace_attr_getlogfullpolicy()*,
 31181 *posix_trace_attr_getstreamfullpolicy()*, *posix_trace_attr_setinherited()*,
 31182 *posix_trace_attr_setlogfullpolicy()*, and *posix_trace_attr_setstreamfullpolicy()* functions may be
 31183 removed in a future version.

31184 **SEE ALSO**

31185 *fork()*, *posix_trace_attr_init()*, *posix_trace_create()*, *posix_trace_flush()*, *posix_trace_get_attr()*,
 31186 *posix_trace_getnext_event()*, *posix_trace_start()*, *posix_trace_timedgetnext_event()*, the Base
 31187 Definitions volume of IEEE Std 1003.1-200x, <trace.h>

31188 **CHANGE HISTORY**

31189 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

31190 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/39 is applied, adding the TRL and TRC
 31191 margin codes to the *posix_trace_attr_setlogfullpolicy()* function.

31192 **Issue 7**

31193 SD5-XSH-ERN-116 is applied, adding the missing **restrict** keyword to the
 31194 *posix_trace_attr_getstreamfullpolicy()* function declaration.

31195 These functions are marked obsolescent.

31196 **NAME**

31197 `posix_trace_attr_getlogsize`, `posix_trace_attr_getmaxdatasize`,
 31198 `posix_trace_attr_getmaxsystemeventsize`, `posix_trace_attr_getmaxusereventsize`,
 31199 `posix_trace_attr_getstreamsize`, `posix_trace_attr_setlogsize`, `posix_trace_attr_setmaxdatasize`,
 31200 `posix_trace_attr_setstreamsize` — retrieve and set trace stream size attributes (**TRACING**)

31201 **SYNOPSIS**

```

31202 OB TRC #include <sys/types.h>
31203 #include <trace.h>

31204 TRL int posix_trace_attr_getlogsize(const trace_attr_t *restrict attr,
31205 size_t *restrict logsize);
31206 int posix_trace_attr_getmaxdatasize(const trace_attr_t *restrict attr,
31207 size_t *restrict maxdatasize);
31208 int posix_trace_attr_getmaxsystemeventsize(
31209 const trace_attr_t *restrict attr,
31210 size_t *restrict eventsize);
31211 int posix_trace_attr_getmaxusereventsize(
31212 const trace_attr_t *restrict attr,
31213 size_t data_len, size_t *restrict eventsize);
31214 int posix_trace_attr_getstreamsize(const trace_attr_t *restrict attr,
31215 size_t *restrict streamsize);
31216 TRL int posix_trace_attr_setlogsize(trace_attr_t *attr,
31217 size_t logsize);
31218 int posix_trace_attr_setmaxdatasize(trace_attr_t *attr,
31219 size_t maxdatasize);
31220 int posix_trace_attr_setstreamsize(trace_attr_t *attr,
31221 size_t streamsize);

```

31222 **DESCRIPTION**

31223 TRL The `posix_trace_attr_getlogsize()` function shall copy the log size, in bytes, from the *log-max-size*
 31224 attribute of the attributes object pointed to by the *attr* argument into the variable pointed to by
 31225 the *logsize* argument. This log size is the maximum total of bytes that shall be allocated for
 31226 system and user trace events in the trace log. The default value for the *log-max-size* attribute is
 31227 implementation-defined.

31228 The `posix_trace_attr_setlogsize()` function shall set the maximum allowed size, in bytes, in the *log-*
 31229 *max-size* attribute of the attributes object pointed to by the *attr* argument, using the size value
 31230 supplied by the *logsize* argument.

31231 The trace log size shall be used if the *log-full-policy* attribute is set to `POSIX_TRACE_LOOP` or
 31232 `POSIX_TRACE_UNTIL_FULL`. If the *log-full-policy* attribute is set to `POSIX_TRACE_APPEND`,
 31233 the implementation shall ignore the *log-max-size* attribute.

31234 The `posix_trace_attr_getmaxdatasize()` function shall copy the maximum user trace event data
 31235 size, in bytes, from the *max-data-size* attribute of the attributes object pointed to by the *attr*
 31236 argument into the variable pointed to by the *maxdatasize* argument. The default value for the
 31237 *max-data-size* attribute is implementation-defined.

31238 The `posix_trace_attr_getmaxsystemeventsize()` function shall calculate the maximum memory size,
 31239 in bytes, required to store a single system trace event. This value is calculated for the trace
 31240 stream attributes object pointed to by the *attr* argument and is returned in the variable pointed
 31241 to by the *eventsize* argument.

31242 The values returned as the maximum memory sizes of the user and system trace events shall be
 31243 such that if the sum of the maximum memory sizes of a set of the trace events that may be

31244 recorded in a trace stream is less than or equal to the *stream-min-size* attribute of that trace
 31245 stream, the system provides the necessary resources for recording all those trace events, without
 31246 loss.

31247 The *posix_trace_attr_getmaxusereventsize()* function shall calculate the maximum memory size, in
 31248 bytes, required to store a single user trace event generated by a call to *posix_trace_event()* with a
 31249 *data_len* parameter equal to the *data_len* value specified in this call. This value is calculated for
 31250 the trace stream attributes object pointed to by the *attr* argument and is returned in the variable
 31251 pointed to by the *eventsize* argument.

31252 The *posix_trace_attr_getstreamsize()* function shall copy the stream size, in bytes, from the *stream-*
 31253 *min-size* attribute of the attributes object pointed to by the *attr* argument into the variable
 31254 pointed to by the *streamsize* argument.

31255 This stream size is the current total memory size reserved for system and user trace events in the
 31256 trace stream. The default value for the *stream-min-size* attribute is implementation-defined. The
 31257 stream size refers to memory used to store trace event records. Other stream data (for example,
 31258 trace attribute values) shall not be included in this size.

31259 The *posix_trace_attr_setmaxdatasize()* function shall set the maximum allowed size, in bytes, in
 31260 the *max-data-size* attribute of the attributes object pointed to by the *attr* argument, using the size
 31261 value supplied by the *maxdatasize* argument. This maximum size is the maximum allowed size
 31262 for the user data argument which may be passed to *posix_trace_event()*. The implementation
 31263 shall be allowed to truncate data passed to *trace_user_event* which is longer than *maxdatasize*.

31264 The *posix_trace_attr_setstreamsize()* function shall set the minimum allowed size, in bytes, in the
 31265 *stream-min-size* attribute of the attributes object pointed to by the *attr* argument, using the size
 31266 value supplied by the *streamsize* argument.

31267 RETURN VALUE

31268 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 31269 return the corresponding error number.

31270 TRL The *posix_trace_attr_getlogsize()* function stores the maximum trace log allowed size in the object
 31271 pointed to by *logsize*, if successful.

31272 The *posix_trace_attr_getmaxdatasize()* function stores the maximum trace event record memory
 31273 size in the object pointed to by *maxdatasize*, if successful.

31274 The *posix_trace_attr_getmaxsystemeventsize()* function stores the maximum memory size to store a
 31275 single system trace event in the object pointed to by *eventsize*, if successful.

31276 The *posix_trace_attr_getmaxusereventsize()* function stores the maximum memory size to store a
 31277 single user trace event in the object pointed to by *eventsize*, if successful.

31278 The *posix_trace_attr_getstreamsize()* function stores the maximum trace stream allowed size in the
 31279 object pointed to by *streamsize*, if successful.

31280 ERRORS

31281 These functions may fail if:

31282 [EINVAL] The value specified by one of the arguments is invalid.

31283

EXAMPLES

31284

None.

31285

APPLICATION USAGE

31286

None.

31287

RATIONALE

31288

None.

31289

FUTURE DIRECTIONS

31290

The `posix_trace_attr_getlogsize()`, `posix_trace_attr_getmaxdatasize()`,

31291

`posix_trace_attr_getmaxsystemeventsizesize()`, `posix_trace_attr_getmaxusereventsizesize()`,

31292

`posix_trace_attr_getstreamsize()`, `posix_trace_attr_setlogsize()`, `posix_trace_attr_setmaxdatasize()`, and

31293

`posix_trace_attr_setstreamsize()` functions may be withdrawn in a future version.

31294

SEE ALSO

31295

`posix_trace_attr_init()`, `posix_trace_create()`, `posix_trace_event()`, `posix_trace_get_attr()`, the Base

31296

Definitions volume of IEEE Std 1003.1-200x, `<sys/types.h>`, `<trace.h>`

31297

CHANGE HISTORY

31298

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

31299

Issue 7

31300

These functions are marked obsolescent.

DRAFT

posix_trace_attr_getname()31301 **NAME**31302 `posix_trace_attr_getname` — retrieve and set information about a trace stream (**TRACING**)31303 **SYNOPSIS**31304 OB TRC `#include <trace.h>`31305 `int posix_trace_attr_getname(const trace_attr_t *attr,`
31306 `char *tracename);`31307 **DESCRIPTION**31308 Refer to [*posix_trace_attr_getclockres\(\)*](#).

31309 **NAME**

31310 `posix_trace_attr_getstreamfullpolicy` — retrieve and set the behavior of a trace stream
31311 (**TRACING**)

31312 **SYNOPSIS**

31313 OB TRC `#include <trace.h>`

```
31314 int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *restrict  
31315 attr, int *restrict streampolicy);
```

31316 **DESCRIPTION**

31317 Refer to [posix_trace_attr_getinherited\(\)](#).

31318 **NAME**31319 posix_trace_attr_getstreamsize — retrieve and set trace stream size attributes (**TRACING**)31320 **SYNOPSIS**

```
31321 OB TRC #include <sys/types.h>  
31322 #include <trace.h>  
  
31323 int posix_trace_attr_getstreamsize(const trace_attr_t *restrict attr,  
31324 size_t *restrict streamsize);
```

31325 **DESCRIPTION**31326 Refer to [posix_trace_attr_getlogsize\(\)](#).

31327 **NAME**
31328 `posix_trace_attr_init` — initialize the trace stream attributes object (**TRACING**)

31329 **SYNOPSIS**

31330 OB TRC `#include <trace.h>`
31331 `int posix_trace_attr_init(trace_attr_t *attr);`

31332 **DESCRIPTION**

31333 Refer to [posix_trace_attr_destroy\(\)](#).

posix_trace_attr_setinherited()31334 **NAME**

31335 `posix_trace_attr_setinherited`, `posix_trace_attr_setlogfullpolicy` — retrieve and set the behavior
 31336 of a trace stream (**TRACING**)

31337 **SYNOPSIS**

```
31338 OB TRC #include <trace.h>
31339 TRI     int posix_trace_attr_setinherited(trace_attr_t *attr,
31340         int inheritancepolicy);
31341 TRL     int posix_trace_attr_setlogfullpolicy(trace_attr_t *attr,
31342         int logpolicy);
```

31343 **DESCRIPTION**

31344 Refer to [*posix_trace_attr_getinherited\(\)*](#).

31345 **NAME**

31346 `posix_trace_attr_setlogsize`, `posix_trace_attr_setmaxdatasize` — retrieve and set trace stream size
 31347 attributes (**TRACING**)

31348 **SYNOPSIS**

```
31349 OB TRC #include <sys/types.h>
31350 #include <trace.h>
31351 TRL int posix_trace_attr_setlogsize(trace_attr_t *attr,
31352 size_t logsize);
31353 TRC int posix_trace_attr_setmaxdatasize(trace_attr_t *attr,
31354 size_t maxdatasize);
```

31355 **DESCRIPTION**

31356 Refer to [posix_trace_attr_getlogsize\(\)](#).

31357 **NAME**31358 `posix_trace_attr_setname` — retrieve and set information about a trace stream (**TRACING**)31359 **SYNOPSIS**31360 OB TRC `#include <trace.h>`31361 `int posix_trace_attr_setname(trace_attr_t *attr,`
31362 `const char *tracename);`31363 **DESCRIPTION**31364 Refer to [*posix_trace_attr_getclockres\(\)*](#).

31365 **NAME**

31366 `posix_trace_attr_setstreamfullpolicy` — retrieve and set the behavior of a trace stream
31367 (**TRACING**)

31368 **SYNOPSIS**

```
31369 OB TRC #include <trace.h>  
31370  
31370 int posix_trace_attr_setstreamfullpolicy(trace_attr_t *attr,  
31371 int streampolicy);
```

31372 **DESCRIPTION**

31373 Refer to [posix_trace_attr_getinherited\(\)](#).

NAME

posix_trace_attr_setstreamsize — retrieve and set trace stream size attributes (**TRACING**)

SYNOPSIS

```
OB TRC #include <sys/types.h>
        #include <trace.h>

        int posix_trace_attr_setstreamsize(trace_attr_t *attr,
            size_t streamsize);
```

DESCRIPTION

Refer to

31383 **NAME**
 31384 `posix_trace_clear` — clear trace stream and trace log (**TRACING**)

31385 **SYNOPSIS**

```
31386 OB TRC #include <sys/types.h>
31387 #include <trace.h>
31388 int posix_trace_clear(trace_id_t trid);
```

31389 **DESCRIPTION**

31390 The `posix_trace_clear()` function shall reinitialize the trace stream identified by the argument `trid`
 31391 as if it were returning from the `posix_trace_create()` function, except that the same allocated
 31392 resources shall be reused, the mapping of trace event type identifiers to trace event names shall
 31393 be unchanged, and the trace stream status shall remain unchanged (that is, if it was running, it
 31394 remains running and if it was suspended, it remains suspended).

31395 All trace events in the trace stream recorded before the call to `posix_trace_clear()` shall be lost. The
 31396 `posix_stream_full_status` status shall be set to `POSIX_TRACE_NOT_FULL`. There is no guarantee
 31397 that all trace events that occurred during the `posix_trace_clear()` call are recorded; the behavior
 31398 with respect to trace points that may occur during this call is unspecified.

31399 OB TRL If the Trace Log option is supported and the trace stream has been created with a log, the
 31400 `posix_trace_clear()` function shall reinitialize the trace stream with the same behavior as if the
 31401 trace stream was created without the log, plus it shall reinitialize the trace log associated with
 31402 the trace stream identified by the argument `trid` as if it were returning from the
 31403 `posix_trace_create_withlog()` function, except that the same allocated resources, for the trace log,
 31404 may be reused and the associated trace stream status remains unchanged. The first trace event
 31405 recorded in the trace log after the call to `posix_trace_clear()` shall be the same as the first trace
 31406 event recorded in the active trace stream after the call to `posix_trace_clear()`. The
 31407 `posix_log_full_status` status shall be set to `POSIX_TRACE_NOT_FULL`. There is no guarantee that
 31408 all trace events that occurred during the `posix_trace_clear()` call are recorded in the trace log; the
 31409 behavior with respect to trace points that may occur during this call is unspecified. If the log full
 31410 policy is `POSIX_TRACE_APPEND`, the effect of a call to this function is unspecified for the trace
 31411 log associated with the trace stream identified by the `trid` argument.

31412 **RETURN VALUE**

31413 Upon successful completion, the `posix_trace_clear()` function shall return a value of zero.
 31414 Otherwise, it shall return the corresponding error number.

31415 **ERRORS**

31416 The `posix_trace_clear()` function shall fail if:

31417 [EINVAL] The value of the `trid` argument does not correspond to an active trace stream.

31418 **EXAMPLES**

31419 None.

31420 **APPLICATION USAGE**

31421 None.

31422 **RATIONALE**

31423 None.

posix_trace_clear()

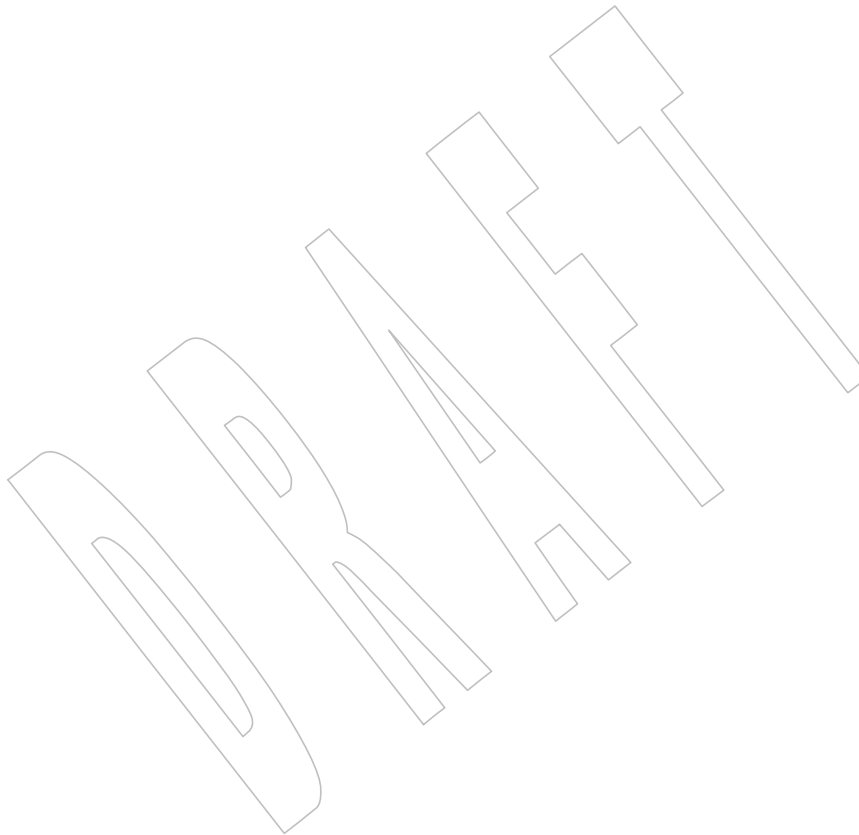
31424 **FUTURE DIRECTIONS**
31425 The *posix_trace_clear()* function may be withdrawn in a future version.

31426 **SEE ALSO**
31427 *posix_trace_attr_init()*, *posix_trace_create()*, *posix_trace_flush()*, *posix_trace_get_attr()*, the Base
31428 Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<trace.h>**

31429 **CHANGE HISTORY**
31430 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

31431 IEEE PASC Interpretation 1003.1 #123 is applied.

31432 **Issue 7**
31433 The *posix_trace_clear()* function is marked obsolescent.



31434 **NAME**31435 `posix_trace_close`, `posix_trace_open`, `posix_trace_rewind` — trace log management (**TRACING**)31436 **SYNOPSIS**31437 OB TRC `#include <trace.h>`31438 TRL `int posix_trace_close(trace_id_t trid);`31439 `int posix_trace_open(int file_desc, trace_id_t *trid);`31440 `int posix_trace_rewind(trace_id_t trid);`31441 **DESCRIPTION**31442 The `posix_trace_close()` function shall deallocate the trace log identifier indicated by `trid`, and all
31443 of its associated resources. If there is no valid trace log pointed to by the `trid`, this function shall
31444 fail.31445 The `posix_trace_open()` function shall allocate the necessary resources and establish the
31446 connection between a trace log identified by the `file_desc` argument and a trace stream identifier
31447 identified by the object pointed to by the `trid` argument. The `file_desc` argument should be a valid
31448 open file descriptor that corresponds to a trace log. The `file_desc` argument shall be open for
31449 reading. The current trace event timestamp, which specifies the timestamp of the trace event that
31450 will be read by the next call to `posix_trace_getnext_event()`, shall be set to the timestamp of the
31451 oldest trace event recorded in the trace log identified by `trid`.31452 The `posix_trace_open()` function shall return a trace stream identifier in the variable pointed to by
31453 the `trid` argument, that may only be used by the following functions:31454 `posix_trace_close()` `posix_trace_get_attr()`
31455 `posix_trace_eventid_equal()` `posix_trace_get_status()`
31456 `posix_trace_eventid_get_name()` `posix_trace_getnext_event()`
31457 `posix_trace_eventtypelist_getnext_id()` `posix_trace_rewind()`
31458 `posix_trace_eventtypelist_rewind()`31459 In particular, notice that the operations normally used by a trace controller process, such as
31460 `posix_trace_start()`, `posix_trace_stop()`, or `posix_trace_shutdown()`, cannot be invoked using the
31461 trace stream identifier returned by the `posix_trace_open()` function.31462 The `posix_trace_rewind()` function shall reset the current trace event timestamp, which specifies
31463 the timestamp of the trace event that will be read by the next call to `posix_trace_getnext_event()`,
31464 to the timestamp of the oldest trace event recorded in the trace log identified by `trid`.31465 **RETURN VALUE**31466 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
31467 return the corresponding error number.31468 If successful, the `posix_trace_open()` function stores the trace stream identifier value in the object
31469 pointed to by `trid`.31470 **ERRORS**31471 The `posix_trace_open()` function shall fail if:

31472 [EINTR] The operation was interrupted by a signal and thus no trace log was opened.

31473 [EINVAL] The object pointed to by `file_desc` does not correspond to a valid trace log.31474 The `posix_trace_close()` and `posix_trace_rewind()` functions may fail if:

posix_trace_close()

31475 [EINVAL] The object pointed to by *trid* does not correspond to a valid trace log.

EXAMPLES

31476 None.
31477

APPLICATION USAGE

31478 None.
31479

RATIONALE

31480 None.
31481

FUTURE DIRECTIONS

31482 The *posix_trace_close()*, *posix_trace_open()*, and *posix_trace_rewind()* functions may be removed in
31483 a future version.
31484

SEE ALSO

31485 *posix_trace_get_attr()*, *posix_trace_get_filter()*, *posix_trace_getnext_event()*, the Base Definitions
31486 volume of IEEE Std 1003.1-200x, <trace.h>
31487

CHANGE HISTORY

31488 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.
31489

31490 IEEE PASC Interpretation 1003.1 #123 is applied.

Issue 7

31491 The *posix_trace_close()*, *posix_trace_open()*, and *posix_trace_rewind()* functions are marked
31492 obsolescent.
31493

DRAFT

31494 **NAME**

31495 `posix_trace_create`, `posix_trace_create_withlog`, `posix_trace_flush`, `posix_trace_shutdown` —
 31496 trace stream initialization, flush, and shutdown from a process (**TRACING**)

31497 **SYNOPSIS**

```
31498 OB TRC #include <sys/types.h>
31499 #include <trace.h>
31500
31501 int posix_trace_create(pid_t pid,
31502     const trace_attr_t *restrict attr,
31503     trace_id_t *restrict trid);
31504 TRL int posix_trace_create_withlog(pid_t pid,
31505     const trace_attr_t *restrict attr, int file_desc,
31506     trace_id_t *restrict trid);
31507 int posix_trace_flush(trace_id_t trid);
31508 int posix_trace_shutdown(trace_id_t trid);
```

31508 **DESCRIPTION**

31509 The `posix_trace_create()` function shall create an active trace stream. It allocates all the resources
 31510 needed by the trace stream being created for tracing the process specified by `pid` in accordance
 31511 with the `attr` argument. The `attr` argument represents the initial attributes of the trace stream and
 31512 shall have been initialized by the function `posix_trace_attr_init()` prior to the `posix_trace_create()`
 31513 call. If the argument `attr` is `NULL`, the default attributes shall be used. The `attr` attributes object
 31514 shall be manipulated through a set of functions described in the `posix_trace_attr` family of
 31515 functions. If the attributes of the object pointed to by `attr` are modified later, the attributes of the
 31516 trace stream shall not be affected. The *creation-time* attribute of the newly created trace stream
 31517 shall be set to the value of the system clock, if the Timers option is not supported, or to the value
 31518 of the `CLOCK_REALTIME` clock, if the Timers option is supported.

31519 The `pid` argument represents the target process to be traced. If the process executing this function
 31520 does not have appropriate privileges to trace the process identified by `pid`, an error shall be
 31521 returned. If the `pid` argument is zero, the calling process shall be traced.

31522 The `posix_trace_create()` function shall store the trace stream identifier of the new trace stream in
 31523 the object pointed to by the `trid` argument. This trace stream identifier shall be used in
 31524 subsequent calls to control tracing. The `trid` argument may only be used by the following
 31525 functions:

31526	<code>posix_trace_clear()</code>	<code>posix_trace_getnext_event()</code>
31527	<code>posix_trace_eventid_equal()</code>	<code>posix_trace_shutdown()</code>
31528	<code>posix_trace_eventid_get_name()</code>	<code>posix_trace_start()</code>
31529	<code>posix_trace_eventtypelist_getnext_id()</code>	<code>posix_trace_stop()</code>
31530	<code>posix_trace_eventtypelist_rewind()</code>	<code>posix_trace_timedgetnext_event()</code>
31531	<code>posix_trace_get_attr()</code>	<code>posix_trace_trid_eventid_open()</code>
31532	<code>posix_trace_get_status()</code>	<code>posix_trace_trygetnext_event()</code>

31533 TEF If the Trace Event Filter option is supported, the following additional functions may use the `trid`
 31534 argument:

```
31535 posix_trace_get_filter()  posix_trace_set_filter()
```

31536 In particular, notice that the operations normally used by a trace analyzer process, such as
 31537 `posix_trace_rewind()` or `posix_trace_close()`, cannot be invoked using the trace stream identifier
 31538 returned by the `posix_trace_create()` function.

posix_trace_create()

System Interfaces

TEF A trace stream shall be created in a suspended state. If the Trace Event Filter option is supported, its trace event type filter shall be empty.

The *posix_trace_create()* function may be called multiple times from the same or different processes, with the system-wide limit indicated by the runtime invariant value {TRACE_SYS_MAX}, which has the minimum value {_POSIX_TRACE_SYS_MAX}.

The trace stream identifier returned by the *posix_trace_create()* function in the argument pointed to by *trid* is valid only in the process that made the function call. If it is used from another process, that is a child process, in functions defined in IEEE Std 1003.1-200x, these functions shall return with the error [EINVAL].

TRL The *posix_trace_create_withlog()* function shall be equivalent to *posix_trace_create()*, except that it associates a trace log with this stream. The *file_desc* argument shall be the file descriptor designating the trace log destination. The function shall fail if this file descriptor refers to a file with a file type that is not compatible with the log policy associated with the trace log. The list of the appropriate file types that are compatible with each log policy is implementation-defined.

The *posix_trace_create_withlog()* function shall return in the parameter pointed to by *trid* the trace stream identifier, which uniquely identifies the newly created trace stream, and shall be used in subsequent calls to control tracing. The *trid* argument may only be used by the following functions:

<i>posix_trace_clear()</i>	<i>posix_trace_get_status()</i>
<i>posix_trace_eventid_equal()</i>	<i>posix_trace_getnext_event()</i>
<i>posix_trace_eventid_get_name()</i>	<i>posix_trace_shutdown()</i>
<i>posix_trace_eventtypelist_getnext_id()</i>	<i>posix_trace_start()</i>
<i>posix_trace_eventtypelist_rewind()</i>	<i>posix_trace_stop()</i>
<i>posix_trace_flush()</i>	<i>posix_trace_timedgetnext_event()</i>
<i>posix_trace_get_attr()</i>	<i>posix_trace_trid_eventid_open()</i>

TEF TRL If the Trace Event Filter option is supported, the following additional functions may use the *trid* argument:

posix_trace_get_filter() *posix_trace_set_filter()*

TRL In particular, notice that the operations normally used by a trace analyzer process, such as *posix_trace_rewind()* or *posix_trace_close()*, cannot be invoked using the trace stream identifier returned by the *posix_trace_create_withlog()* function.

The *posix_trace_flush()* function shall initiate a flush operation which copies the contents of the trace stream identified by the argument *trid* into the trace log associated with the trace stream at the creation time. If no trace log has been associated with the trace stream pointed to by *trid*, this function shall return an error. The termination of the flush operation can be polled by the *posix_trace_get_status()* function. During the flush operation, it shall be possible to trace new trace events up to the point when the trace stream becomes full. After flushing is completed, the

31584 31585		POSIX_TRACE_APPEND The trace events that have not yet been flushed shall be appended to the trace log.
31586 31587 31588		The <i>posix_trace_shutdown()</i> function shall stop the tracing of trace events in the trace stream identified by <i>trid</i> , as if <i>posix_trace_stop()</i> had been invoked. The <i>posix_trace_shutdown()</i> function shall free all the resources associated with the trace stream.
31589 31590 31591 31592 31593 31594		The <i>posix_trace_shutdown()</i> function shall not return until all the resources associated with the trace stream have been freed. When the <i>posix_trace_shutdown()</i> function returns, the <i>trid</i> argument becomes an invalid trace stream identifier. A call to this function shall unconditionally deallocate the resources regardless of whether all trace events have been retrieved by the analyzer process. Any thread blocked on one of the <i>trace_getnext_event()</i> functions (which specified this <i>trid</i>) before this call is unblocked with the error [EINVAL].
31595 31596 31597		If the process exits, invokes a member of the <i>exec</i> family of functions, or is terminated, the trace streams that the process had created and that have not yet been shut down, shall be automatically shut down as if an explicit call were made to the <i>posix_trace_shutdown()</i> function.
31598 31599 31600	TRL	For an active trace stream with log, when the <i>posix_trace_shutdown()</i> function is called, all trace events that have not yet been flushed to the trace log shall be flushed, as in the <i>posix_trace_flush()</i> function, and the trace log shall be closed.
31601 31602 31603 31604		When a trace log is closed, all the information that may be retrieved later from the trace log through the trace interface shall have been written to the trace log. This information includes the trace attributes, the list of trace event types (with the mapping between trace event names and trace event type identifiers), and the trace status.
31605 31606		In addition, unspecified information shall be written to the trace log to allow detection of a valid trace log during the <i>posix_trace_open()</i> operation.
31607		The <i>posix_trace_shutdown()</i> function shall not return until all trace events have been flushed.
31608		RETURN VALUE
31609 31610		Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.
31611 31612	TRL	The <i>posix_trace_create()</i> and <i>posix_trace_create_withlog()</i> functions store the trace stream identifier value in the object pointed to by <i>trid</i> , if successful.
31613		ERRORS
31614	TRL	The <i>posix_trace_create()</i> and <i>posix_trace_create_withlog()</i> functions shall fail if:
31615 31616		[EAGAIN] No more trace streams can be started now. {TRACE_SYS_MAX} has been exceeded.
31617		[EINTR] The operation was interrupted by a signal. No trace stream was created.
31618		[EINVAL] One or more of the trace parameters specified by the <i>attr</i> parameter is invalid.
31619 31620		[ENOMEM] The implementation does not currently have sufficient memory to create the trace stream with the specified parameters.
31621 31622		[EPERM] The caller does not have appropriate privilege to trace the process specified by <i>pid</i> .
31623		[ESRCH] The <i>pid</i> argument does not refer to an existing process.
31624	TRL	The <i>posix_trace_create_withlog()</i> function shall fail if:
31625		[EBADF] The <i>file_desc</i> argument is not a valid file descriptor open for writing.

posix_trace_create()

System Interfaces

31626	[EINVAL]	The <i>file_desc</i> argument refers to a file with a file type that does not support the log policy associated with the trace log.
31627		
31628	[ENOSPC]	No space left on device. The device corresponding to the argument <i>file_desc</i> does not contain the space required to create this trace log.
31629		

31630 TRL The *posix_trace_flush()* and *posix_trace_shutdown()* functions shall fail if:

31631	[EINVAL]	The value of the <i>trid</i> argument does not correspond to an active trace stream with log.
31632		
31633	[EFBIG]	The trace log file has attempted to exceed an implementation-defined maximum file size.
31634		
31635	[ENOSPC]	No space left on device.

EXAMPLES

31636 None.

APPLICATION USAGE

31637 None.

RATIONALE

31638 None.

FUTURE DIRECTIONS

31642 The *posix_trace_create()*, *posix_trace_create_withlog()*, *posix_trace_flush()*, and *posix_trace_shutdown()* functions may be withdrawn in a future version.

SEE ALSO

31643 *clock_getres()*, *exec*, *posix_trace_attr_init()*, *posix_trace_clear()*, *posix_trace_close()*,
 31644 *posix_trace_eventid_equal()*, *posix_trace_eventtypelist_getnext_id()*, *posix_trace_flush()*,
 31645 *posix_trace_get_attr()*, *posix_trace_get_filter()*, *posix_trace_get_status()*, *posix_trace_getnext_event()*,
 31646 *posix_trace_open()*, *posix_trace_set_filter()*, *posix_trace_shutdown()*, *posix_trace_start()*,
 31647 *posix_trace_timedgetnext_event()*, *posix_trace_trid_eventid_open()*, *posix_trace_start()*, *time()*, the
 31648 Base Definitions volume of IEEE Std 1003.1-200x, <sys/types.h>, <trace.h>

CHANGE HISTORY

31649 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

Issue 7

31650 These functions are marked obsolescent.

31651 SD5-XSH-ERN-154 is applied, updating the DESCRIPTION to remove the
 31652 *posix_trace_trygetnext_event()* function from the list of functions that use the *trid* argument.

31658 **NAME**

31659 `posix_trace_event`, `posix_trace_eventid_open` — trace functions for instrumenting application
 31660 code (**TRACING**)

31661 **SYNOPSIS**

```
31662 OB TRC #include <sys/types.h>
31663 #include <trace.h>
31664
31664 void posix_trace_event(trace_event_id_t event_id,
31665     const void *restrictdata_ptr, size_t data_len);
31666 int posix_trace_eventid_open(const char *restrict event_name,
31667     trace_event_id_t *restrict event_id);
```

31668 **DESCRIPTION**

31669 The `posix_trace_event()` function shall record the `event_id` and the user data pointed to by `data_ptr`
 31670 in the trace stream into which the calling process is being traced and in which `event_id` is not
 31671 filtered out. If the total size of the user trace event data represented by `data_len` is not greater
 31672 than the declared maximum size for user trace event data, then the `truncation-status` attribute of
 31673 the trace event recorded is `POSIX_TRACE_NOT_TRUNCATED`. Otherwise, the user trace event
 31674 data is truncated to this declared maximum size and the `truncation-status` attribute of the trace
 31675 event recorded is `POSIX_TRACE_TRUNCATED_RECORD`.

31676 If there is no trace stream created for the process or if the created trace stream is not running, or
 31677 if the trace event specified by `event_id` is filtered out in the trace stream, the `posix_trace_event()`
 31678 function shall have no effect.

31679 The `posix_trace_eventid_open()` function shall associate a user trace event name with a trace event
 31680 type identifier for the calling process. The trace event name is the string pointed to by the
 31681 argument `event_name`. It shall have a maximum of `{TRACE_EVENT_NAME_MAX}` characters
 31682 (which has the minimum value `{_POSIX_TRACE_EVENT_NAME_MAX}`). The number of user
 31683 trace event type identifiers that can be defined for any given process is limited by the maximum
 31684 value `{TRACE_USER_EVENT_MAX}`, which has the minimum value
 31685 `{POSIX_TRACE_USER_EVENT_MAX}`.

31686 If the Trace Inherit option is not supported, the `posix_trace_eventid_open()` function shall associate
 31687 the user trace event name pointed to by the `event_name` argument with a trace event type
 31688 identifier that is unique for the traced process, and is returned in the variable pointed to by the
 31689 `event_id` argument. If the user trace event name has already been mapped for the traced process,
 31690 then the previously assigned trace event type identifier shall be returned. If the per-process user
 31691 trace event name limit represented by `{TRACE_USER_EVENT_MAX}` has been reached, the pre-
 31692 defined `POSIX_TRACE_UNNAMED_USEREVENT` (see [Table 2-7](#) (on page 81)) user trace event
 31693 shall be returned.

31694 TRI If the Trace Inherit option is supported, the `posix_trace_eventid_open()` function shall associate the
 31695 user trace event name pointed to by the `event_name` argument with a trace event type identifier
 31696 that is unique for all the processes being traced in this same trace stream, and is returned in the
 31697 variable pointed to by the `event_id` argument. If the user trace event name has already been
 31698 mapped for the traced processes, then the previously assigned trace event type identifier shall be
 31699 returned. If the per-process user trace event name limit represented by
 31700 `{TRACE_USER_EVENT_MAX}` has been reached, the pre-defined
 31701 `POSIX_TRACE_UNNAMED_USEREVENT` ([Table 2-7](#) (on page 81)) user trace event shall be
 31702 returned.

Note: The above procedure, together with the fact that multiple processes can only be traced into the same trace stream by inheritance, ensure that all the processes that are traced into a trace stream have the same mapping of trace event names to trace event type identifiers.

31703
31704

31705 If there is no trace stream created, the *posix_trace_eventid_open()* function shall store this
31706 information for future trace streams created for this process.

RETURN VALUE

31707 No return value is defined for the *posix_trace_event()* function.
31708

31709 Upon successful completion, the *posix_trace_eventid_open()* function shall return a value of zero.
31710 Otherwise, it shall return the corresponding error number. The *posix_trace_eventid_open()*
31711 function stores the trace event type identifier value in the object pointed to by *event_id*, if
31712 successful.

ERRORS

31713 The *posix_trace_eventid_open()* function shall fail if:
31714

[ENAMETOOLONG]

31715 The size of the name pointed to by the *event_name* argument was longer than
31716 the implementation-defined value {TRACE_EVENT_NAME_MAX}.
31717

EXAMPLES

31718 None.
31719

APPLICATION USAGE

31720 None.
31721

RATIONALE

31722 None.
31723

FUTURE DIRECTIONS

31724 The *posix_trace_event()* and *posix_trace_eventid_open()* functions may be withdrawn in a future
31725 version.
31726

SEE ALSO

31727 [Table 2-7](#) (on page 81), *exec*, *posix_trace_start()*, *posix_trace_trid_eventid_open()*, the Base
31728 Definitions volume of IEEE Std 1003.1-200x, [<sys/types.h>](#), [<trace.h>](#)
31729

CHANGE HISTORY

31730 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.
31731

31732 IEEE PASC Interpretation 1003.1 #123 is applied.

31733 IEEE PASC Interpretation 1003.1 #127 is applied, correcting some editorial errors in the names of
31734 the *posix_trace_eventid_open()* function and the *event_id* argument.

Issue 7

31735 The *posix_trace_event()* and *posix_trace_eventid_open()* functions are marked obsolescent.
31736

31737 **NAME**

31738 posix_trace_eventid_equal, posix_trace_eventid_get_name, posix_trace_trid_eventid_open —
 31739 manipulate the trace event type identifier (**TRACING**)

31740 **SYNOPSIS**

```
31741 OB TRC #include <trace.h>
31742
31743 int posix_trace_eventid_equal(trace_id_t trid, trace_event_id_t event1,
31744                             trace_event_id_t event2);
31745 int posix_trace_eventid_get_name(trace_id_t trid,
31746                                 trace_event_id_t event, char *event_name);
31747 TEF int posix_trace_trid_eventid_open(trace_id_t trid,
31748                                     const char *restrict event_name,
31749                                     trace_event_id_t *restrict event);
```

31749 **DESCRIPTION**

31750 The *posix_trace_eventid_equal()* function shall compare the trace event type identifiers *event1* and
 31751 *event2* from the same trace stream or the same trace log identified by the *trid* argument. If the
 31752 trace event type identifiers *event1* and *event2* are from different trace streams, the return value
 31753 shall be unspecified.

31754 The *posix_trace_eventid_get_name()* function shall return, in the argument pointed to by
 31755 *event_name*, the trace event name associated with the trace event type identifier identified by the
 31756 argument *event*, for the trace stream or for the trace log identified by the *trid* argument. The
 31757 name of the trace event shall have a maximum of {TRACE_EVENT_NAME_MAX} characters
 31758 (which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). Successive calls to
 31759 this function with the same trace event type identifier and the same trace stream identifier shall
 31760 return the same event name.

31761 TEF The *posix_trace_trid_eventid_open()* function shall associate a user trace event name with a trace
 31762 event type identifier for a given trace stream. The trace stream is identified by the *trid* argument,
 31763 and it shall be an active trace stream. The trace event name is the string pointed to by the
 31764 argument *event_name*. It shall have a maximum of {TRACE_EVENT_NAME_MAX} characters
 31765 (which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). The number of user
 31766 trace event type identifiers that can be defined for any given process is limited by the maximum
 31767 value {TRACE_USER_EVENT_MAX}, which has the minimum value
 31768 {_POSIX_TRACE_USER_EVENT_MAX}.

31769 If the Trace Inherit option is not supported, the *posix_trace_trid_eventid_open()* function shall
 31770 associate the user trace event name pointed to by the *event_name* argument with a trace event
 31771 type identifier that is unique for the process being traced in the trace stream identified by the *trid*
 31772 argument, and is returned in the variable pointed to by the *event* argument. If the user trace
 31773 event name has already been mapped for the traced process, then the previously assigned trace
 31774 event type identifier shall be returned. If the per-process user trace event name limit represented
 31775 by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined
 31776 POSIX_TRACE_UNNAMED_USEREVENT (see [Table 2-7](#) (on page 81)) user trace event shall be
 31777 returned.

31778 TEF TRI If the Trace Inherit option is supported, the *posix_trace_trid_eventid_open()* function shall
 31779 associate the user trace event name pointed to by the *event_name* argument with a trace event
 31780 type identifier that is unique for all the processes being traced in the trace stream identified by
 31781 the *trid* argument, and is returned in the variable pointed to by the *event* argument. If the user
 31782 trace event name has already been mapped for the traced processes, then the previously
 31783 assigned trace event type identifier shall be returned. If the per-process user trace event name

posix_trace_eventid_equal()

System Interfaces

31784 limit represented by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined
 31785 POSIX_TRACE_UNNAMED_USEREVENT (see Table 2-7 (on page 81)) user trace event shall be
 31786 returned.

RETURN VALUE

31787
 31788 TEF Upon successful completion, the *posix_trace_eventid_get_name()* and
 31789 *posix_trace_trid_eventid_open()* functions shall return a value of zero. Otherwise, they shall return
 31790 the corresponding error number.

31791 The *posix_trace_eventid_equal()* function shall return a non-zero value if *event1* and *event2* are
 31792 equal; otherwise, a value of zero shall be returned. No errors are defined. If either *event1* or
 31793 *event2* are not valid trace event type identifiers for the trace stream specified by *trid* or if the *trid*
 31794 is invalid, the behavior shall be unspecified.

31795 The *posix_trace_eventid_get_name()* function stores the trace event name value in the object
 31796 pointed to by *event_name*, if successful.

31797 TEF The *posix_trace_trid_eventid_open()* function stores the trace event type identifier value in the
 31798 object pointed to by *event*, if successful.

ERRORS

31799
 31800 TEF The *posix_trace_eventid_get_name()* and *posix_trace_trid_eventid_open()* functions shall fail if:

31801 [EINVAL] The *trid* argument was not a valid trace stream identifier.

31802 TEF The *posix_trace_trid_eventid_open()* function shall fail if:

31803 TEF [ENAMETOOLONG]

31804 The size of the name pointed to by the *event_name* argument was longer than
 31805 the implementation-defined value {TRACE_EVENT_NAME_MAX}.

31806 The *posix_trace_eventid_get_name()* function shall fail if:

31807 [EINVAL] The trace event type identifier *event* was not associated with any name.

EXAMPLES

31808 None.
 31809

APPLICATION USAGE

31810 None.
 31811

RATIONALE

31812 None.
 31813

FUTURE DIRECTIONS

31814 The *posix_trace_eventid_equal()*, *posix_trace_eventid_get_name()*, and
 31815 *posix_trace_trid_eventid_open()* functions may be withdrawn in a future version.
 31816

SEE ALSO

31817 Table 2-7 (on page 81), *exec*, *posix_trace_event()*, *posix_trace_getnext_event()*, the Base Definitions
 31818 volume of IEEE Std 1003.1-200x, <trace.h>
 31819

CHANGE HISTORY

31820 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.
 31821

31822 IEEE PASC Interpretations 1003.1 #123 and #129 are applied.

Issue 7

31823 These functions are marked obsolescent.
 31824

31825 **NAME**
31826 `posix_trace_eventid_open` — trace functions for instrumenting application code (**TRACING**)

31827 **SYNOPSIS**

```
31828 OB TRC #include <sys/types.h>  
31829 #include <trace.h>  
  
31830 int posix_trace_eventid_open(const char *restrict event_name,  
31831 trace_event_id_t *restrict event_id);
```

31832 **DESCRIPTION**

31833 Refer to [posix_trace_event\(\)](#).

31834 **NAME**

31835 `posix_trace_eventset_add`, `posix_trace_eventset_del`, `posix_trace_eventset_empty`,
 31836 `posix_trace_eventset_fill`, `posix_trace_eventset_ismember` — manipulate trace event type sets
 31837 (TRACING)

31838 **SYNOPSIS**

```
31839 OB TRC #include <trace.h>
31840 TEF int posix_trace_eventset_add(trace_event_id_t event_id,
31841 trace_event_set_t *set);
31842 int posix_trace_eventset_del(trace_event_id_t event_id,
31843 trace_event_set_t *set);
31844 int posix_trace_eventset_empty(trace_event_set_t *set);
31845 int posix_trace_eventset_fill(trace_event_set_t *set, int what);
31846 int posix_trace_eventset_ismember(trace_event_id_t event_id,
31847 const trace_event_set_t *restrict set, int *restrict ismember);
```

31848 **DESCRIPTION**

31849 These primitives manipulate sets of trace event types. They operate on data objects addressable
 31850 by the application, not on the current trace event filter of any trace stream.

31851 The `posix_trace_eventset_add()` and `posix_trace_eventset_del()` functions, respectively, shall add or
 31852 delete the individual trace event type specified by the value of the argument `event_id` to or from
 31853 the trace event type set pointed to by the argument `set`. Adding a trace event type already in the
 31854 set or deleting a trace event type not in the set shall not be considered an error.

31855 The `posix_trace_eventset_empty()` function shall initialize the trace event type set pointed to by
 31856 the `set` argument such that all trace event types defined, both system and user, shall be excluded
 31857 from the set.

31858 The `posix_trace_eventset_fill()` function shall initialize the trace event type set pointed to by the
 31859 argument `set`, such that the set of trace event types defined by the argument `what` shall be
 31860 included in the set. The value of the argument `what` shall consist of one of the following values,
 31861 as defined in the `<trace.h>` header:

31862 **POSIX_TRACE_WOPID_EVENTS**

31863 All the process-independent implementation-defined system trace event types are included
 31864 in the set.

31865 **POSIX_TRACE_SYSTEM_EVENTS**

31866 All the implementation-defined system trace event types are included in the set, as are those
 31867 defined in IEEE Std 1003.1-200x.

31868 **POSIX_TRACE_ALL_EVENTS**

31869 All trace event types defined, both system and user, are included in the set.

31870 Applications shall call either `posix_trace_eventset_empty()` or `posix_trace_eventset_fill()` at least
 31871 once for each object of type `trace_event_set_t` prior to any other use of that object. If such an
 31872 object is not initialized in this way, but is nonetheless supplied as an argument to any of the
 31873 `posix_trace_eventset_add()`, `posix_trace_eventset_del()`, or `posix_trace_eventset_ismember()` functions,
 31874 the results are undefined.

31875 The `posix_trace_eventset_ismember()` function shall test whether the trace event type specified by
 31876 the value of the argument `event_id` is a member of the set pointed to by the argument `set`. The
 31877 value returned in the object pointed to by `ismember` argument is zero if the trace event type
 31878 identifier is not a member of the set and a value different from zero if it is a member of the set.

31879 RETURN VALUE

31880 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 31881 return the corresponding error number.

31882 ERRORS

31883 These functions may fail if:

31884 [EINVAL] The value of one of the arguments is invalid.

31885 EXAMPLES

31886 None.

31887 APPLICATION USAGE

31888 None.

31889 RATIONALE

31890 None.

31891 FUTURE DIRECTIONS

31892 The *posix_trace_eventset_add()*, *posix_trace_eventset_del()*, *posix_trace_eventset_empty()*,
 31893 *posix_trace_eventset_fill()*, and *posix_trace_eventset_ismember()* functions may be removed in a
 31894 future version.

31895 SEE ALSO

31896 *posix_trace_set_filter()*, *posix_trace_trid_eventid_open()*, the Base Definitions volume of
 31897 IEEE Std 1003.1-200x, <trace.h>

31898 CHANGE HISTORY

31899 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

31900 Issue 7

31901 The *posix_trace_eventset_add()*, *posix_trace_eventset_del()*, *posix_trace_eventset_empty()*,
 31902 *posix_trace_eventset_fill()*, and *posix_trace_eventset_ismember()* functions are marked obsolescent.

31903 **NAME**

31904 `posix_trace_eventtypelist_getnext_id`, `posix_trace_eventtypelist_rewind` — iterate over a
 31905 mapping of trace event types (**TRACING**)

31906 **SYNOPSIS**

```
31907 OB TRC #include <trace.h>
31908
31908 int posix_trace_eventtypelist_getnext_id(trace_id_t trid,
31909     trace_event_id_t *restrict event, int *restrict unavailable);
31910 int posix_trace_eventtypelist_rewind(trace_id_t trid);
```

31911 **DESCRIPTION**

31912 The first time `posix_trace_eventtypelist_getnext_id()` is called, the function shall return in the
 31913 variable pointed to by `event` the first trace event type identifier of the list of trace events of the
 31914 trace stream identified by the `trid` argument. Successive calls to
 31915 `posix_trace_eventtypelist_getnext_id()` return in the variable pointed to by `event` the next trace
 31916 event type identifier in that same list. Each time a trace event type identifier is successfully
 31917 written into the variable pointed to by the `event` argument, the variable pointed to by the
 31918 `unavailable` argument shall be set to zero. When no more trace event type identifiers are available,
 31919 and so none is returned, the variable pointed to by the `unavailable` argument shall be set to a
 31920 value different from zero.

31921 The `posix_trace_eventtypelist_rewind()` function shall reset the next trace event type identifier to
 31922 be read to the first trace event type identifier from the list of trace events used in the trace stream
 31923 identified by `trid`.

31924 **RETURN VALUE**

31925 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 31926 return the corresponding error number.

31927 The `posix_trace_eventtypelist_getnext_id()` function stores the trace event type identifier value in
 31928 the object pointed to by `event`, if successful.

31929 **ERRORS**

31930 These functions shall fail if:

31931 [EINVAL] The `trid` argument was not a valid trace stream identifier.

31932 **EXAMPLES**

31933 None.

31934 **APPLICATION USAGE**

31935 None.

31936 **RATIONALE**

31937 None.

31938 **FUTURE DIRECTIONS**

31939 The `posix_trace_eventtypelist_getnext_id()` and `posix_trace_eventtypelist_rewind()` functions may be
 31940 removed in a future version.

31941 **SEE ALSO**

31942 [posix_trace_event\(\)](#), [posix_trace_getnext_event\(\)](#), [posix_trace_trid_eventid_open\(\)](#), the Base
 31943 Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

31944

CHANGE HISTORY

31945

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

31946

IEEE PASC Interpretations 1003.1 #123 and #129 are applied.

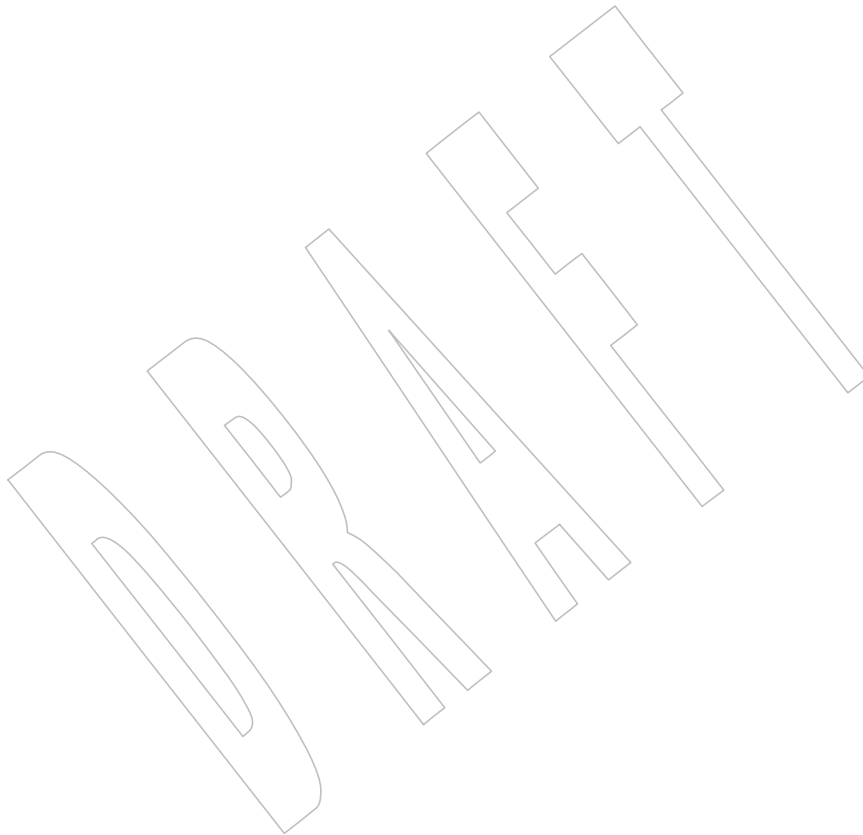
31947

Issue 7

31948

The *posix_trace_eventtypelist_getnext_id()* and *posix_trace_eventtypelist_rewind()* functions are marked obsolescent.

31949



posix_trace_flush()

31950 **NAME**
31951 `posix_trace_flush` — trace stream flush from a process (**TRACING**)

SYNOPSIS

```
31953 OB TRC #include <sys/types.h>  
31954 #include <trace.h>  
31955 TRL int posix_trace_flush(trace_id_t trid);
```

DESCRIPTION

31956 Refer to [posix_trace_create\(\)](#).
31957

31958 **NAME**

31959 `posix_trace_get_attr`, `posix_trace_get_status` — retrieve the trace attributes or trace status
 31960 (**TRACING**)

31961 **SYNOPSIS**

```
31962 OB TRC #include <trace.h>
31963
31963 int posix_trace_get_attr(trace_id_t trid, trace_attr_t *attr);
31964 int posix_trace_get_status(trace_id_t trid,
31965 struct posix_trace_status_info *statusinfo);
```

31966 **DESCRIPTION**

31967 The `posix_trace_get_attr()` function shall copy the attributes of the active trace stream identified
 31968 TRL by `trid` into the object pointed to by the `attr` argument. If the Trace Log option is supported, `trid`
 31969 may represent a pre-recorded trace log.

31970 The `posix_trace_get_status()` function shall return, in the structure pointed to by the `statusinfo`
 31971 argument, the current trace status for the trace stream identified by the `trid` argument. These
 31972 status values returned in the structure pointed to by `statusinfo` shall have been appropriately
 31973 TRL read to ensure that the returned values are consistent. If the Trace Log option is supported and
 31974 the `trid` argument refers to a pre-recorded trace stream, the status shall be the status of the
 31975 completed trace stream.

31976 Each time the `posix_trace_get_status()` function is used, the overrun status of the trace stream
 31977 TRL shall be reset to `POSIX_TRACE_NO_OVERRUN` immediately after the call completes. If the
 31978 Trace Log option is supported, the `posix_trace_get_status()` function shall behave the same as
 31979 when the option is not supported except for the following differences:

- 31980 • If the `trid` argument refers to a trace stream with log, each time the `posix_trace_get_status()`
 31981 function is used, the log overrun status of the trace stream shall be reset to
 31982 `POSIX_TRACE_NO_OVERRUN` and the `flush_error` status shall be reset to zero
 31983 immediately after the call completes.
- 31984 • If the `trid` argument refers to a pre-recorded trace stream, the status returned shall be the
 31985 status of the completed trace stream and the status values of the trace stream shall not be
 31986 reset.

31987 **RETURN VALUE**

31988 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 31989 return the corresponding error number.

31990 The `posix_trace_get_attr()` function stores the trace attributes in the object pointed to by `attr`, if
 31991 successful.

31992 The `posix_trace_get_status()` function stores the trace status in the object pointed to by `statusinfo`,
 31993 if successful.

31994 **ERRORS**

31995 These functions shall fail if:

- 31996 [EINVAL] The trace stream argument `trid` does not correspond to a valid active trace
 31997 stream or a valid trace log.

posix_trace_get_attr()

31998

EXAMPLES

31999

None.

32000

APPLICATION USAGE

32001

None.

32002

RATIONALE

32003

None.

32004

FUTURE DIRECTIONS

32005

The *posix_trace_get_attr()* and *posix_trace_get_status()* functions may be withdrawn in a future version.

32006

32007

SEE ALSO

32008

posix_trace_attr_destroy(), *posix_trace_attr_init()*, *posix_trace_create()*, *posix_trace_open()*, the Base Definitions volume of IEEE Std 1003.1-200x, <trace.h>

32009

32010

CHANGE HISTORY

32011

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

32012

IEEE PASC Interpretation 1003.1 #123 is applied.

32013

Issue 7

32014

The *posix_trace_get_attr()* and *posix_trace_get_status()* functions are marked obsolescent.

DRAFT

32015 **NAME**

32016 `posix_trace_get_filter`, `posix_trace_set_filter` — retrieve and set the filter of an initialized trace
 32017 stream (**TRACING**)

32018 **SYNOPSIS**

```
32019 OB TRC #include <trace.h>
32020 TEF int posix_trace_get_filter(trace_id_t trid, trace_event_set_t *set);
32021 int posix_trace_set_filter(trace_id_t trid,
32022 const trace_event_set_t *set, int how);
```

32023 **DESCRIPTION**

32024 The `posix_trace_get_filter()` function shall retrieve, into the argument pointed to by `set`, the actual
 32025 trace event filter from the trace stream specified by `trid`.

32026 The `posix_trace_set_filter()` function shall change the set of filtered trace event types after a trace
 32027 stream identified by the `trid` argument is created. This function may be called prior to starting
 32028 the trace stream, or while the trace stream is active. By default, if no call is made to
 32029 `posix_trace_set_filter()`, all trace events shall be recorded (that is, none of the trace event types are
 32030 filtered out).

32031 If this function is called while the trace is in progress, a special system trace event,
 32032 `POSIX_TRACE_FILTER`, shall be recorded in the trace indicating both the old and the new sets
 32033 of filtered trace event types (see [Table 2-4](#) and [Table 2-6](#) (on page 80)).

32034 If the `posix_trace_set_filter()` function is interrupted by a signal, an error shall be returned and the
 32035 filter shall not be changed. In this case, the state of the trace stream shall not be changed.

32036 The value of the argument `how` indicates the manner in which the set is to be changed and shall
 32037 have one of the following values, as defined in the `<trace.h>` header:

32038 `POSIX_TRACE_SET_EVENTSET`

32039 The resulting set of trace event types to be filtered shall be the trace event type set pointed
 32040 to by the argument `set`.

32041 `POSIX_TRACE_ADD_EVENTSET`

32042 The resulting set of trace event types to be filtered shall be the union of the current set and
 32043 the trace event type set pointed to by the argument `set`.

32044 `POSIX_TRACE_SUB_EVENTSET`

32045 The resulting set of trace event types to be filtered shall be all trace event types in the
 32046 current set that are not in the set pointed to by the argument `set`; that is, remove each
 32047 element of the specified set from the current filter.

32048 **RETURN VALUE**

32049 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 32050 return the corresponding error number.

32051 The `posix_trace_get_filter()` function stores the set of filtered trace event types in `set`, if successful.

32052 **ERRORS**

32053 These functions shall fail if:

32054 `[EINVAL]` The value of the `trid` argument does not correspond to an active trace stream
 32055 or the value of the argument pointed to by `set` is invalid.

32056 [EINTR] The operation was interrupted by a signal.

EXAMPLES

32057 None.
32058

APPLICATION USAGE

32059 None.
32060

RATIONALE

32061 None.
32062

FUTURE DIRECTIONS

32063 The *posix_trace_get_filter()* and *posix_trace_set_filter()* functions may be removed in a future
32064 version.
32065

SEE ALSO

32066 [Table 2-4](#) (on page 79), [Table 2-6](#) (on page 80), [posix_trace_eventset_add\(\)](#), the Base Definitions
32067 volume of IEEE Std 1003.1-200x, **<trace.h>**
32068

CHANGE HISTORY

32069 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.
32070

32071 IEEE PASC Interpretation 1003.1 #123 is applied.

Issue 7

32072 The *posix_trace_get_filter()* and *posix_trace_set_filter()* functions are marked obsolescent.
32073

32074 **NAME**
32075 `posix_trace_get_status` — retrieve the trace status (**TRACING**)

32076 **SYNOPSIS**

32077 OB TRC `#include <trace.h>`
32078 `int posix_trace_get_status(trace_id_t trid,`
32079 `struct posix_trace_status_info *statusinfo);`

32080 **DESCRIPTION**

32081 Refer to [posix_trace_get_attr\(\)](#).

32082 **NAME**

32083 `posix_trace_getnext_event`, `posix_trace_timedgetnext_event`, `posix_trace_trygetnext_event` —
 32084 retrieve a trace event (**TRACING**)

32085 **SYNOPSIS**

```
32086 OB TRC #include <sys/types.h>
32087 #include <trace.h>

32088 int posix_trace_getnext_event(trace_id_t trid,
32089     struct posix_trace_event_info *restrict event,
32090     void *restrict data, size_t num_bytes,
32091     size_t *restrict data_len, int *restrict unavailable);
32092 int posix_trace_timedgetnext_event(trace_id_t trid,
32093     struct posix_trace_event_info *restrict event,
32094     void *restrict data, size_t num_bytes,
32095     size_t *restrict data_len, int *restrict unavailable,
32096     const struct timespec *restrict abs_timeout);
32097 int posix_trace_trygetnext_event(trace_id_t trid,
32098     struct posix_trace_event_info *restrict event,
32099     void *restrict data, size_t num_bytes,
32100     size_t *restrict data_len, int *restrict unavailable);
```

32101 **DESCRIPTION**

32102 The `posix_trace_getnext_event()` function shall report a recorded trace event either from an active
 32103 **TRL** trace stream without log or a pre-recorded trace stream identified by the `trid` argument. The
 32104 `posix_trace_trygetnext_event()` function shall report a recorded trace event from an active trace
 32105 stream without log identified by the `trid` argument.

32106 The trace event information associated with the recorded trace event shall be copied by the
 32107 function into the structure pointed to by the argument `event` and the data associated with the
 32108 trace event shall be copied into the buffer pointed to by the `data` argument.

32109 The `posix_trace_getnext_event()` function shall block if the `trid` argument identifies an active trace
 32110 stream and there is currently no trace event ready to be retrieved. When returning, if a recorded
 32111 trace event was reported, the variable pointed to by the `unavailable` argument shall be set to zero.
 32112 Otherwise, the variable pointed to by the `unavailable` argument shall be set to a value different
 32113 from zero.

32114 The `posix_trace_timedgetnext_event()` function shall attempt to get another trace event from an
 32115 active trace stream without log, as in the `posix_trace_getnext_event()` function. However, if no
 32116 trace event is available from the trace stream, the implied wait shall be terminated when the
 32117 timeout specified by the argument `abs_timeout` expires, and the function shall return the error
 32118 [ETIMEDOUT].

32119 The timeout shall expire when the absolute time specified by `abs_timeout` passes, as measured by
 32120 the clock upon which timeouts are based (that is, when the value of that clock equals or exceeds
 32121 `abs_timeout`), or if the absolute time specified by `abs_timeout` has already passed at the time of the
 32122 call.

32123 The timeout shall be based on the `CLOCK_REALTIME` clock. The resolution of the timeout shall
 32124 be the resolution of the clock on which it is based. The `timespec` data type is defined in the
 32125 `<time.h>` header.

32126 Under no circumstance shall the function fail with a timeout if a trace event is immediately
 32127 available from the trace stream. The validity of the `abs_timeout` argument need not be checked if

32128 a trace event is immediately available from the trace stream.

32129 The behavior of this function for a pre-recorded trace stream is unspecified.

32130 TRL The *posix_trace_trygetnext_event()* function shall not block. This function shall return an error if
 32131 the *trid* argument identifies a pre-recorded trace stream. If a recorded trace event was reported,
 32132 the variable pointed to by the *unavailable* argument shall be set to zero. Otherwise, if no trace
 32133 event was reported, the variable pointed to by the *unavailable* argument shall be set to a value
 32134 different from zero.

32135 The argument *num_bytes* shall be the size of the buffer pointed to by the *data* argument. The
 32136 argument *data_len* reports to the application the length in bytes of the data record just
 32137 transferred. If *num_bytes* is greater than or equal to the size of the data associated with the trace
 32138 event pointed to by the *event* argument, all the recorded data shall be transferred. In this case,
 32139 the *truncation-status* member of the trace event structure shall be either
 32140 POSIX_TRACE_NOT_TRUNCATED, if the trace event data was recorded without truncation
 32141 while tracing, or POSIX_TRACE_TRUNCATED_RECORD, if the trace event data was truncated
 32142 when it was recorded. If the *num_bytes* argument is less than the length of recorded trace event
 32143 data, the data transferred shall be truncated to a length of *num_bytes*, the value stored in the
 32144 variable pointed to by *data_len* shall be equal to *num_bytes*, and the *truncation-status* member of
 32145 the *event* structure argument shall be set to POSIX_TRACE_TRUNCATED_READ (see the
 32146 **posix_trace_event_info** structure defined in <trace.h>).

32147 The report of a trace event shall be sequential starting from the oldest recorded trace event. Trace
 32148 events shall be reported in the order in which they were generated, up to an implementation-
 32149 defined time resolution that causes the ordering of trace events occurring very close to each
 32150 other to be unknown. Once reported, a trace event cannot be reported again from an active trace
 32151 stream. Once a trace event is reported from an active trace stream without log, the trace stream
 32152 shall make the resources associated with that trace event available to record future generated
 32153 trace events.

32154 RETURN VALUE

32155 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 32156 return the corresponding error number.

32157 If successful, these functions store:

- 32158 • The recorded trace event in the object pointed to by *event*
- 32159 • The trace event information associated with the recorded trace event in the object pointed
 32160 to by *data*
- 32161 • The length of this trace event information in the object pointed to by *data_len*
- 32162 • The value of zero in the object pointed to by *unavailable*

32163 ERRORS

32164 These functions shall fail if:

32165 [EINVAL] The trace stream identifier argument *trid* is invalid.

32166 The *posix_trace_getnext_event()* and *posix_trace_timedgetnext_event()* functions shall fail if:

32167 [EINTR] The operation was interrupted by a signal, and so the call had no effect.

32168 The *posix_trace_trygetnext_event()* function shall fail if:

32169 [EINVAL] The trace stream identifier argument *trid* does not correspond to an active
 32170 trace stream.

32171 The *posix_trace_timedgetnext_event()* function shall fail if:

posix_trace_getnext_event()

- 32172 [EINVAL] There is no trace event immediately available from the trace stream, and the
32173 *timeout* argument is invalid.
- 32174 [ETIMEDOUT] No trace event was available from the trace stream before the specified
32175 timeout *timeout* expired.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

The `posix_trace_getnext_event()`, `posix_trace_timedgetnext_event()`, and
`posix_trace_trygetnext_event()` functions may be removed in a future version.

SEE ALSO

`posix_trace_create()`, `posix_trace_open()`, the Base Definitions volume of IEEE Std 1003.1-200x,
<sys/types.h>, <trace.h>

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE PASC Interpretation 1003.1 #123 is applied.

Issue 7

The `posix_trace_getnext_event()`, `posix_trace_timedgetnext_event()`, and
`posix_trace_trygetnext_event()` functions are marked obsolescent.

Functionality relating to the Timers option is moved to the Base.

32195 **NAME**
32196 `posix_trace_open`, `posix_trace_rewind` — trace log management (**TRACING**)

32197 **SYNOPSIS**

```
32198 OB TRC #include <trace.h>  
32199 TRL     int posix_trace_open(int file_desc, trace_id_t *trid);  
32200         int posix_trace_rewind(trace_id_t trid);
```

32201 **DESCRIPTION**

32202 Refer to [*posix_trace_close\(\)*](#).

32203 **NAME**32204 `posix_trace_set_filter` — set filter of an initialized trace stream (**TRACING**)32205 **SYNOPSIS**32206 OB TRC `#include <trace.h>`32207 TEF `int posix_trace_set_filter(trace_id_t trid,`
32208 `const trace_event_set_t *set, int how);`32209 **DESCRIPTION**32210 Refer to [*posix_trace_get_filter\(\)*](#).

32211 **NAME**32212 posix_trace_shutdown — trace stream shutdown from a process (**TRACING**)32213 **SYNOPSIS**

32214 OB TRC #include <sys/types.h>

32215 #include <trace.h>

32216 int posix_trace_shutdown(trace_id_t trid);

32217 **DESCRIPTION**32218 Refer to [posix_trace_create\(\)](#).

32219 **NAME**32220 `posix_trace_start`, `posix_trace_stop` — trace start and stop (**TRACING**)32221 **SYNOPSIS**

```
32222 OB TRC #include <trace.h>
32223
32223 int posix_trace_start(trace_id_t trid);
32224 int posix_trace_stop (trace_id_t trid);
```

32225 **DESCRIPTION**

32226 The `posix_trace_start()` and `posix_trace_stop()` functions, respectively, shall start and stop the trace
 32227 stream identified by the argument `trid`.

32228 The effect of calling the `posix_trace_start()` function shall be recorded in the trace stream as the
 32229 POSIX_TRACE_START system trace event and the status of the trace stream shall become
 32230 POSIX_TRACE_RUNNING. If the trace stream is in progress when this function is called, the
 32231 POSIX_TRACE_START system trace event shall not be recorded and the trace stream shall
 32232 continue to run. If the trace stream is full, the POSIX_TRACE_START system trace event shall
 32233 not be recorded and the status of the trace stream shall not be changed.

32234 The effect of calling the `posix_trace_stop()` function shall be recorded in the trace stream as the
 32235 POSIX_TRACE_STOP system trace event and the status of the trace stream shall become
 32236 POSIX_TRACE_SUSPENDED. If the trace stream is suspended when this function is called, the
 32237 POSIX_TRACE_STOP system trace event shall not be recorded and the trace stream shall remain
 32238 suspended. If the trace stream is full, the POSIX_TRACE_STOP system trace event shall not be
 32239 recorded and the status of the trace stream shall not be changed.

32240 **RETURN VALUE**

32241 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 32242 return the corresponding error number.

32243 **ERRORS**

32244 These functions shall fail if:

32245 [EINVAL] The value of the argument `trid` does not correspond to an active trace stream
 32246 and thus no trace stream was started or stopped.

32247 [EINTR] The operation was interrupted by a signal and thus the trace stream was not
 32248 necessarily started or stopped.

32249 **EXAMPLES**

32250 None.

32251 **APPLICATION USAGE**

32252 None.

32253 **RATIONALE**

32254 None.

32255 **FUTURE DIRECTIONS**32256 The `posix_trace_start()` and `posix_trace_stop()` functions may be removed in a future version.32257 **SEE ALSO**32258 [*posix_trace_create\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<trace.h>`

32259

CHANGE HISTORY

32260

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

32261

IEEE PASC Interpretation 1003.1 #123 is applied.

32262

Issue 7

32263

The *posix_trace_start()* and *posix_trace_stop()* functions are marked obsolescent.



32264 **NAME**
 32265 `posix_trace_timedgetnext_event` — retrieve a trace event (**TRACING**)

32266 **SYNOPSIS**

```
32267 OB TRC #include <sys/types.h>
32268 #include <trace.h>

32269 int posix_trace_timedgetnext_event(trace_id_t trid,
32270 struct posix_trace_event_info *restrict event,
32271 void *restrict data, size_t num_bytes,
32272 size_t *restrict data_len, int *restrict unavailable,
32273 const struct timespec *restrict abs_timeout);
```

32274 **DESCRIPTION**

32275 Refer to [posix_trace_getnext_event\(\)](#).

32276 **NAME**32277 posix_trace_trid_eventid_open — open a trace event type identifier (**TRACING**)32278 **SYNOPSIS**32279 **OB TRC** #include <trace.h>32280 **TEF** int posix_trace_trid_eventid_open(trace_id_t trid,
32281 const char *restrict event_name,
32282 trace_event_id_t *restrict event);32283 **DESCRIPTION**32284 Refer to [posix_trace_eventid_equal\(\)](#).

posix_trace_trygetnext_event()*System Interfaces*

32285 **NAME**
32286 `posix_trace_trygetnext_event` — retrieve a trace event (**TRACING**)

SYNOPSIS

```
32288 OB TRC #include <sys/types.h>  
32289 #include <trace.h>  
  
32290 int posix_trace_trygetnext_event(trace_id_t trid,  
32291 struct posix_trace_event_info *restrict event,  
32292 void *restrict data, size_t num_bytes,  
32293 size_t *restrict data_len, int *restrict unavailable);
```

DESCRIPTION

32294 Refer to [posix_trace_getnext_event\(\)](#).
32295

32296 **NAME**32297 `posix_typed_mem_get_info` — query typed memory information (**ADVANCED REALTIME**)32298 **SYNOPSIS**

```
32299 TYM      #include <sys/mman.h>
32300
32301      int posix_typed_mem_get_info(int fildev,
          struct posix_typed_mem_info *info);
```

32302 **DESCRIPTION**

32303 The `posix_typed_mem_get_info()` function shall return, in the `posix_tmi_length` field of the
 32304 **posix_typed_mem_info** structure pointed to by `info`, the maximum length which may be
 32305 successfully allocated by the typed memory object designated by `fildev`. This maximum length
 32306 shall take into account the flag `POSIX_TYPED_MEM_ALLOCATE` or
 32307 `POSIX_TYPED_MEM_ALLOCATE_CONTIG` specified when the typed memory object
 32308 represented by `fildev` was opened. The maximum length is dynamic; therefore, the value
 32309 returned is valid only while the current mapping of the corresponding typed memory pool
 32310 remains unchanged.

32311 If `fildev` represents a typed memory object opened with neither the
 32312 `POSIX_TYPED_MEM_ALLOCATE` flag nor the `POSIX_TYPED_MEM_ALLOCATE_CONTIG`
 32313 flag specified, the returned value of `info->posix_tmi_length` is unspecified.

32314 The `posix_typed_mem_get_info()` function may return additional implementation-defined
 32315 information in other fields of the **posix_typed_mem_info** structure pointed to by `info`.

32316 If the memory object specified by `fildev` is not a typed memory object, then the behavior of this
 32317 function is undefined.

32318 **RETURN VALUE**

32319 Upon successful completion, the `posix_typed_mem_get_info()` function shall return zero;
 32320 otherwise, the corresponding error status value shall be returned.

32321 **ERRORS**

32322 The `posix_typed_mem_get_info()` function shall fail if:

32323 [EBADF] The `fildev` argument is not a valid open file descriptor.

32324 [ENODEV] The `fildev` argument is not connected to a memory object supported by this
 32325 function.

32326 This function shall not return an error code of [EINTR].

32327 **EXAMPLES**

32328 None.

32329 **APPLICATION USAGE**

32330 None.

32331 **RATIONALE**

32332 An application that needs to allocate a block of typed memory with length dependent upon the
 32333 amount of memory currently available must either query the typed memory object to obtain the
 32334 amount available, or repeatedly invoke `mmap()` attempting to guess an appropriate length.
 32335 While the latter method is existing practice with `malloc()`, it is awkward and imprecise. The
 32336 `posix_typed_mem_get_info()` function allows an application to immediately determine available
 32337 memory. This is particularly important for typed memory objects that may in some cases be
 32338 scarce resources. Note that when a typed memory pool is a shared resource, some form of
 32339 mutual-exclusion or synchronization may be required while typed memory is being queried and

posix_typed_mem_get_info()

32340 allocated to prevent race conditions.

32341 The existing *fstat()* function is not suitable for this purpose. We realize that implementations
32342 may wish to provide other attributes of typed memory objects (for example, alignment
32343 requirements, page size, and so on). The *fstat()* function returns a structure which is not
32344 extensible and, furthermore, contains substantial information that is inappropriate for typed
32345 memory objects.

FUTURE DIRECTIONS

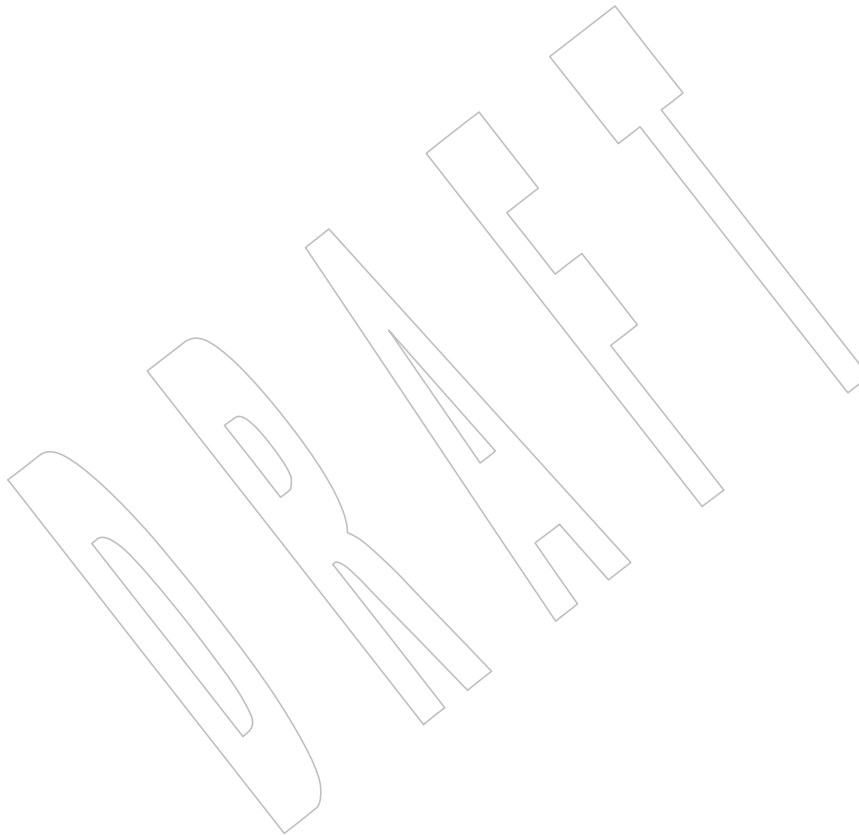
32346 None.
32347

SEE ALSO

32348 *fstat()*, *mmap()*, *posix_typed_mem_open()*, the Base Definitions volume of IEEE Std 1003.1-200x,
32349 <sys/mman.h>
32350

CHANGE HISTORY

32351 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.
32352



32353 **NAME**32354 `posix_typed_mem_open` — open a typed memory object (**ADVANCED REALTIME**)32355 **SYNOPSIS**

```
32356 TYM #include <sys/mman.h>
32357 int posix_typed_mem_open(const char *name, int oflag, int tflag);
```

32358 **DESCRIPTION**

32359 The `posix_typed_mem_open()` function shall establish a connection between the typed memory
 32360 object specified by the string pointed to by `name` and a file descriptor. It shall create an open file
 32361 description that refers to the typed memory object and a file descriptor that refers to that open
 32362 file description. The file descriptor is used by other functions to refer to that typed memory
 32363 object. It is unspecified whether the name appears in the file system and is visible to other
 32364 functions that take pathnames as arguments. The `name` argument shall conform to the
 32365 construction rules for a pathname. If `name` begins with the slash character, then processes calling
 32366 `posix_typed_mem_open()` with the same value of `name` shall refer to the same typed memory
 32367 object. If `name` does not begin with the slash character, the effect is implementation-defined. The
 32368 interpretation of slash characters other than the leading slash character in `name` is
 32369 implementation-defined.

32370 Each typed memory object supported in a system shall be identified by a name which specifies
 32371 not only its associated typed memory pool, but also the path or port by which it is accessed. That
 32372 is, the same typed memory pool accessed via several different ports shall have several different
 32373 corresponding names. The binding between names and typed memory objects is established in
 32374 an implementation-defined manner. Unlike shared memory objects, there is no way within
 32375 IEEE Std 1003.1-200x for a program to create a typed memory object.

32376 The value of `tflag` shall determine how the typed memory object behaves when subsequently
 32377 mapped by calls to `mmap()`. At most, one of the following flags defined in `<sys/mman.h>` may
 32378 be specified:

32379 `POSIX_TYPED_MEM_ALLOCATE`
 32380 Allocate on `mmap()`.

32381 `POSIX_TYPED_MEM_ALLOCATE_CONTIG`
 32382 Allocate contiguously on `mmap()`.

32383 `POSIX_TYPED_MEM_MAP_ALLOCATABLE`
 32384 Map on `mmap()`, without affecting allocatability.

32385 If `tflag` has the flag `POSIX_TYPED_MEM_ALLOCATE` specified, any subsequent call to `mmap()`
 32386 using the returned file descriptor shall result in allocation and mapping of typed memory from
 32387 the specified typed memory pool. The allocated memory may be a contiguous previously
 32388 unallocated area of the typed memory pool or several non-contiguous previously unallocated
 32389 areas (mapped to a contiguous portion of the process address space). If `tflag` has the flag
 32390 `POSIX_TYPED_MEM_ALLOCATE_CONTIG` specified, any subsequent call to `mmap()` using the
 32391 returned file descriptor shall result in allocation and mapping of a single contiguous previously
 32392 unallocated area of the typed memory pool (also mapped to a contiguous portion of the process
 32393 address space). If `tflag` has none of the flags `POSIX_TYPED_MEM_ALLOCATE` or
 32394 `POSIX_TYPED_MEM_ALLOCATE_CONTIG` specified, any subsequent call to `mmap()` using the
 32395 returned file descriptor shall map an application-chosen area from the specified typed memory
 32396 pool such that this mapped area becomes unavailable for allocation until unmapped by all
 32397 processes. If `tflag` has the flag `POSIX_TYPED_MEM_MAP_ALLOCATABLE` specified, any
 32398 subsequent call to `mmap()` using the returned file descriptor shall map an application-chosen

32399 area from the specified typed memory pool without an effect on the availability of that area for
 32400 allocation; that is, mapping such an object leaves each byte of the mapped area unallocated if it
 32401 was unallocated prior to the mapping or allocated if it was allocated prior to the mapping. The
 32402 appropriate privilege to specify the `POSIX_TYPED_MEM_MAP_ALLOCATABLE` flag is
 32403 implementation-defined.

32404 If successful, `posix_typed_mem_open()` shall return a file descriptor for the typed memory object
 32405 that is the lowest numbered file descriptor not currently open for that process. The open file
 32406 description is new, and therefore the file descriptor shall not share it with any other processes. It
 32407 is unspecified whether the file offset is set. The `FD_CLOEXEC` file descriptor flag associated
 32408 with the new file descriptor shall be cleared.

32409 The behavior of `msync()`, `ftruncate()`, and all file operations other than `mmap()`,
 32410 `posix_mem_offset()`, `posix_typed_mem_get_info()`, `fstat()`, `dup()`, `dup2()`, and `close()`, is unspecified
 32411 when passed a file descriptor connected to a typed memory object by this function.

32412 The file status flags of the open file description shall be set according to the value of `oflag`.
 32413 Applications shall specify exactly one of the three access mode values described below and
 32414 defined in the `<fcntl.h>` header, as the value of `oflag`.

32415 `O_RDONLY` Open for read access only.
 32416 `O_WRONLY` Open for write access only.
 32417 `O_RDWR` Open for read or write access.

RETURN VALUE

32418 Upon successful completion, the `posix_typed_mem_open()` function shall return a non-negative
 32419 integer representing the lowest numbered unused file descriptor. Otherwise, it shall return `-1`
 32420 and set `errno` to indicate the error.
 32421

ERRORS

32422 The `posix_typed_mem_open()` function shall fail if:

32423 [EACCES] The typed memory object exists and the permissions specified by `oflag` are
 32424 denied.
 32425
 32426 [EINTR] The `posix_typed_mem_open()` operation was interrupted by a signal.
 32427
 32428 [EINVAL] The flags specified in `tflag` are invalid (more than one of
 32429 `POSIX_TYPED_MEM_ALLOCATE`,
 32430 `POSIX_TYPED_MEM_ALLOCATE_CONTIG`, or
 32431 `POSIX_TYPED_MEM_MAP_ALLOCATABLE` is specified).
 32432
 32433 [EMFILE] All file descriptors available to the process are currently open.
 32434
 32435 [ENAMETOOLONG] The length of the `name` argument exceeds `{PATH_MAX}` or a pathname
 32436 component is longer than `{NAME_MAX}`.
 32437
 32438 [ENFILE] Too many file descriptors are currently open in the system.
 32439
 32440 [ENOENT] The named typed memory object does not exist.
 32441
 32442 [EPERM] The caller lacks the appropriate privilege to specify the flag
 32443 `POSIX_TYPED_MEM_MAP_ALLOCATABLE` in argument `tflag`.
 32444

32439
32440

32441
32442

32443
32444

32445
32446

32447
32448
32449
32450

32451
32452

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

close(), *dup()*, *exec*, *fcntl()*, *fstat()*, *fruncate()*, *mmap()*, *msync()*, *posix_mem_offset()*, *posix_typed_mem_get_info()*, *umask()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<fcntl.h>`, `<sys/mman.h>`

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

DRAFT

32453 **NAME**32454 `pow`, `powf`, `powl` — power function32455 **SYNOPSIS**32456 `#include <math.h>`32457 `double pow(double x, double y);`32458 `float powf(float x, float y);`32459 `long double powl(long double x, long double y);`32460 **DESCRIPTION**32461 CX The functionality described on this reference page is aligned with the ISO C standard. Any
32462 conflict between the requirements described here and the ISO C standard is unintentional. This
32463 volume of IEEE Std 1003.1-200x defers to the ISO C standard.32464 These functions shall compute the value of x raised to the power y , x^y . If x is negative, the
32465 application shall ensure that y is an integer value.32466 An application wishing to check for error situations should set *errno* to zero and call
32467 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
32468 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
32469 zero, an error has occurred.32470 **RETURN VALUE**32471 Upon successful completion, these functions shall return the value of x raised to the power y .32472 MX For finite values of $x < 0$, and finite non-integer values of y , a domain error shall occur and
32473 either a NaN (if representable), or an implementation-defined value shall be returned.32474 If the correct value would cause overflow, a range error shall occur and *pow()*, *powf()*, and
32475 *powl()* shall return \pm HUGE_VAL, \pm HUGE_VALF, and \pm HUGE_VALL, respectively, with the
32476 same sign as the correct value of the function.32477 If the correct value would cause underflow, and is not representable, a range error may occur,
32478 and either 0.0 (if supported), or an implementation-defined value shall be returned.32479 CX For $y < 0$, if x is zero, a pole error may occur and *pow()*, *powf()*, and *powl()* shall return
32480 \pm HUGE_VAL, \pm HUGE_VALF, and \pm HUGE_VALL, respectively. On systems that support the
32481 IEC 60559 Floating-Point option, a pole error shall occur and *pow()*, *powf()*, and *powl()* shall
32482 return \pm HUGE_VAL, \pm HUGE_VALF, and \pm HUGE_VALL, respectively if y is an odd integer, or
32483 HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively if y is not an odd integer.32484 MX If x or y is a NaN, a NaN shall be returned (unless specified elsewhere in this description).32485 For any value of y (including NaN), if x is +1, 1.0 shall be returned.32486 For any value of x (including NaN), if y is ± 0 , 1.0 shall be returned.32487 For any odd integer value of $y > 0$, if x is ± 0 , ± 0 shall be returned.32488 For $y > 0$ and not an odd integer, if x is ± 0 , +0 shall be returned.32489 If x is -1 , and y is \pm Inf, 1.0 shall be returned.32490 For $|x| < 1$, if y is $-\text{Inf}$, $+\text{Inf}$ shall be returned.32491 For $|x| > 1$, if y is $-\text{Inf}$, +0 shall be returned.32492 For $|x| < 1$, if y is $+\text{Inf}$, +0 shall be returned.32493 For $|x| > 1$, if y is $+\text{Inf}$, $+\text{Inf}$ shall be returned.

32494 For y an odd integer < 0 , if x is $-\text{Inf}$, -0 shall be returned.

32495 For $y < 0$ and not an odd integer, if x is $-\text{Inf}$, $+0$ shall be returned.

32496 For y an odd integer > 0 , if x is $-\text{Inf}$, $-\text{Inf}$ shall be returned.

32497 For $y > 0$ and not an odd integer, if x is $-\text{Inf}$, $+\text{Inf}$ shall be returned.

32498 For $y < 0$, if x is $+\text{Inf}$, $+0$ shall be returned.

32499 For $y > 0$, if x is $+\text{Inf}$, $+\text{Inf}$ shall be returned.

32500 If the correct value would cause underflow, and is representable, a range error may occur and

32501 the correct value shall be returned.

ERRORS

32502 These functions shall fail if:

- 32503
- 32504 **Domain Error** The value of x is negative and y is a finite non-integer.
- 32505 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
- 32506 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
- 32507 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
- 32508 shall be raised.
- 32509 **Pole Error** The value of x is zero and y is negative.
- 32510 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
- 32511 then *errno* shall be set to [ERANGE]. If the integer expression
- 32512 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by-zero
- 32513 floating-point exception shall be raised.
- 32514 **Range Error** The result overflows.
- 32515 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
- 32516 then *errno* shall be set to [ERANGE]. If the integer expression
- 32517 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
- 32518 floating-point exception shall be raised.
- 32519 These functions may fail if:
- 32520 **Pole Error** The value of x is zero and y is negative.
- 32521 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
- 32522 then *errno* shall be set to [ERANGE]. If the integer expression
- 32523 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by-zero
- 32524 floating-point exception shall be raised.
- 32525 **Range Error** The result underflows.
- 32526 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
- 32527 then *errno* shall be set to [ERANGE]. If the integer expression
- 32528 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
- 32529 floating-point exception shall be raised.

pow()

32530

EXAMPLES

32531

None.

32532

APPLICATION USAGE

32533

On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

32534

32535

RATIONALE

32536

None.

32537

FUTURE DIRECTIONS

32538

None.

32539

SEE ALSO

32540

exp(), *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

32541

32542

CHANGE HISTORY

32543

First released in Issue 1. Derived from Issue 1 of the SVID.

32544

Issue 5

32545

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

32546

32547

Issue 6

32548

The normative text is updated to avoid use of the term “must” for application requirements.

32549

The *powf()* and *powl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

32550

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

32551

32552

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

32553

32554

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/42 is applied, correcting the third paragraph in the RETURN VALUE section.

32555

32556

Issue 7

32557

ISO/IEC 9899:1999 standard, Technical Corrigendum 2 #51 (SD5-XSH-ERN-81) is applied.

32558

NAME

32559

pread — read from a file

32560

SYNOPSIS

32561

#include <unistd.h>

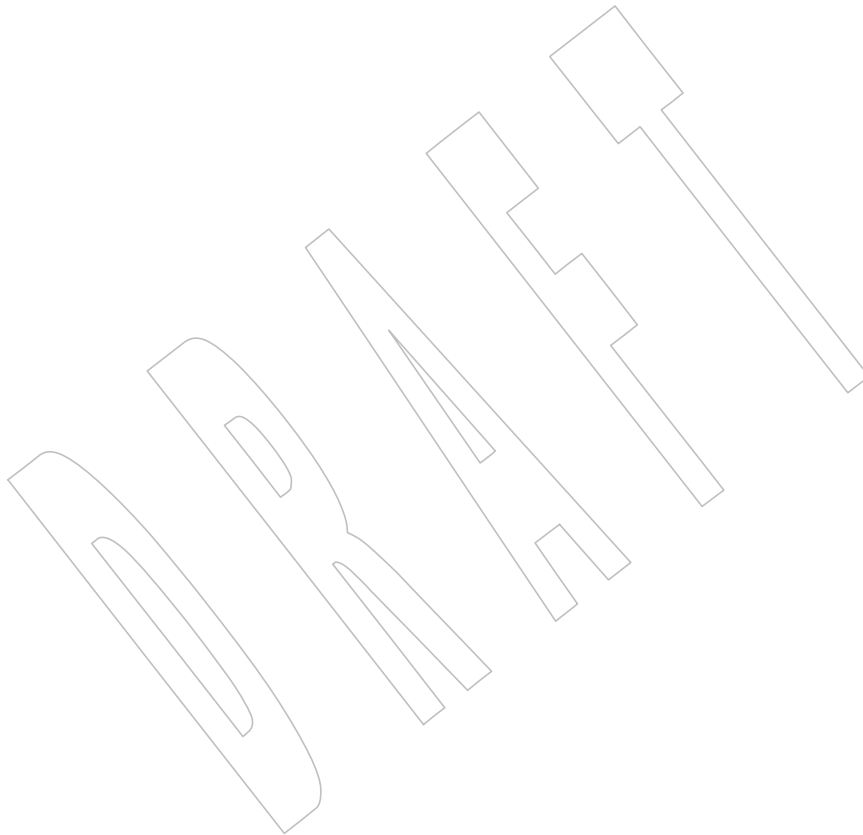
32562

ssize_t pread(int *fd*, void **buf*, size_t *nbyte*, off_t *offset*);

32563

DESCRIPTION

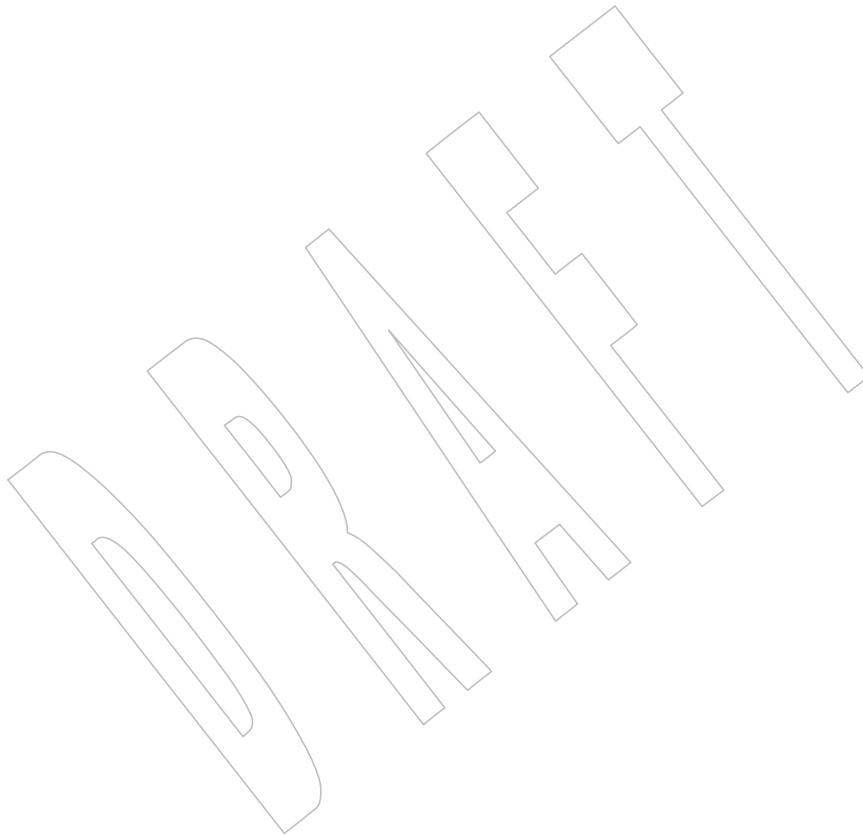
32564

Refer to *read()*.

32565 **NAME**
32566 printf — print formatted output

32567 **SYNOPSIS**
32568 #include <stdio.h>
32569 int printf(const char *restrict *format*, ...);

32570 **DESCRIPTION**
32571 Refer to *fprintf()*.



32572 **NAME**
 32573 pselect, select — synchronous I/O multiplexing

32574 **SYNOPSIS**

```
32575 #include <sys/select.h>
32576
32576 int pselect(int nfds, fd_set *restrict readfds,
32577            fd_set *restrict writefds, fd_set *restrict errorfds,
32578            const struct timespec *restrict timeout,
32579            const sigset_t *restrict sigmask);
32580 int select(int nfds, fd_set *restrict readfds,
32581           fd_set *restrict writefds, fd_set *restrict errorfds,
32582           struct timeval *restrict timeout);
32583 void FD_CLR(int fd, fd_set *fdset);
32584 int FD_ISSET(int fd, fd_set *fdset);
32585 void FD_SET(int fd, fd_set *fdset);
32586 void FD_ZERO(fd_set *fdset);
```

32587 **DESCRIPTION**

32588 The *pselect()* function shall examine the file descriptor sets whose addresses are passed in the
 32589 *readfds*, *writefds*, and *errorfds* parameters to see whether some of their descriptors are ready for
 32590 reading, are ready for writing, or have an exceptional condition pending, respectively.

32591 The *select()* function shall be equivalent to the *pselect()* function, except as follows:

- 32592 • For the *select()* function, the timeout period is given in seconds and microseconds in an
 32593 argument of type **struct timeval**, whereas for the *pselect()* function the timeout period is
 32594 given in seconds and nanoseconds in an argument of type **struct timespec**.
- 32595 • The *select()* function has no *sigmask* argument; it shall behave as *pselect()* does when
 32596 *sigmask* is a null pointer.
- 32597 • Upon successful completion, the *select()* function may modify the object pointed to by the
 32598 *timeout* argument.

32599 The *pselect()* and *select()* functions shall support regular files, terminal and pseudo-terminal
 32600 devices, **STREAMS-based files**, FIFOs, pipes, and sockets. The behavior of *pselect()* and *select()*
 32601 on file descriptors that refer to other types of file is unspecified.

32602 The *nfds* argument specifies the range of descriptors to be tested. The first *nfds* descriptors shall
 32603 be checked in each set; that is, the descriptors from zero through *nfds*-1 in the descriptor sets
 32604 shall be examined.

32605 If the *readfds* argument is not a null pointer, it points to an object of type **fd_set** that on input
 32606 specifies the file descriptors to be checked for being ready to read, and on output indicates
 32607 which file descriptors are ready to read.

32608 If the *writefds* argument is not a null pointer, it points to an object of type **fd_set** that on input
 32609 specifies the file descriptors to be checked for being ready to write, and on output indicates
 32610 which file descriptors are ready to write.

32611 If the *errorfds* argument is not a null pointer, it points to an object of type **fd_set** that on input
 32612 specifies the file descriptors to be checked for error conditions pending, and on output indicates
 32613 which file descriptors have error conditions pending.

32614 Upon successful completion, the *pselect()* or *select()* function shall modify the objects pointed to
 32615 by the *readfds*, *writefds*, and *errorfds* arguments to indicate which file descriptors are ready for
 32616 reading, ready for writing, or have an error condition pending, respectively, and shall return the

32617 total number of ready descriptors in all the output sets. For each file descriptor less than *nfds*, the
 32618 corresponding bit shall be set on successful completion if it was set on input and the associated
 32619 condition is true for that file descriptor.

32620 If none of the selected descriptors are ready for the requested operation, the *pselect()* or *select()*
 32621 function shall block until at least one of the requested operations becomes ready, until the
 32622 *timeout* occurs, or until interrupted by a signal. The *timeout* parameter controls how long the
 32623 *pselect()* or *select()* function shall take before timing out. If the *timeout* parameter is not a null
 32624 pointer, it specifies a maximum interval to wait for the selection to complete. If the specified
 32625 time interval expires without any requested operation becoming ready, the function shall return.
 32626 If the *timeout* parameter is a null pointer, then the call to *pselect()* or *select()* shall block
 32627 indefinitely until at least one descriptor meets the specified criteria. To effect a poll, the *timeout*
 32628 parameter should not be a null pointer, and should point to a zero-valued **timespec** structure.

32629 The use of a timeout does not affect any pending timers set up by *alarm()* or *setitimer()*.

32630 Implementations may place limitations on the maximum timeout interval supported. All
 32631 implementations shall support a maximum timeout interval of at least 31 days. If the *timeout*
 32632 argument specifies a timeout interval greater than the implementation-defined maximum value,
 32633 the maximum value shall be used as the actual timeout value. Implementations may also place
 32634 limitations on the granularity of timeout intervals. If the requested timeout interval requires a
 32635 finer granularity than the implementation supports, the actual timeout interval shall be rounded
 32636 up to the next supported value.

32637 If *sigmask* is not a null pointer, then the *pselect()* function shall replace the signal mask of the
 32638 caller by the set of signals pointed to by *sigmask* before examining the descriptors, and shall
 32639 restore the signal mask of the calling thread before returning.

32640 A descriptor shall be considered ready for reading when a call to an input function with
 32641 O_NONBLOCK clear would not block, whether or not the function would transfer data
 32642 successfully. (The function might return data, an end-of-file indication, or an error other than
 32643 one indicating that it is blocked, and in each of these cases the descriptor shall be considered
 32644 ready for reading.)

32645 A descriptor shall be considered ready for writing when a call to an output function with
 32646 O_NONBLOCK clear would not block, whether or not the function would transfer data
 32647 successfully.

32648 If a socket has a pending error, it shall be considered to have an exceptional condition pending.
 32649 Otherwise, what constitutes an exceptional condition is file type-specific. For a file descriptor for
 32650 use with a socket, it is protocol-specific except as noted below. For other file types it is
 32651 implementation-defined. If the operation is meaningless for a particular file type, *pselect()* or
 32652 *select()* shall indicate that the descriptor is ready for read or write operations, and shall indicate
 32653 that the descriptor has no exceptional condition pending.

32654 If a descriptor refers to a socket, the implied input function is the *recvmsg()* function with
 32655 parameters requesting normal and ancillary data, such that the presence of either type shall
 32656 cause the socket to be marked as readable. The presence of out-of-band data shall be checked if
 32657 the socket option SO_OOBNLINE has been enabled, as out-of-band data is enqueued with
 32658 normal data. If the socket is currently listening, then it shall be marked as readable if an
 32659 incoming connection request has been received, and a call to the *accept()* function shall complete
 32660 without blocking.

32661 If a descriptor refers to a socket, the implied output function is the *sendmsg()* function supplying
 32662 an amount of normal data equal to the current value of the SO_SNDLOWAT option for the
 32663 socket. If a non-blocking call to the *connect()* function has been made for a socket, and the
 32664 connection attempt has either succeeded or failed leaving a pending error, the socket shall be
 32665 marked as writable.

32666 A socket shall be considered to have an exceptional condition pending if a receive operation
 32667 with `O_NONBLOCK` clear for the open file description and with the `MSG_OOB` flag set would
 32668 return out-of-band data without blocking. (It is protocol-specific whether the `MSG_OOB` flag
 32669 would be used to read out-of-band data.) A socket shall also be considered to have an
 32670 exceptional condition pending if an out-of-band data mark is present in the receive queue. Other
 32671 circumstances under which a socket may be considered to have an exceptional condition
 32672 pending are protocol-specific and implementation-defined.

32673 If the `readfds`, `writfds`, and `errorfds` arguments are all null pointers and the `timeout` argument is
 32674 not a null pointer, the `pselect()` or `select()` function shall block for the time specified, or until
 32675 interrupted by a signal. If the `readfds`, `writfds`, and `errorfds` arguments are all null pointers and
 32676 the `timeout` argument is a null pointer, the `pselect()` or `select()` function shall block until
 32677 interrupted by a signal.

32678 File descriptors associated with regular files shall always select true for ready to read, ready to
 32679 write, and error conditions.

32680 On failure, the objects pointed to by the `readfds`, `writfds`, and `errorfds` arguments shall not be
 32681 modified. If the timeout interval expires without the specified condition being true for any of the
 32682 specified file descriptors, the objects pointed to by the `readfds`, `writfds`, and `errorfds` arguments
 32683 shall have all bits set to 0.

32684 File descriptor masks of type `fd_set` can be initialized and tested with `FD_CLR()`, `FD_ISSET()`,
 32685 `FD_SET()`, and `FD_ZERO()`. It is unspecified whether each of these is a macro or a function. If a
 32686 macro definition is suppressed in order to access an actual function, or a program defines an
 32687 external identifier with any of these names, the behavior is undefined.

32688 `FD_CLR(fd, fdsetp)` shall remove the file descriptor *fd* from the set pointed to by *fdsetp*. If *fd* is not
 32689 a member of this set, there shall be no effect on the set, nor will an error be returned.

32690 `FD_ISSET(fd, fdsetp)` shall evaluate to non-zero if the file descriptor *fd* is a member of the set
 32691 pointed to by *fdsetp*, and shall evaluate to zero otherwise.

32692 `FD_SET(fd, fdsetp)` shall add the file descriptor *fd* to the set pointed to by *fdsetp*. If the file
 32693 descriptor *fd* is already in this set, there shall be no effect on the set, nor will an error be
 32694 returned.

32695 `FD_ZERO(fdsetp)` shall initialize the descriptor set pointed to by *fdsetp* to the null set. No error is
 32696 returned if the set is not empty at the time `FD_ZERO()` is invoked.

32697 The behavior of these macros is undefined if the *fd* argument is less than 0 or greater than or
 32698 equal to `FD_SETSIZE`, or if *fd* is not a valid file descriptor, or if any of the arguments are
 32699 expressions with side effects.

32700 If a thread gets canceled during a `pselect()` call, the signal mask in effect when executing the
 32701 registered cleanup functions is either the original signal mask or the signal mask installed as part
 32702 of the `pselect()` call.

32703 RETURN VALUE

32704 Upon successful completion, the `pselect()` and `select()` functions shall return the total number of
 32705 bits set in the bit masks. Otherwise, `-1` shall be returned, and `errno` shall be set to indicate the
 32706 error.

32707 `FD_CLR()`, `FD_SET()`, and `FD_ZERO()` do not return a value. `FD_ISSET()` shall return a non-
 32708 zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and
 32709 0 otherwise.

pselect()**ERRORS**

- 32710 Under the following conditions, *pselect()* and *select()* shall fail and set *errno* to:
- 32711
- 32712 [EBADF] One or more of the file descriptor sets specified a file descriptor that is not a
32713 valid open file descriptor.
- 32714 [EINTR] The function was interrupted before any of the selected events occurred and
32715 before the timeout interval expired.
- 32716 XSI If SA_RESTART has been set for the interrupting signal, it is implementation-
32717 defined whether the function restarts or returns with [EINTR].
- 32718 [EINVAL] An invalid timeout interval was specified.
- 32719 [EINVAL] The *nfds* argument is less than 0 or greater than FD_SETSIZE.
- 32720 OB XSR [EINVAL] One of the specified file descriptors refers to a STREAM or multiplexer that is
32721 linked (directly or indirectly) downstream from a multiplexer.

EXAMPLES

32722 None.
32723

APPLICATION USAGE

32724 None.
32725

RATIONALE

32726 In previous versions of the Single UNIX Specification, the *select()* function was defined in the
32727 `<sys/time.h>` header. This is now changed to `<sys/select.h>`. The rationale for this change was
32728 as follows: the introduction of the *pselect()* function included the `<sys/select.h>` header and the
32729 `<sys/select.h>` header defines all the related definitions for the *pselect()* and *select()* functions.
32730 Backwards-compatibility to existing XSI implementations is handled by allowing `<sys/time.h>`
32731 to include `<sys/select.h>`.
32732

32733 Code which wants to avoid the ambiguity of the signal mask for thread cancellation handlers
32734 can install an additional cancellation handler which resets the signal mask to the expected value.

```

32735 void cleanup(void *arg)
32736 {
32737     sigset_t *ss = (sigset_t *) arg;
32738     pthread_sigmask(SIG_SETMASK, ss, NULL);
32739 }
32740 int call_pselect(int nfds, fd_set *readfds, fd_set *writefds,
32741                 fd_set errorfds, const struct timespec *timeout,
32742                 const sigset_t *sigmask)
32743 {
32744     sigset_t oldmask;
32745     int result;
32746     pthread_sigmask(SIG_SETMASK, NULL, &oldmask);
32747     pthread_cleanup_push(cleanup, &oldmask);
32748     result = pselect(nfds, readfds, writefds, errorfds, timeout, sigmask)
32749     pthread_cleanup_pop(0);
32750     return result;
32751 }

```

FUTURE DIRECTIONS

32752 None.
32753

32754
32755
32756
32757
32758
32759
32760
32761
32762
32763
32764
32765
32766
32767
32768
32769
32770
32771
32772
32773
32774
32775
32776
32777
32778
32779
32780
32781
32782
32783
32784
32785
32786

SEE ALSO

accept(), *alarm()*, *connect()*, *fcntl()*, *poll()*, *read()*, *recvmsg()*, *sendmsg()*, *setitimer()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<sys/select.h>`, `<sys/time.h>`

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

In the ERRORS section, the text has been changed to indicate that [EINVAL] is returned when *nfds* is less than 0 or greater than FD_SETSIZE. It previously stated less than 0, or greater than or equal to FD_SETSIZE.

Text about *timeout* is moved from the APPLICATION USAGE section to the DESCRIPTION.

Issue 6

The Open Group Corrigendum U026/6 is applied, changing the occurrences of *readfs* and *writefs* in the *select()* DESCRIPTION to be *readfds* and *writefds*.

Text referring to sockets is added to the DESCRIPTION.

The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are marked as part of the XSI STREAMS Option Group.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- These functions are now mandatory.

The *pselect()* function is added for alignment with IEEE Std 1003.1g-2000 and additional detail related to sockets semantics is added to the DESCRIPTION.

The *select()* function now requires inclusion of `<sys/select.h>`.

The **restrict** keyword is added to the *select()* prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/70 is applied, updating the DESCRIPTION to reference the signal mask in terms of the calling thread rather than the process.

Issue 7

SD5-XSH-ERN-122 is applied, adding text to the DESCRIPTION for when a thread is canceled during a call to *pselect()*, and adding example code to the RATIONALE.

Functionality relating to the XSI STREAMS option is marked obsolescent.

Functionality relating to the Threads option is moved to the Base.

32787 **NAME**
 32788 `psiginfo, psignal` — print signal information to standard error

32789 **SYNOPSIS**

```
32790 CX #include <signal.h>
32791 void psiginfo(siginfo_t *pinfo, const char *message);
32792 void psignal(int signum, const char *message);
```

32793 **DESCRIPTION**

32794 The `psiginfo()` and `psignal()` functions shall print a message out on `stderr` associated with a signal
 32795 number. If `message` is not null and is not the empty string, then the string pointed to by the
 32796 `message` argument shall be printed first, followed by a colon, a space, and the signal description
 32797 string indicated by `signum`, or by the signal associated with `pinfo`. If the `message` argument is null
 32798 or points to an empty string, then only the signal description shall be printed. For `psiginfo()`, the
 32799 argument `pinfo` references a valid **`siginfo_t`** structure. For `psignal()`, if `signum` is not a valid signal
 32800 number, the behavior is implementation-defined.

32801 **RETURN VALUE**

32802 These functions shall not return a value.

32803 **ERRORS**

32804 No errors are defined.

32805 **EXAMPLES**

32806 None.

32807 **APPLICATION USAGE**

32808 None.

32809 **RATIONALE**

32810 System V historically has `psignal()` and `psiginfo()` in **`<siginfo.h>`**. However, the **`<siginfo.h>`**
 32811 header is not specified in the Base Definitions volume of IEEE Std 1003.1-200x, and the type
 32812 **`siginfo_t`** is defined in **`<signal.h>`**.

32813 **FUTURE DIRECTIONS**

32814 None.

32815 **SEE ALSO**

32816 [*`perror\(\)`*](#), [*`strsignal\(\)`*](#), the Base Definitions volume of IEEE Std 1003.1-200x, **`<signal.h>`**

32817 **CHANGE HISTORY**

32818 First released in Issue 7.

32819 **NAME**
32820 `psignal` — print signal information to standard error

32821 **SYNOPSIS**

32822 CX `#include <signal.h>`
32823 `void psignal(int signum, const char *message);`

32824 **DESCRIPTION**

32825 Refer to *psiginfo()*.

32826 **NAME**

32827 pthread_atfork — register fork handlers

32828 **SYNOPSIS**

32829 #include <pthread.h>

32830 int pthread_atfork(void (*prepare)(void), void (*parent)(void),
32831 void (*child)(void));32832 **DESCRIPTION**

32833 The *pthread_atfork()* function shall declare fork handlers to be called before and after *fork()*, in
 32834 the context of the thread that called *fork()*. The *prepare* fork handler shall be called before *fork()*
 32835 processing commences. The *parent* fork handle shall be called after *fork()* processing completes
 32836 in the parent process. The *child* fork handler shall be called after *fork()* processing completes
 32837 in the child process. If no handling is desired at one or more of these three points, the
 32838 corresponding fork handler address(es) may be set to NULL.

32839 The order of calls to *pthread_atfork()* is significant. The *parent* and *child* fork handlers shall be
 32840 called in the order in which they were established by calls to *pthread_atfork()*. The *prepare* fork
 32841 handlers shall be called in the opposite order.

32842 **RETURN VALUE**

32843 Upon successful completion, *pthread_atfork()* shall return a value of zero; otherwise, an error
 32844 number shall be returned to indicate the error.

32845 **ERRORS**32846 The *pthread_atfork()* function shall fail if:

32847 [ENOMEM] Insufficient table space exists to record the fork handler addresses.

32848 The *pthread_atfork()* function shall not return an error code of [EINTR].32849 **EXAMPLES**

32850 None.

32851 **APPLICATION USAGE**

32852 None.

32853 **RATIONALE**

32854 There are at least two serious problems with the semantics of *fork()* in a multi-threaded
 32855 program. One problem has to do with state (for example, memory) covered by mutexes.
 32856 Consider the case where one thread has a mutex locked and the state covered by that mutex is
 32857 inconsistent while another thread calls *fork()*. In the child, the mutex is in the locked state
 32858 (locked by a nonexistent thread and thus can never be unlocked). Having the child simply
 32859 reinitialize the mutex is unsatisfactory since this approach does not resolve the question about
 32860 how to correct or otherwise deal with the inconsistent state in the child.

32861 It is suggested that programs that use *fork()* call an *exec* function very soon afterwards in the
 32862 child process, thus resetting all states. In the meantime, only a short list of async-signal-safe
 32863 library routines are promised to be available.

32864 Unfortunately, this solution does not address the needs of multi-threaded libraries. Application
 32865 programs may not be aware that a multi-threaded library is in use, and they feel free to call any
 32866 number of library routines between the *fork()* and *exec* calls, just as they always have. Indeed,
 32867 they may be extant single-threaded programs and cannot, therefore, be expected to obey new
 32868 restrictions imposed by the threads library.

32869 On the other hand, the multi-threaded library needs a way to protect its internal state during
 32870 *fork()* in case it is re-entered later in the child process. The problem arises especially in multi-

32871 threaded I/O libraries, which are almost sure to be invoked between the *fork()* and *exec* calls to
 32872 effect I/O redirection. The solution may require locking mutex variables during *fork()*, or it may
 32873 entail simply resetting the state in the child after the *fork()* processing completes.

32874 The *pthread_atfork()* function provides multi-threaded libraries with a means to protect
 32875 themselves from innocent application programs that call *fork()*, and it provides multi-threaded
 32876 application programs with a standard mechanism for protecting themselves from *fork()* calls in a
 32877 library routine or the application itself.

32878 The expected usage is that the *prepare* handler acquires all mutex locks and the other two fork
 32879 handlers release them.

32880 For example, an application can supply a *prepare* routine that acquires the necessary mutexes the
 32881 library maintains and supply *child* and *parent* routines that release those mutexes, thus ensuring
 32882 that the child gets a consistent snapshot of the state of the library (and that no mutexes are left
 32883 stranded). Alternatively, some libraries might be able to supply just a *child* routine that
 32884 reinitializes the mutexes in the library and all associated states to some known value (for
 32885 example, what it was when the image was originally executed).

32886 When *fork()* is called, only the calling thread is duplicated in the child process. Synchronization
 32887 variables remain in the same state in the child as they were in the parent at the time *fork()* was
 32888 called. Thus, for example, mutex locks may be held by threads that no longer exist in the child
 32889 process, and any associated states may be inconsistent. The parent process may avoid this by
 32890 explicit code that acquires and releases locks critical to the child via *pthread_atfork()*. In addition,
 32891 any critical threads need to be recreated and reinitialized to the proper state in the child (also via
 32892 *pthread_atfork()*).

32893 A higher-level package may acquire locks on its own data structures before invoking lower-level
 32894 packages. Under this scenario, the order specified for fork handler calls allows a simple rule of
 32895 initialization for avoiding package deadlock: a package initializes all packages on which it
 32896 depends before it calls the *pthread_atfork()* function for itself.

32897 **FUTURE DIRECTIONS**

32898 None.

32899 **SEE ALSO**

32900 *atexit()*, *exec*, *fork()*, the Base Definitions volume of IEEE Std 1003.1-200x, <sys/types.h>

32901 **CHANGE HISTORY**

32902 First released in Issue 5. Derived from the POSIX Threads Extension.

32903 IEEE PASC Interpretation 1003.1c #4 is applied.

32904 **Issue 6**

32905 The *pthread_atfork()* function is marked as part of the Threads option.

32906 The <pthread.h> header is added to the SYNOPSIS.

32907 **Issue 7**

32908 The *pthread_atfork()* function is moved from the Threads option to the Base.

32909 **NAME**

32910 pthread_attr_destroy, pthread_attr_init — destroy and initialize the thread attributes object

32911 **SYNOPSIS**

32912 #include <pthread.h>

32913 int pthread_attr_destroy(pthread_attr_t *attr);

32914 int pthread_attr_init(pthread_attr_t *attr);

32915 **DESCRIPTION**

32916 The *pthread_attr_destroy()* function shall destroy a thread attributes object. An implementation
 32917 may cause *pthread_attr_destroy()* to set *attr* to an implementation-defined invalid value. A
 32918 destroyed *attr* attributes object can be reinitialized using *pthread_attr_init()*; the results of
 32919 otherwise referencing the object after it has been destroyed are undefined.

32920 The *pthread_attr_init()* function shall initialize a thread attributes object *attr* with the default
 32921 value for all of the individual attributes used by a given implementation.

32922 The resulting attributes object (possibly modified by setting individual attribute values) when
 32923 used by *pthread_create()* defines the attributes of the thread created. A single attributes object can
 32924 be used in multiple simultaneous calls to *pthread_create()*. Results are undefined if
 32925 *pthread_attr_init()* is called specifying an already initialized *attr* attributes object.

32926 **RETURN VALUE**

32927 Upon successful completion, *pthread_attr_destroy()* and *pthread_attr_init()* shall return a value of
 32928 0; otherwise, an error number shall be returned to indicate the error.

32929 **ERRORS**32930 The *pthread_attr_init()* function shall fail if:

32931 [ENOMEM] Insufficient memory exists to initialize the thread attributes object.

32932 The *pthread_attr_destroy()* function may fail if:32933 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute
32934 object.32935 The *pthread_attr_init()* function may fail if:32936 [EBUSY] The implementation has detected an attempt to reinitialize the thread attribute
32937 referenced by *attr*, a previously initialized, but not yet destroyed, thread
32938 attribute.

32939 These functions shall not return an error code of [EINTR].

32940 **EXAMPLES**

32941 None.

32942 **APPLICATION USAGE**

32943 None.

32944 **RATIONALE**

32945 Attributes objects are provided for threads, mutexes, and condition variables as a mechanism to
 32946 support probable future standardization in these areas without requiring that the function itself
 32947 be changed.

32948 Attributes objects provide clean isolation of the configurable aspects of threads. For example,
 32949 “stack size” is an important attribute of a thread, but it cannot be expressed portably. When
 32950 porting a threaded program, stack sizes often need to be adjusted. The use of attributes objects
 32951 can help by allowing the changes to be isolated in a single place, rather than being spread across

32952

every instance of thread creation.

32953

32954

32955

32956

32957

Attributes objects can be used to set up “classes” of threads with similar attributes; for example, “threads with large stacks and high priority” or “threads with minimal stacks”. These classes can be defined in a single place and then referenced wherever threads need to be created. Changes to “class” decisions become straightforward, and detailed analysis of each *pthread_create()* call is not required.

32958

32959

32960

32961

The attributes objects are defined as opaque types as an aid to extensibility. If these objects had been specified as structures, adding new attributes would force recompilation of all multi-threaded programs when the attributes objects are extended; this might not be possible if different program components were supplied by different vendors.

32962

32963

32964

32965

32966

Additionally, opaque attributes objects present opportunities for improving performance. Argument validity can be checked once when attributes are set, rather than each time a thread is created. Implementations often need to cache kernel objects that are expensive to create. Opaque attributes objects provide an efficient mechanism to detect when cached objects become invalid due to attribute changes.

32967

32968

32969

32970

Since assignment is not necessarily defined on a given opaque type, implementation-defined default values cannot be defined in a portable way. The solution to this problem is to allow attributes objects to be initialized dynamically by attributes object initialization functions, so that default values can be supplied automatically by the implementation.

32971

The following proposal was provided as a suggested alternative to the supplied attributes:

32972

32973

32974

32975

32976

32977

1. Maintain the style of passing a parameter formed by the bitwise-inclusive OR of flags to the initialization routines (*pthread_create()*, *pthread_mutex_init()*, *pthread_cond_init()*). The parameter containing the flags should be an opaque type for extensibility. If no flags are set in the parameter, then the objects are created with default characteristics. An implementation may specify implementation-defined flag values and associated behavior.
2. If further specialization of mutexes and condition variables is necessary, implementations may specify additional procedures that operate on the **pthread_mutex_t** and **pthread_cond_t** objects (instead of on attributes objects).

32978

32979

32980

The difficulties with this solution are:

32981

32982

32983

32984

32985

32986

32987

1. A bitmask is not opaque if bits have to be set into bitvector attributes objects using explicitly-coded bitwise-inclusive OR operations. If the set of options exceeds an **int**, application programmers need to know the location of each bit. If bits are set or read by encapsulation (that is, get and set functions), then the bitmask is merely an implementation of attributes objects as currently defined and should not be exposed to the programmer.
2. Many attributes are not Boolean or very small integral values. For example, scheduling policy may be placed in 3-bit or 4-bit, but priority requires 5-bit or more, thereby taking up at least 8 bits out of a possible 16 bits on machines with 16-bit integers. Because of this, the bitmask can only reasonably control whether particular attributes are set or not, and it cannot serve as the repository of the value itself. The value needs to be specified as a function parameter (which is non-extensible), or by setting a structure field (which is non-opaque), or by get and set functions (making the bitmask a redundant addition to the attributes objects).

32988

32989

32990

32991

32992

32993

32994

32995

32996

32997

32998

32999

Stack size is defined as an optional attribute because the very notion of a stack is inherently machine-dependent. Some implementations may not be able to change the size of the stack, for example, and others may not need to because stack pages may be discontinuous and can be allocated and released on demand.

33000 The attribute mechanism has been designed in large measure for extensibility. Future extensions
 33001 to the attribute mechanism or to any attributes object defined in this volume of
 33002 IEEE Std 1003.1-200x has to be done with care so as not to affect binary-compatibility.

33003 Attributes objects, even if allocated by means of dynamic allocation functions such as *malloc()*,
 33004 may have their size fixed at compile time. This means, for example, a *pthread_create()* in an
 33005 implementation with extensions to **pthread_attr_t** cannot look beyond the area that the binary
 33006 application assumes is valid. This suggests that implementations should maintain a size field in
 33007 the attributes object, as well as possibly version information, if extensions in different directions
 33008 (possibly by different vendors) are to be accommodated.

33009 FUTURE DIRECTIONS

33010 None.

33011 SEE ALSO

33012 *pthread_attr_getstacksize()*, *pthread_attr_getdetachstate()*, *pthread_create()*, the Base Definitions
 33013 volume of IEEE Std 1003.1-200x, <pthread.h>

33014 CHANGE HISTORY

33015 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33016 Issue 6

33017 The *pthread_attr_destroy()* and *pthread_attr_init()* functions are marked as part of the Threads
 33018 option.

33019 IEEE PASC Interpretation 1003.1 #107 is applied, noting that the effect of initializing an already
 33020 initialized thread attributes object is undefined.

33021 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/71 is applied, updating the ERRORS
 33022 section to add the optional [EINVAL] error for the *pthread_attr_destroy()* function, and the
 33023 optional [EBUSY] error for the *pthread_attr_init()* function.

33024 Issue 7

33025 The *pthread_attr_destroy()* and *pthread_attr_init()* functions are moved from the Threads option
 33026 to the Base.

33027 **NAME**

33028 pthread_attr_getdetachstate, pthread_attr_setdetachstate — get and set the detachstate attribute

33029 **SYNOPSIS**

```
33030 #include <pthread.h>
33031
33031 int pthread_attr_getdetachstate(const pthread_attr_t *attr,
33032                               int *detachstate);
33033 int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

33034 **DESCRIPTION**

33035 The *detachstate* attribute controls whether the thread is created in a detached state. If the thread
 33036 is created detached, then use of the ID of the newly created thread by the *pthread_detach()* or
 33037 *pthread_join()* function is an error.

33038 The *pthread_attr_getdetachstate()* and *pthread_attr_setdetachstate()* functions, respectively, shall get
 33039 and set the *detachstate* attribute in the *attr* object.

33040 For *pthread_attr_getdetachstate()*, *detachstate* shall be set to either
 33041 PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE.

33042 For *pthread_attr_setdetachstate()*, the application shall set *detachstate* to either
 33043 PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE.

33044 A value of PTHREAD_CREATE_DETACHED shall cause all threads created with *attr* to be in
 33045 the detached state, whereas using a value of PTHREAD_CREATE_JOINABLE shall cause all
 33046 threads created with *attr* to be in the joinable state. The default value of the *detachstate* attribute
 33047 shall be PTHREAD_CREATE_JOINABLE.

33048 **RETURN VALUE**

33049 Upon successful completion, *pthread_attr_getdetachstate()* and *pthread_attr_setdetachstate()* shall
 33050 return a value of 0; otherwise, an error number shall be returned to indicate the error.

33051 The *pthread_attr_getdetachstate()* function stores the value of the *detachstate* attribute in *detachstate*
 33052 if successful.

33053 **ERRORS**

33054 The *pthread_attr_setdetachstate()* function shall fail if:

33055 [EINVAL] The value of *detachstate* was not valid

33056 These functions may fail if:

33057 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute
 33058 object.

33059 These functions shall not return an error code of [EINTR].

33060 **EXAMPLES**33061 **Retrieving the detachstate Attribute**

33062 This example shows how to obtain the *detachstate* attribute of a thread attribute object.

```
33063 #include <pthread.h>
33064
33064 pthread_attr_t thread_attr;
33065 int detachstate;
33066 int rc;
33067
33067 /* code initializing thread_attr */
```

```

33068     ...
33069     rc = pthread_attr_getdetachstate (&thread_attr, &detachstate);
33070     if (rc!=0) {
33071         /* handle error */
33072         ...
33073     }
33074     else {
33075         /* legal values for detachstate are:
33076          * PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE
33077          */
33078         ...
33079     }

```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[pthread_attr_destroy\(\)](#), [pthread_attr_getstacksize\(\)](#), [pthread_create\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, <pthread.h>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The [pthread_attr_setdetachstate\(\)](#) and [pthread_attr_getdetachstate\(\)](#) functions are marked as part of the Threads option.

The normative text is updated to avoid use of the term “must” for application requirements.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/72 is applied, adding the example to the EXAMPLES section.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/73 is applied, updating the ERRORS section to include the optional [EINVAL] error.

Issue 7

The [pthread_attr_setdetachstate\(\)](#) and [pthread_attr_getdetachstate\(\)](#) functions are moved from the Threads option to the Base.

33102 **NAME**

33103 pthread_attr_getguardsize, pthread_attr_setguardsize — get and set the thread guardsize
 33104 attribute

33105 **SYNOPSIS**

```
33106 #include <pthread.h>
33107
33107 int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
33108 size_t *restrict guardsize);
33109 int pthread_attr_setguardsize(pthread_attr_t *attr,
33110 size_t guardsize);
```

33111 **DESCRIPTION**

33112 The *pthread_attr_getguardsize()* function shall get the *guardsize* attribute in the *attr* object. This
 33113 attribute shall be returned in the *guardsize* parameter.

33114 The *pthread_attr_setguardsize()* function shall set the *guardsize* attribute in the *attr* object. The new
 33115 value of this attribute shall be obtained from the *guardsize* parameter. If *guardsize* is zero, a guard
 33116 area shall not be provided for threads created with *attr*. If *guardsize* is greater than zero, a guard
 33117 area of at least size *guardsize* bytes shall be provided for each thread created with *attr*.

33118 The *guardsize* attribute controls the size of the guard area for the created thread's stack. The
 33119 *guardsize* attribute provides protection against overflow of the stack pointer. If a thread's stack is
 33120 created with guard protection, the implementation allocates extra memory at the overflow end
 33121 of the stack as a buffer against stack overflow of the stack pointer. If an application overflows
 33122 into this buffer an error shall result (possibly in a SIGSEGV signal being delivered to the thread).

33123 A conforming implementation may round up the value contained in *guardsize* to a multiple of
 33124 the configurable system variable {PAGESIZE} (see <sys/mman.h>). If an implementation
 33125 rounds up the value of *guardsize* to a multiple of {PAGESIZE}, a call to *pthread_attr_getguardsize()*
 33126 specifying *attr* shall store in the *guardsize* parameter the guard size specified by the previous
 33127 *pthread_attr_setguardsize()* function call.

33128 The default value of the *guardsize* attribute is {PAGESIZE} bytes. The actual value of {PAGESIZE}
 33129 is implementation-defined.

33130 If the *stackaddr* attribute has been set (that is, the caller is allocating and managing its own thread
 33131 stacks), the *guardsize* attribute shall be ignored and no protection shall be provided by the
 33132 implementation. It is the responsibility of the application to manage stack overflow along with
 33133 stack allocation and management in this case.

33134 **RETURN VALUE**

33135 If successful, the *pthread_attr_getguardsize()* and *pthread_attr_setguardsize()* functions shall return
 33136 zero; otherwise, an error number shall be returned to indicate the error.

33137 **ERRORS**

33138 These functions shall fail if:

33139 [EINVAL] The parameter *guardsize* is invalid.

33140 These functions may fail if:

33141 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute
 33142 object.

33143 These functions shall not return an error code of [EINTR].

33144 **EXAMPLES**33145 **Retrieving the guardsize Attribute**33146 This example shows how to obtain the *guardsize* attribute of a thread attribute object.

```

33147 #include <pthread.h>
33148 pthread_attr_t thread_attr;
33149 size_t guardsize;
33150 int rc;
33151 /* code initializing thread_attr */
33152 ...
33153 rc = pthread_attr_getguardsize (&thread_attr, &guardsize);
33154 if (rc != 0) {
33155     /* handle error */
33156     ...
33157 }
33158 else {
33159     if (guardsize > 0) {
33160         /* a guard area of at least guardsize bytes is provided */
33161         ...
33162     }
33163     else {
33164         /* no guard area provided */
33165         ...
33166     }
33167 }

```

33168 **APPLICATION USAGE**

33169 None.

33170 **RATIONALE**33171 The *guardsize* attribute is provided to the application for two reasons:

- 33172 1. Overflow protection can potentially result in wasted system resources. An application
33173 that creates a large number of threads, and which knows its threads never overflow their
33174 stack, can save system resources by turning off guard areas.
- 33175 2. When threads allocate large data structures on the stack, large guard areas may be needed
33176 to detect stack overflow.

33177 **FUTURE DIRECTIONS**

33178 None.

33179 **SEE ALSO**33180 The Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**, **<sys/mman.h>**33181 **CHANGE HISTORY**

33182 First released in Issue 5.

33183 **Issue 6**33184 In the ERRORS section, a third [EINVAL] error condition is removed as it is covered by the
33185 second error condition.33186 The **restrict** keyword is added to the *pthread_attr_getguardsize()* prototype for alignment with the
33187 ISO/IEC 9899:1999 standard.33188 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/74 is applied, updating the ERRORS
33189 section to remove the [EINVAL] error ("The attribute *attr* is invalid."), and replacing it with the

33190

optional [EINVAL] error.

33191

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/76 is applied, adding the example to the EXAMPLES section.

33192

33193

Issue 7

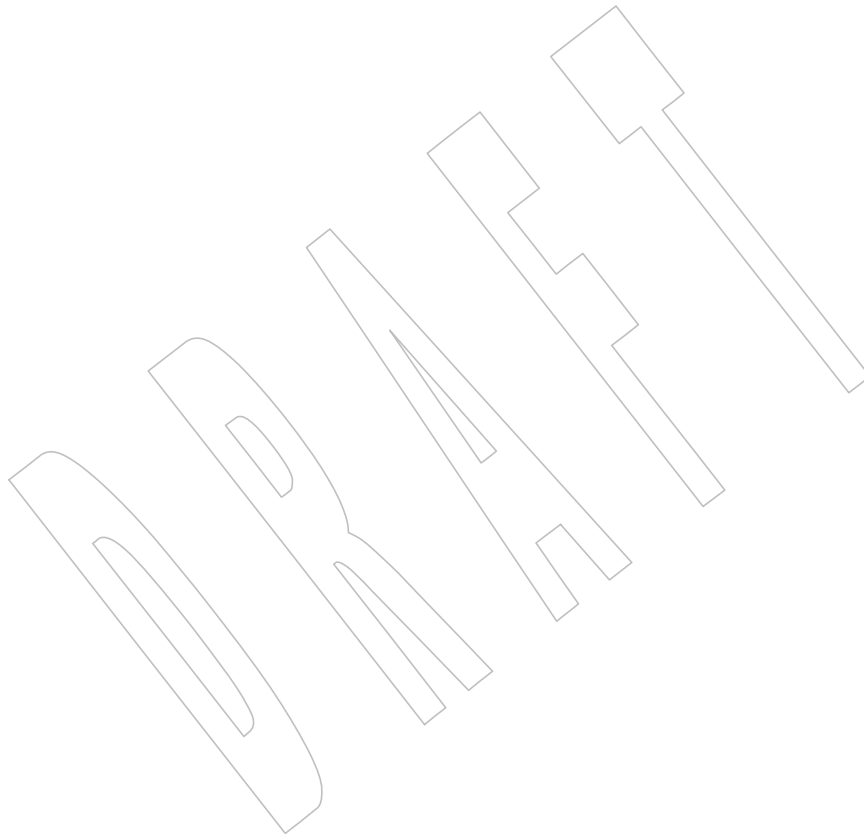
33194

SD5-XSH-ERN-111 is applied, removing the reference to the *stack* attribute in the DESCRIPTION.

33195

The *pthread_attr_getguardsize()* and *pthread_attr_setguardsize()* functions are moved from the XSI option to the Base.

33196



33197 **NAME**

33198 pthread_attr_getinheritsched, pthread_attr_setinheritsched — get and set the inheritsched
 33199 attribute (**REALTIME THREADS**)

33200 **SYNOPSIS**

```
33201 TPS #include <pthread.h>
33202
33202 int pthread_attr_getinheritsched(const pthread_attr_t *restrict attr,
33203 int *restrict inheritsched);
33204 int pthread_attr_setinheritsched(pthread_attr_t *attr,
33205 int inheritsched);
```

33206 **DESCRIPTION**

33207 The *pthread_attr_getinheritsched()*, and *pthread_attr_setinheritsched()* functions, respectively, shall
 33208 get and set the *inheritsched* attribute in the *attr* argument.

33209 When the attributes objects are used by *pthread_create()*, the *inheritsched* attribute determines
 33210 how the other scheduling attributes of the created thread shall be set.

33211 The supported values of *inheritsched* shall be:

33212 PTHREAD_INHERIT_SCHED

33213 Specifies that the thread scheduling attributes shall be inherited from the creating thread,
 33214 and the scheduling attributes in this *attr* argument shall be ignored.

33215 PTHREAD_EXPLICIT_SCHED

33216 Specifies that the thread scheduling attributes shall be set to the corresponding values from
 33217 this attributes object.

33218 The symbols PTHREAD_INHERIT_SCHED and PTHREAD_EXPLICIT_SCHED are defined in
 33219 the **<pthread.h>** header.

33220 The following thread scheduling attributes defined by IEEE Std 1003.1-200x are affected by the
 33221 *inheritsched* attribute: scheduling policy (*schedpolicy*), scheduling parameters (*schedparam*), and
 33222 scheduling contention scope (*contentionscope*).

33223 **RETURN VALUE**

33224 If successful, the *pthread_attr_getinheritsched()* and *pthread_attr_setinheritsched()* functions shall
 33225 return zero; otherwise, an error number shall be returned to indicate the error.

33226 **ERRORS**

33227 The *pthread_attr_getinheritsched()* function may fail if:

33228 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute
 33229 object.

33230 The *pthread_attr_setinheritsched()* function may fail if:

33231 [EINVAL] The value of *inheritsched* is not valid.

33232 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute
 33233 object.

33234 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

33235 These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

After these attributes have been set, a thread can be created with the specified attributes using *pthread_create()*. Using these routines does not affect the current running thread.

See [Section 2.9.4](#) for further details on thread scheduling attributes and their default settings.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_destroy(), *pthread_attr_getscope()*, *pthread_attr_getschedpolicy()*, *pthread_attr_getschedparam()*, *pthread_create()*, the Base Definitions volume of IEEE Std 1003.1-200x, [<pthread.h>](#), [<sched.h>](#)

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

Issue 6

The *pthread_attr_getinheritsched()* and *pthread_attr_setinheritsched()* functions are marked as part of the Threads and Thread Execution Scheduling options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The **restrict** keyword is added to the *pthread_attr_getinheritsched()* prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/75 is applied, clarifying the values of *inheritsched* in the DESCRIPTION and adding two optional [EINVAL] errors to the ERRORS section for checking when *attr* refers to an uninitialized thread attribute object.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/77 is applied, adding a reference to [Section 2.9.4](#) in the APPLICATION USAGE section.

Issue 7

The *pthread_attr_getinheritsched()* and *pthread_attr_setinheritsched()* functions are moved from the Threads option.

33268 **NAME**

33269 pthread_attr_getschedparam, pthread_attr_setschedparam — get and set the schedparam
 33270 attribute

33271 **SYNOPSIS**

```
33272 #include <pthread.h>

33273 int pthread_attr_getschedparam(const pthread_attr_t *restrict attr,
33274                               struct sched_param *restrict param);
33275 int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
33276                               const struct sched_param *restrict param);
```

33277 **DESCRIPTION**

33278 The *pthread_attr_getschedparam()*, and *pthread_attr_setschedparam()* functions, respectively, shall
 33279 get and set the scheduling parameter attributes in the *attr* argument. The contents of the *param*
 33280 structure are defined in the **<sched.h>** header. For the SCHED_FIFO and SCHED_RR policies,
 33281 the only required member of *param* is *sched_priority*.

33282 TSP For the SCHED_SPORADIC policy, the required members of the *param* structure are
 33283 *sched_priority*, *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*, and
 33284 *sched_ss_max_repl*. The specified *sched_ss_repl_period* must be greater than or equal to the
 33285 specified *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.
 33286 The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the
 33287 function to succeed; if not, the function shall fail.

33288 **RETURN VALUE**

33289 If successful, the *pthread_attr_getschedparam()* and *pthread_attr_setschedparam()* functions shall
 33290 return zero; otherwise, an error number shall be returned to indicate the error.

33291 **ERRORS**

33292 The *pthread_attr_getschedparam()* function may fail if:

33293 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute
 33294 object.

33295 The *pthread_attr_setschedparam()* function may fail if:

33296 [EINVAL] The value of *param* is not valid, or the value specified by *attr* does not refer to
 33297 an initialized thread attribute object.

33298 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

33299 These functions shall not return an error code of [EINTR].

33300 **EXAMPLES**

33301 None.

33302 **APPLICATION USAGE**

33303 After these attributes have been set, a thread can be created with the specified attributes using
 33304 *pthread_create()*. Using these routines does not affect the current running thread.

33305 **RATIONALE**

33306 None.

33307 **FUTURE DIRECTIONS**

33308 None.

33309
33310
33311
33312**SEE ALSO**

pthread_attr_destroy(), *pthread_attr_getscope()*, *pthread_attr_getinheritsched()*, *pthread_attr_getschedpolicy()*, *pthread_create()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<pthread.h>`, `<sched.h>`

33313
33314**CHANGE HISTORY**

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33315
33316
33317**Issue 6**

The *pthread_attr_getschedparam()* and *pthread_attr_setschedparam()* functions are marked as part of the Threads option.

33318

The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

33319
33320

The **restrict** keyword is added to the *pthread_attr_getschedparam()* and *pthread_attr_setschedparam()* prototypes for alignment with the ISO/IEC 9899:1999 standard.

33321
33322
33323

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/78 is applied, updating the ERRORS section to include optional errors for the case when *attr* refers to an uninitialized thread attribute object.

33324
33325
33326**Issue 7**

The *pthread_attr_getschedparam()* and *pthread_attr_setschedparam()* functions are moved from the Threads option to the Base.

DRAFT

33327 **NAME**

33328 pthread_attr_getschedpolicy, pthread_attr_setschedpolicy — get and set the schedpolicy
 33329 attribute (**REALTIME THREADS**)

33330 **SYNOPSIS**

```
33331 TPS #include <pthread.h>
33332
33332 int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr,
33333 int *restrict policy);
33334 int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

33335 **DESCRIPTION**

33336 The *pthread_attr_getschedpolicy()* and *pthread_attr_setschedpolicy()* functions, respectively, shall
 33337 get and set the *schedpolicy* attribute in the *attr* argument.

33338 The supported values of *policy* shall include SCHED_FIFO, SCHED_RR, and SCHED_OTHER,
 33339 which are defined in the **<sched.h>** header. When threads executing with the scheduling policy
 33340 TSP SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC are waiting on a mutex, they shall acquire
 33341 the mutex in priority order when the mutex is unlocked.

33342 **RETURN VALUE**

33343 If successful, the *pthread_attr_getschedpolicy()* and *pthread_attr_setschedpolicy()* functions shall
 33344 return zero; otherwise, an error number shall be returned to indicate the error.

33345 **ERRORS**

33346 The *pthread_attr_getschedpolicy()* function may fail if:

33347 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute
 33348 object.

33349 The *pthread_attr_setschedpolicy()* function may fail if:

33350 [EINVAL] The value of *policy* is not valid, or the value specified by *attr* does not refer to
 33351 an initialized thread attribute object.

33352 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

33353 These functions shall not return an error code of [EINTR].

33354 **EXAMPLES**

33355 None.

33356 **APPLICATION USAGE**

33357 After these attributes have been set, a thread can be created with the specified attributes using
 33358 *pthread_create()*. Using these routines does not affect the current running thread.

33359 See [Section 2.9.4](#) for further details on thread scheduling attributes and their default settings.

33360 **RATIONALE**

33361 None.

33362 **FUTURE DIRECTIONS**

33363 None.

33364 **SEE ALSO**

33365 *pthread_attr_destroy()*, *pthread_attr_getscope()*, *pthread_attr_getinheritsched()*,
 33366 *pthread_attr_getschedparam()*, *pthread_create()*, the Base Definitions volume of
 33367 IEEE Std 1003.1-200x, **<pthread.h>**, **<sched.h>**

33368
33369
33370
33371
33372
33373
33374
33375
33376
33377
33378
33379
33380
33381
33382
33383
33384
33385
33386

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

Issue 6

The *pthread_attr_getschedpolicy()* and *pthread_attr_setschedpolicy()* functions are marked as part of the Threads and Thread Execution Scheduling options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

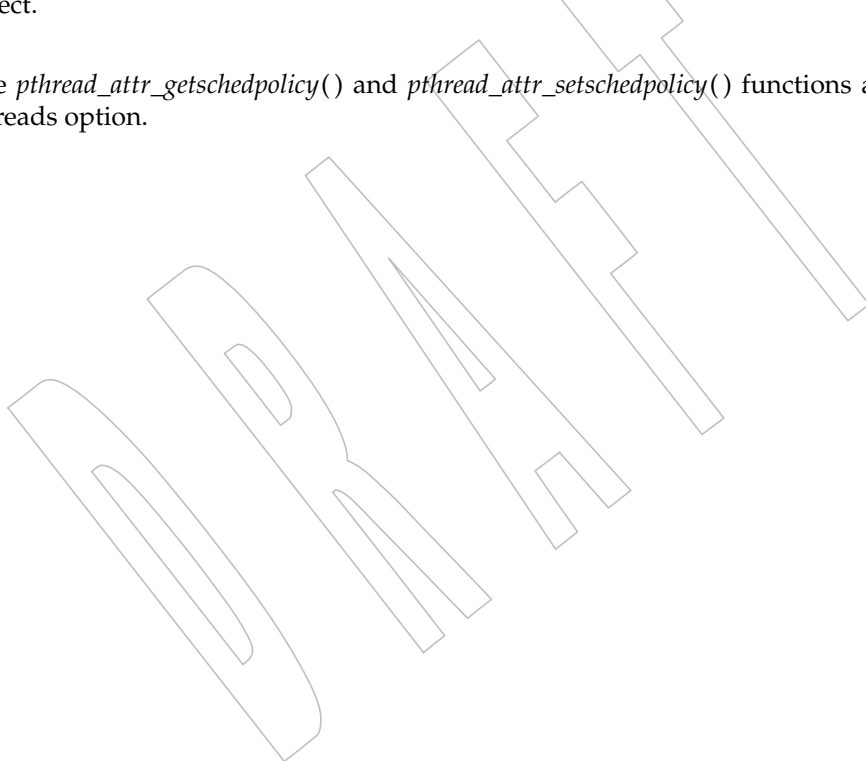
The **restrict** keyword is added to the *pthread_attr_getschedpolicy()* prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/79 is applied, adding a reference to [Section 2.9.4](#) in the APPLICATION USAGE section.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/80 is applied, updating the ERRORS section to include optional errors for the case when *attr* refers to an uninitialized thread attribute object.

Issue 7

The *pthread_attr_getschedpolicy()* and *pthread_attr_setschedpolicy()* functions are moved from the Threads option.



33387 **NAME**

33388 pthread_attr_getscope, pthread_attr_setscope — get and set the contentionscope attribute
 33389 (**REALTIME THREADS**)

33390 **SYNOPSIS**

```
33391 TPS #include <pthread.h>
33392
33392 int pthread_attr_getscope(const pthread_attr_t *restrict attr,
33393 int *restrict contentionscope);
33394 int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

33395 **DESCRIPTION**

33396 The *pthread_attr_getscope()* and *pthread_attr_setscope()* functions, respectively, shall get and set
 33397 the *contentionscope* attribute in the *attr* object.

33398 The *contentionscope* attribute may have the values PTHREAD_SCOPE_SYSTEM, signifying
 33399 system scheduling contention scope, or PTHREAD_SCOPE_PROCESS, signifying process
 33400 scheduling contention scope. The symbols PTHREAD_SCOPE_SYSTEM and
 33401 PTHREAD_SCOPE_PROCESS are defined in the **<pthread.h>** header.

33402 **RETURN VALUE**

33403 If successful, the *pthread_attr_getscope()* and *pthread_attr_setscope()* functions shall return zero;
 33404 otherwise, an error number shall be returned to indicate the error.

33405 **ERRORS**

33406 The *pthread_attr_getscope()* function may fail if:

33407 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute
 33408 object.

33409 The *pthread_attr_setscope()* function may fail if:

33410 [EINVAL] The value of *contentionscope* is not valid, or the value specified by *attr* does not
 33411 refer to an initialized thread attribute object.

33412 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

33413 These functions shall not return an error code of [EINTR].

33414 **EXAMPLES**

33415 None.

33416 **APPLICATION USAGE**

33417 After these attributes have been set, a thread can be created with the specified attributes using
 33418 *pthread_create()*. Using these routines does not affect the current running thread.

33419 See [Section 2.9.4](#) for further details on thread scheduling attributes and their default settings.

33420 **RATIONALE**

33421 None.

33422 **FUTURE DIRECTIONS**

33423 None.

33424 **SEE ALSO**

33425 *pthread_attr_destroy()*, *pthread_attr_getinheritsched()*, *pthread_attr_getschedpolicy()*,
 33426 *pthread_attr_getschedparam()*, *pthread_create()*, the Base Definitions volume of
 33427 IEEE Std 1003.1-200x, **<pthread.h>**, **<sched.h>**

33428
33429
33430
33431
33432
33433
33434
33435
33436
33437
33438
33439
33440
33441
33442
33443
33444
33445

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

Issue 6

The *pthread_attr_getscope()* and *pthread_attr_setscope()* functions are marked as part of the Threads and Thread Execution Scheduling options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

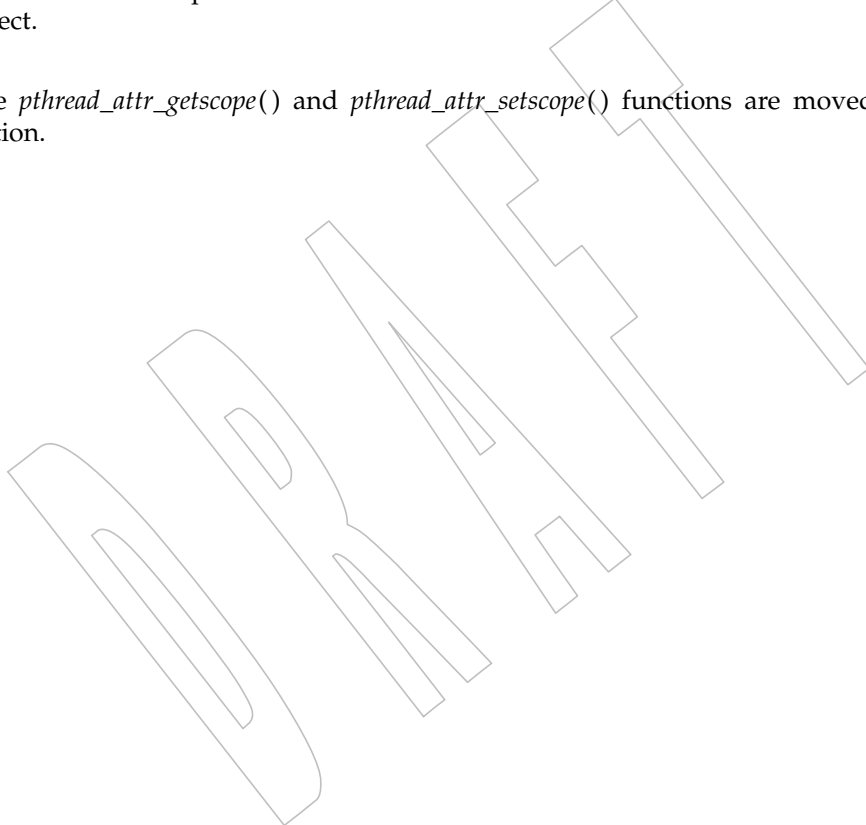
The **restrict** keyword is added to the *pthread_attr_getscope()* prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/81 is applied, adding a reference to [Section 2.9.4](#) in the APPLICATION USAGE section.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/82 is applied, updating the ERRORS section to include optional errors for the case when *attr* refers to an uninitialized thread attribute object.

Issue 7

The *pthread_attr_getscope()* and *pthread_attr_setscope()* functions are moved from the Threads option.



33446 **NAME**

33447 pthread_attr_getstack, pthread_attr_setstack — get and set stack attributes

33448 **SYNOPSIS**

```

33449 TSA TSS #include <pthread.h>
33450
33451 int pthread_attr_getstack(const pthread_attr_t *restrict attr,
33452 void **restrict stackaddr, size_t *restrict stacksize);
33453 int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
33454 size_t stacksize);

```

33454 **DESCRIPTION**33455 The *pthread_attr_getstack()* and *pthread_attr_setstack()* functions, respectively, shall get and set the
33456 thread creation stack attributes *stackaddr* and *stacksize* in the *attr* object.

33457 The stack attributes specify the area of storage to be used for the created thread's stack. The base
33458 (lowest addressable byte) of the storage shall be *stackaddr*, and the size of the storage shall be
33459 *stacksize* bytes. The *stacksize* shall be at least {PTHREAD_STACK_MIN}. The *stackaddr* shall be
33460 aligned appropriately to be used as a stack; for example, *pthread_attr_setstack()* may fail with
33461 [EINVAL] if (*stackaddr* & 0x7) is not 0. All pages within the stack described by *stackaddr* and
33462 *stacksize* shall be both readable and writable by the thread.

33463 If the *pthread_attr_getstack()* function is called before the *stackaddr* attribute has been set, the
33464 behavior is unspecified.

33465 **RETURN VALUE**33466 Upon successful completion, these functions shall return a value of 0; otherwise, an error
33467 number shall be returned to indicate the error.

33468 The *pthread_attr_getstack()* function shall store the stack attribute values in *stackaddr* and *stacksize*
33469 if successful.

33470 **ERRORS**33471 The *pthread_attr_setstack()* function shall fail if:

33472 [EINVAL] The value of *stacksize* is less than {PTHREAD_STACK_MIN} or exceeds an
33473 implementation-defined limit.

33474 The *pthread_attr_getstack()* function may fail if:

33475 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute
33476 object.

33477 The *pthread_attr_setstack()* function may fail if:

33478 [EINVAL] The value of *stackaddr* does not have proper alignment to be used as a stack, or
33479 (*stackaddr* + *stacksize*) lacks proper alignment, or the value specified by *attr*
33480 does not refer to an initialized thread attribute object.

33481 [EACCES] The stack page(s) described by *stackaddr* and *stacksize* are not both readable
33482 and writable by the thread.

33483 These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

These functions are appropriate for use by applications in an environment where the stack for a thread must be placed in some particular region of memory.

While it might seem that an application could detect stack overflow by providing a protected page outside the specified stack region, this cannot be done portably. Implementations are free to place the thread's initial stack pointer anywhere within the specified region to accommodate the machine's stack pointer behavior and allocation requirements. Furthermore, on some architectures, such as the IA-64, "overflow" might mean that two separate stack pointers allocated within the region will overlap somewhere in the middle of the region.

After a successful call to *pthread_attr_setstack()*, the storage area specified by the *stackaddr* parameter is under the control of the implementation, as described in [Section 2.9.8](#) (on page 59).

The specification of the *stackaddr* attribute presents several ambiguities that make portable use of these functions impossible.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_init(), *pthread_attr_setdetachstate()*, *pthread_attr_setstacksize()*, *pthread_create()*, the Base Definitions volume of IEEE Std 1003.1-200x, [<limits.h>](#), [<pthread.h>](#)

CHANGE HISTORY

First released in Issue 6. Developed as part of the XSI option and brought into the BASE by IEEE PASC Interpretation 1003.1 #101.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/83 is applied, updating the APPLICATION USAGE section to refer to [Section 2.9.8](#) (on page 59).

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC/D6/84 is applied, updating the ERRORS section to include optional errors for the case when *attr* refers to an uninitialized thread attribute object.

Issue 7

SD5-XSH-ERN-66 is applied, correcting the use of *attr* in the [EINVAL] error condition.

Austin Group Interpretation 1003.1-2001 #057 is applied, clarifying the behavior if the function is called before the *stackaddr* attribute is set.

SD5-XSH-ERN-157 is applied, updating the APPLICATION USAGE section.

33519 **NAME**

33520 pthread_attr_getstacksize, pthread_attr_setstacksize — get and set the stacksize attribute

33521 **SYNOPSIS**

```

33522 TSS      #include <pthread.h>
33523          int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
33524                                     size_t *restrict stacksize);
33525          int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);

```

33526 **DESCRIPTION**33527 The *pthread_attr_getstacksize()* and *pthread_attr_setstacksize()* functions, respectively, shall get and
33528 set the thread creation *stacksize* attribute in the *attr* object.33529 The *stacksize* attribute shall define the minimum stack size (in bytes) allocated for the created
33530 threads stack.33531 **RETURN VALUE**33532 Upon successful completion, *pthread_attr_getstacksize()* and *pthread_attr_setstacksize()* shall
33533 return a value of 0; otherwise, an error number shall be returned to indicate the error.33534 The *pthread_attr_getstacksize()* function stores the *stacksize* attribute value in *stacksize* if
33535 successful.33536 **ERRORS**33537 The *pthread_attr_setstacksize()* function shall fail if:33538 [EINVAL] The value of *stacksize* is less than {PTHREAD_STACK_MIN} or exceeds a
33539 system-imposed limit.

33540 These functions may fail if:

33541 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute
33542 object.

33543 These functions shall not return an error code of [EINTR].

33544 **EXAMPLES**

33545 None.

33546 **APPLICATION USAGE**

33547 None.

33548 **RATIONALE**

33549 None.

33550 **FUTURE DIRECTIONS**

33551 None.

33552 **SEE ALSO**33553 *pthread_attr_destroy()*, *pthread_attr_getdetachstate()*, *pthread_create()*, the Base Definitions volume
33554 of IEEE Std 1003.1-200x, <limits.h>, <pthread.h>33555 **CHANGE HISTORY**

33556 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33557

Issue 6

33558

The *pthread_attr_getstacksize()* and *pthread_attr_setstacksize()* functions are marked as part of the Threads and Thread Stack Size Attribute options.

33559

33560

The **restrict** keyword is added to the *pthread_attr_getstacksize()* prototype for alignment with the ISO/IEC 9899:1999 standard.

33561

33562

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/43 is applied, correcting the margin code in the SYNOPSIS from TSA to TSS and updating the CHANGE HISTORY from “Thread Stack Address Attribute” option to “Thread Stack Size Attribute” option.

33563

33564

33565

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/87 is applied, updating the ERRORS section to include optional errors for the case when *attr* refers to an uninitialized thread attribute object.

33566

33567

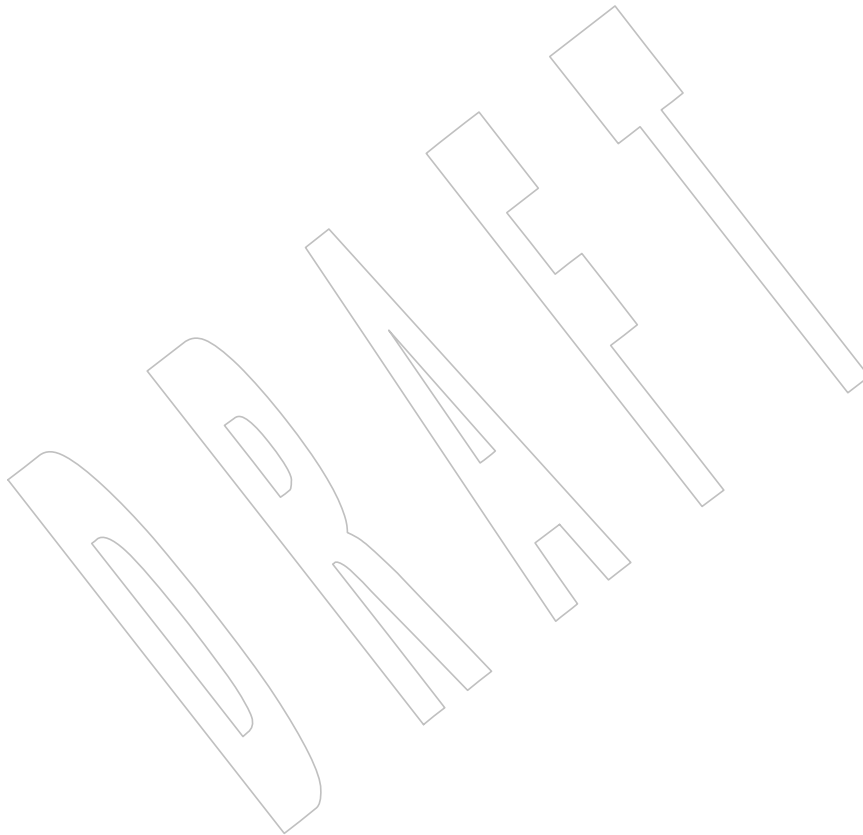
33568

Issue 7

33569

The *pthread_attr_getstacksize()* and *pthread_attr_setstacksize()* functions are moved from the Threads option.

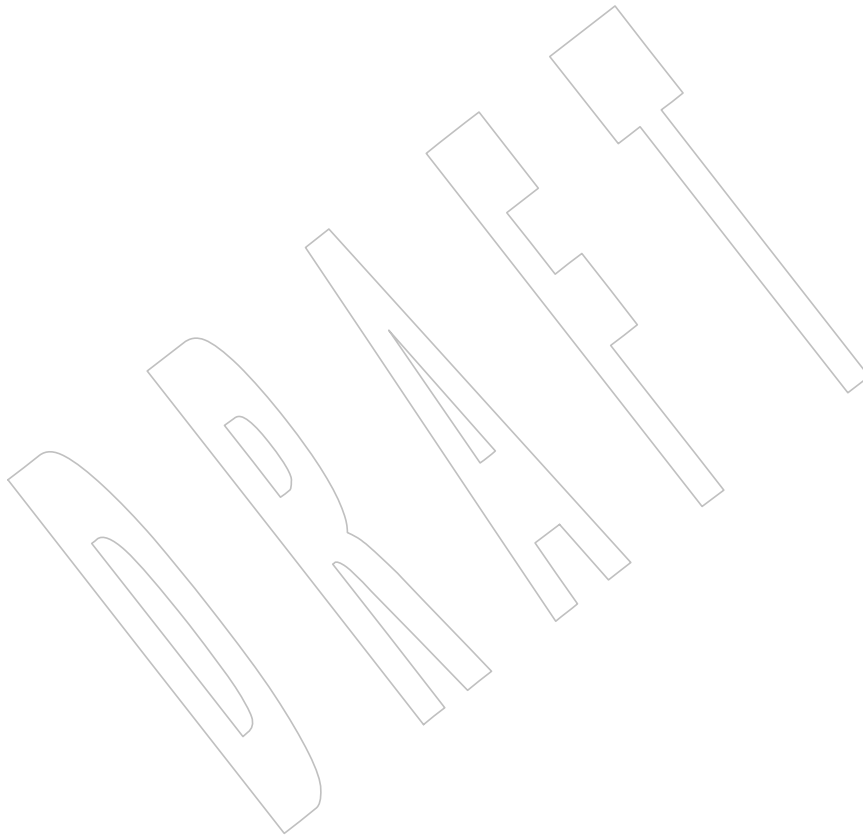
33570



33571 **NAME**
33572 pthread_attr_init — initialize the thread attributes object

33573 **SYNOPSIS**
33574 #include <pthread.h>
33575 int pthread_attr_init(pthread_attr_t *attr);

33576 **DESCRIPTION**
33577 Refer to *pthread_attr_destroy()*.



33578

NAME

33579

`pthread_attr_setdetachstate` — set the detachstate attribute

33580

SYNOPSIS

33581

`#include <pthread.h>`

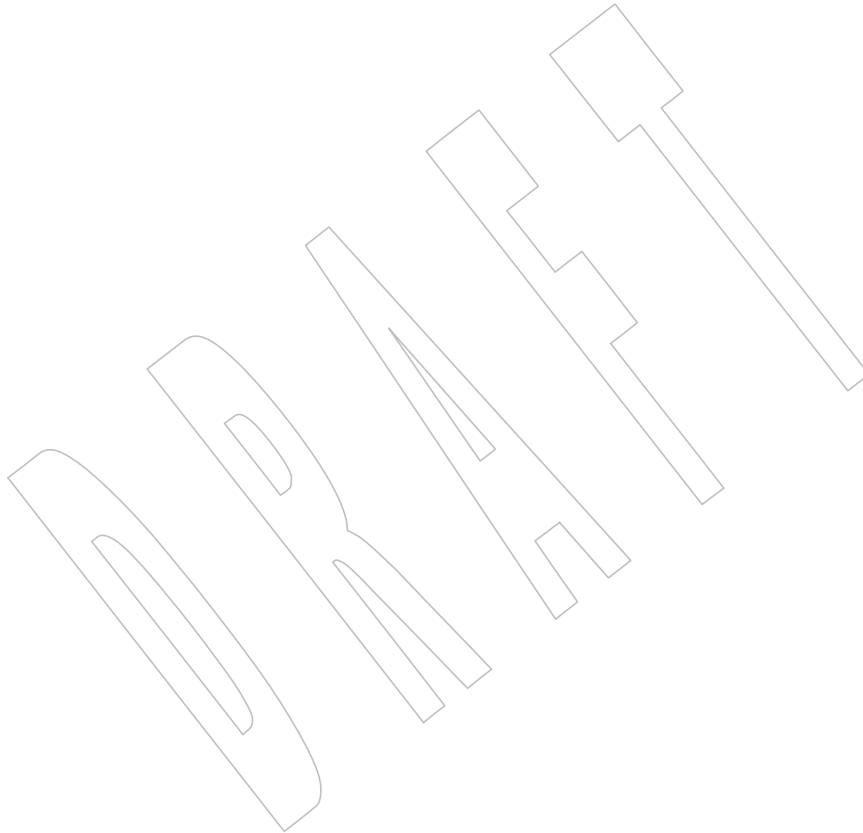
33582

`int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);`

33583

DESCRIPTION

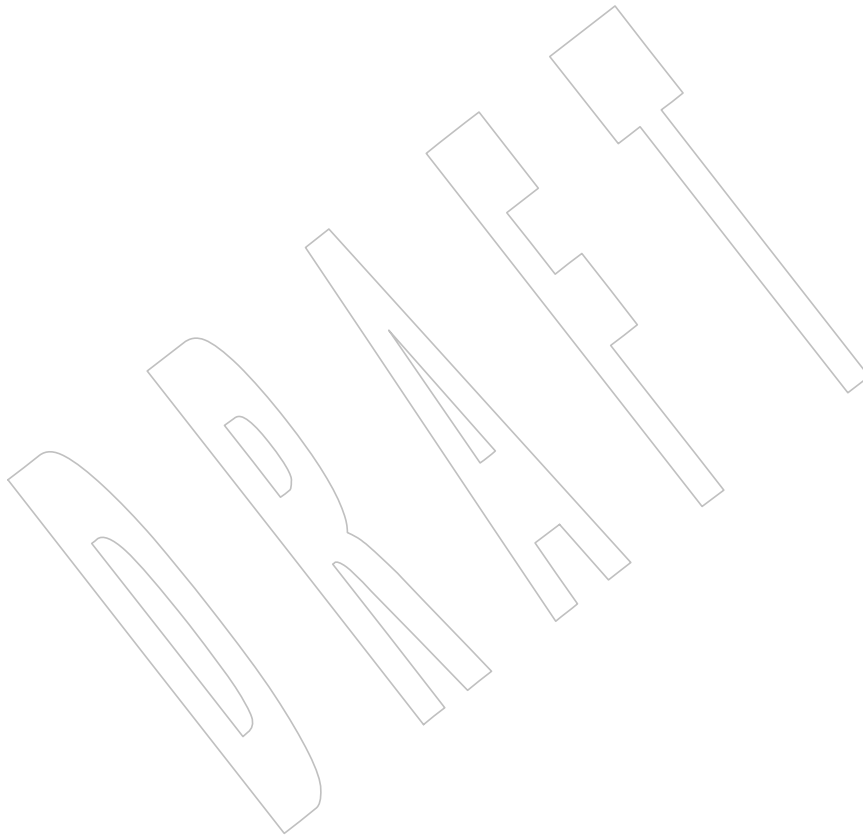
33584

Refer to [pthread_attr_getdetachstate\(\)](#).

33585 **NAME**
33586 pthread_attr_setguardsize — set the thread guardsize attribute

33587 **SYNOPSIS**
33588 #include <pthread.h>
33589 int pthread_attr_setguardsize(pthread_attr_t *attr,
33590 size_t guardsize);

33591 **DESCRIPTION**
33592 Refer to *pthread_attr_getguardsize()*.



33593 **NAME**
33594 pthread_attr_setinheritsched — set the inheritsched attribute (**REALTIME THREADS**)

33595 **SYNOPSIS**

33596 TPS #include <pthread.h>
33597 int pthread_attr_setinheritsched(pthread_attr_t *attr,
33598 int inheritsched);

33599 **DESCRIPTION**

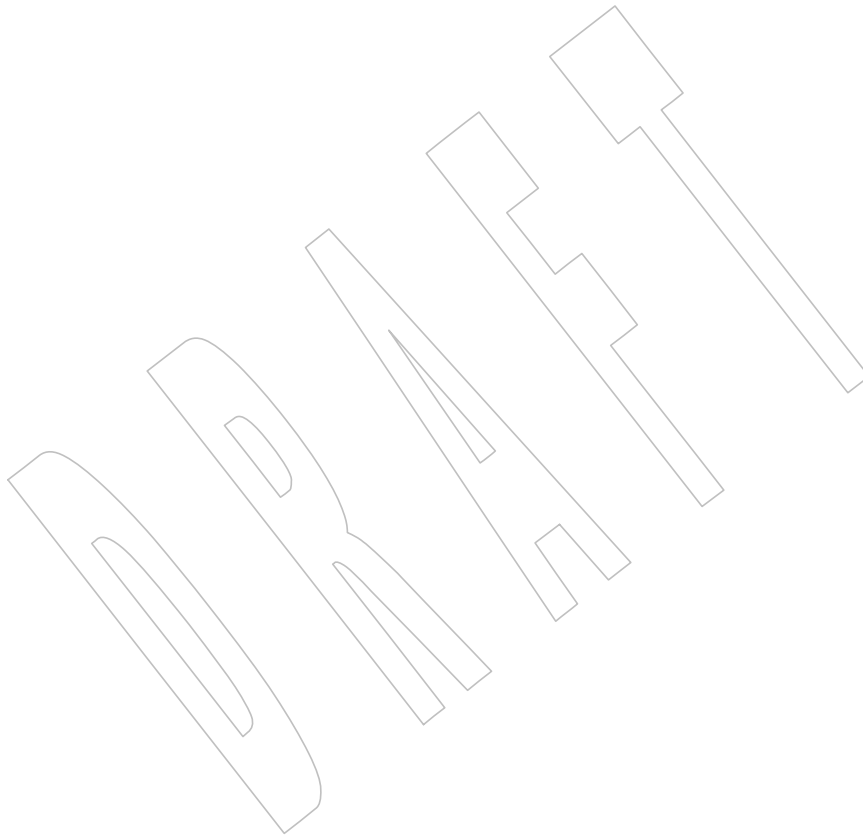
33600 Refer to *pthread_attr_getinheritsched()*.

pthread_attr_setschedparam()

33601 **NAME**
33602 pthread_attr_setschedparam — set the schedparam attribute

33603 **SYNOPSIS**
33604 #include <pthread.h>
33605 int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
33606 const struct sched_param *restrict param);

33607 **DESCRIPTION**
33608 Refer to *pthread_attr_getschedparam()*.



33609 **NAME**
33610 pthread_attr_setschedpolicy — set the schedpolicy attribute (**REALTIME THREADS**)

33611 **SYNOPSIS**

33612 TPS #include <pthread.h>
33613 int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

33614 **DESCRIPTION**

33615 Refer to [pthread_attr_getschedpolicy\(\)](#).

33616 **NAME**33617 pthread_attr_setscope — set the contentionscope attribute (**REALTIME THREADS**)33618 **SYNOPSIS**

33619 TPS #include <pthread.h>

33620 int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);

33621 **DESCRIPTION**33622 Refer to *pthread_attr_getscope()*.

33623 **NAME**
33624 pthread_attr_setstack — set the stack attribute

33625 **SYNOPSIS**

```
33626 TSA TSS #include <pthread.h>  
33627 int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,  
33628 size_t stacksize);
```

33629 **DESCRIPTION**

33630 Refer to *pthread_attr_getstack()*.

pthread_attr_setstacksize()*System Interfaces*

33631 **NAME**
33632 pthread_attr_setstacksize — set the stacksize attribute

SYNOPSIS

33633 TSS #include <pthread.h>
33634
33635 int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);

DESCRIPTION

33636 Refer to *pthread_attr_getstacksize()*.
33637

33638 **NAME**

33639 pthread_barrier_destroy, pthread_barrier_init — destroy and initialize a barrier object

33640 **SYNOPSIS**

33641 #include <pthread.h>

```
33642 int pthread_barrier_destroy(pthread_barrier_t *barrier);
33643 int pthread_barrier_init(pthread_barrier_t *restrict barrier,
33644     const pthread_barrierattr_t *restrict attr, unsigned count);
```

33645 **DESCRIPTION**

33646 The *pthread_barrier_destroy()* function shall destroy the barrier referenced by *barrier* and release
 33647 any resources used by the barrier. The effect of subsequent use of the barrier is undefined until
 33648 the barrier is reinitialized by another call to *pthread_barrier_init()*. An implementation may use
 33649 this function to set *barrier* to an invalid value. The results are undefined if
 33650 *pthread_barrier_destroy()* is called when any thread is blocked on the barrier, or if this function is
 33651 called with an uninitialized barrier.

33652 The *pthread_barrier_init()* function shall allocate any resources required to use the barrier
 33653 referenced by *barrier* and shall initialize the barrier with attributes referenced by *attr*. If *attr* is
 33654 NULL, the default barrier attributes shall be used; the effect is the same as passing the address of
 33655 a default barrier attributes object. The results are undefined if *pthread_barrier_init()* is called
 33656 when any thread is blocked on the barrier (that is, has not returned from the
 33657 *pthread_barrier_wait()* call). The results are undefined if a barrier is used without first being
 33658 initialized. The results are undefined if *pthread_barrier_init()* is called specifying an already
 33659 initialized barrier.

33660 The *count* argument specifies the number of threads that must call *pthread_barrier_wait()* before
 33661 any of them successfully return from the call. The value specified by *count* must be greater than
 33662 zero.

33663 If the *pthread_barrier_init()* function fails, the barrier shall not be initialized and the contents of
 33664 *barrier* are undefined.

33665 Only the object referenced by *barrier* may be used for performing synchronization. The result of
 33666 referring to copies of that object in calls to *pthread_barrier_destroy()* or *pthread_barrier_wait()* is
 33667 undefined.

33668 **RETURN VALUE**

33669 Upon successful completion, these functions shall return zero; otherwise, an error number shall
 33670 be returned to indicate the error.

33671 **ERRORS**

33672 The *pthread_barrier_destroy()* function may fail if:

33673 [EBUSY] The implementation has detected an attempt to destroy a barrier while it is in
 33674 use (for example, while being used in a *pthread_barrier_wait()* call) by another
 33675 thread.

33676 [EINVAL] The value specified by *barrier* is invalid.

33677 The *pthread_barrier_init()* function shall fail if:

33678 [EAGAIN] The system lacks the necessary resources to initialize another barrier.

33679 [EINVAL] The value specified by *count* is equal to zero.

pthread_barrier_destroy()

33680 [ENOMEM] Insufficient memory exists to initialize the barrier.

33681 The *pthread_barrier_init()* function may fail if:

33682 [EBUSY] The implementation has detected an attempt to reinitialize a barrier while it is
 33683 in use (for example, while being used in a *pthread_barrier_wait()* call) by
 33684 another thread.

33685 [EINVAL] The value specified by *attr* is invalid.

33686 These functions shall not return an error code of [EINTR].

33687 **EXAMPLES**

33688 None.

33689 **APPLICATION USAGE**

33690 None.

33691 **RATIONALE**

33692 None.

33693 **FUTURE DIRECTIONS**

33694 None.

33695 **SEE ALSO**

33696 *pthread_barrier_wait()*, the Base Definitions volume of IEEE Std 1003.1-200x, <pthread.h>

33697 **CHANGE HISTORY**

33698 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

33699 **Issue 7**

33700 The *pthread_barrier_destroy()* and *pthread_barrier_init()* functions are moved from the Barriers
 33701 option to the Base.

33702 **NAME**

33703 pthread_barrier_wait — synchronize at a barrier

33704 **SYNOPSIS**

33705 #include <pthread.h>

33706 int pthread_barrier_wait(pthread_barrier_t *barrier);

33707 **DESCRIPTION**33708 The *pthread_barrier_wait()* function shall synchronize participating threads at the barrier
33709 referenced by *barrier*. The calling thread shall block until the required number of threads have
33710 called *pthread_barrier_wait()* specifying the barrier.33711 When the required number of threads have called *pthread_barrier_wait()* specifying the barrier,
33712 the constant PTHREAD_BARRIER_SERIAL_THREAD shall be returned to one unspecified
33713 thread and zero shall be returned to each of the remaining threads. At this point, the barrier shall
33714 be reset to the state it had as a result of the most recent *pthread_barrier_init()* function that
33715 referenced it.33716 The constant PTHREAD_BARRIER_SERIAL_THREAD is defined in <pthread.h> and its value
33717 shall be distinct from any other value returned by *pthread_barrier_wait()*.

33718 The results are undefined if this function is called with an uninitialized barrier.

33719 If a signal is delivered to a thread blocked on a barrier, upon return from the signal handler the
33720 thread shall resume waiting at the barrier if the barrier wait has not completed (that is, if the
33721 required number of threads have not arrived at the barrier during the execution of the signal
33722 handler); otherwise, the thread shall continue as normal from the completed barrier wait. Until
33723 the thread in the signal handler returns from it, it is unspecified whether other threads may
33724 proceed past the barrier once they have all reached it.33725 A thread that has blocked on a barrier shall not prevent any unblocked thread that is eligible to
33726 use the same processing resources from eventually making forward progress in its execution.
33727 Eligibility for processing resources shall be determined by the scheduling policy.33728 **RETURN VALUE**33729 Upon successful completion, the *pthread_barrier_wait()* function shall return
33730 PTHREAD_BARRIER_SERIAL_THREAD for a single (arbitrary) thread synchronized at the
33731 barrier and zero for each of the other threads. Otherwise, an error number shall be returned to
33732 indicate the error.33733 **ERRORS**33734 The *pthread_barrier_wait()* function may fail if:33735 [EINVAL] The value specified by *barrier* does not refer to an initialized barrier object.

33736 This function shall not return an error code of [EINTR].

33737 **EXAMPLES**

33738 None.

33739 **APPLICATION USAGE**33740 Applications using this function may be subject to priority inversion, as discussed in the Base
33741 Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

pthread_barrier_wait()*System Interfaces*

33742

RATIONALE

33743

None.

33744

FUTURE DIRECTIONS

33745

None.

33746

SEE ALSO

33747

pthread_barrier_destroy(), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, **<pthread.h>**

33748

33749

CHANGE HISTORY

33750

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

33751

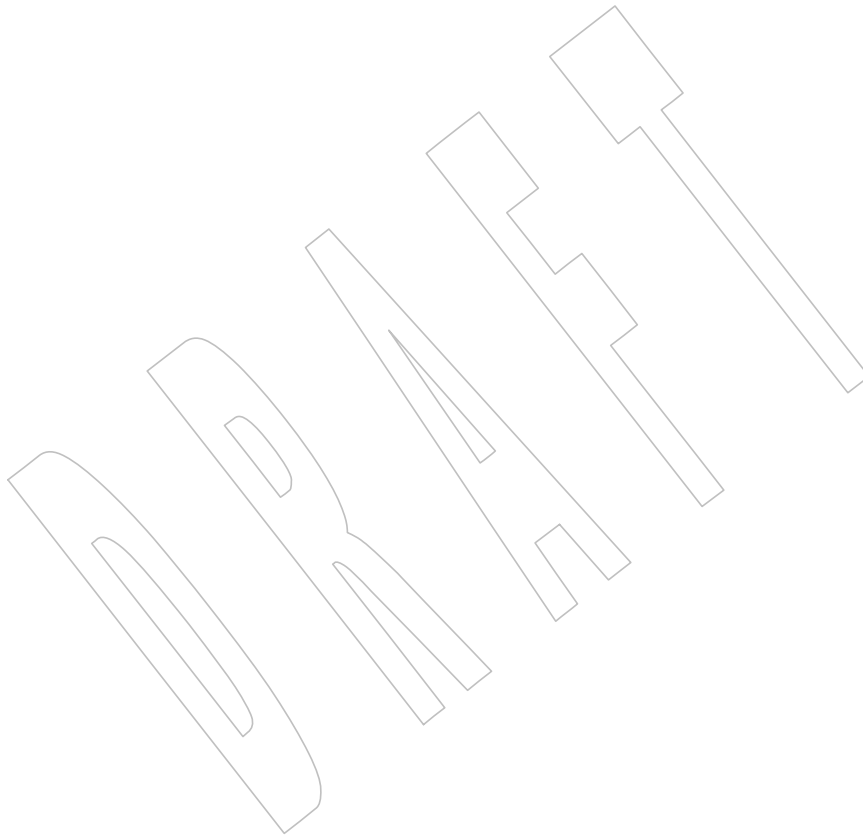
In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

33752

Issue 7

33753

The *pthread_barrier_wait()* function is moved from the Barriers option to the Base.



33754 **NAME**

33755 pthread_barrierattr_destroy, pthread_barrierattr_init — destroy and initialize the barrier
 33756 attributes object

33757 **SYNOPSIS**

33758 #include <pthread.h>

33759 int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);

33760 int pthread_barrierattr_init(pthread_barrierattr_t *attr);

33761 **DESCRIPTION**

33762 The *pthread_barrierattr_destroy()* function shall destroy a barrier attributes object. A destroyed
 33763 *attr* attributes object can be reinitialized using *pthread_barrierattr_init()*; the results of otherwise
 33764 referencing the object after it has been destroyed are undefined. An implementation may cause
 33765 *pthread_barrierattr_destroy()* to set the object referenced by *attr* to an invalid value.

33766 The *pthread_barrierattr_init()* function shall initialize a barrier attributes object *attr* with the
 33767 default value for all of the attributes defined by the implementation.

33768 Results are undefined if *pthread_barrierattr_init()* is called specifying an already initialized *attr*
 33769 attributes object.

33770 After a barrier attributes object has been used to initialize one or more barriers, any function
 33771 affecting the attributes object (including destruction) shall not affect any previously initialized
 33772 barrier.

33773 **RETURN VALUE**

33774 If successful, the *pthread_barrierattr_destroy()* and *pthread_barrierattr_init()* functions shall return
 33775 zero; otherwise, an error number shall be returned to indicate the error.

33776 **ERRORS**

33777 The *pthread_barrierattr_destroy()* function may fail if:

33778 [EINVAL] The value specified by *attr* is invalid.

33779 The *pthread_barrierattr_init()* function shall fail if:

33780 [ENOMEM] Insufficient memory exists to initialize the barrier attributes object.

33781 These functions shall not return an error code of [EINTR].

33782 **EXAMPLES**

33783 None.

33784 **APPLICATION USAGE**

33785 None.

33786 **RATIONALE**

33787 None.

33788 **FUTURE DIRECTIONS**

33789 None.

33790 **SEE ALSO**

33791 *pthread_barrierattr_getpshared()*, *pthread_barrierattr_setpshared()*, the Base Definitions volume of
 33792 IEEE Std 1003.1-200x, <pthread.h>.

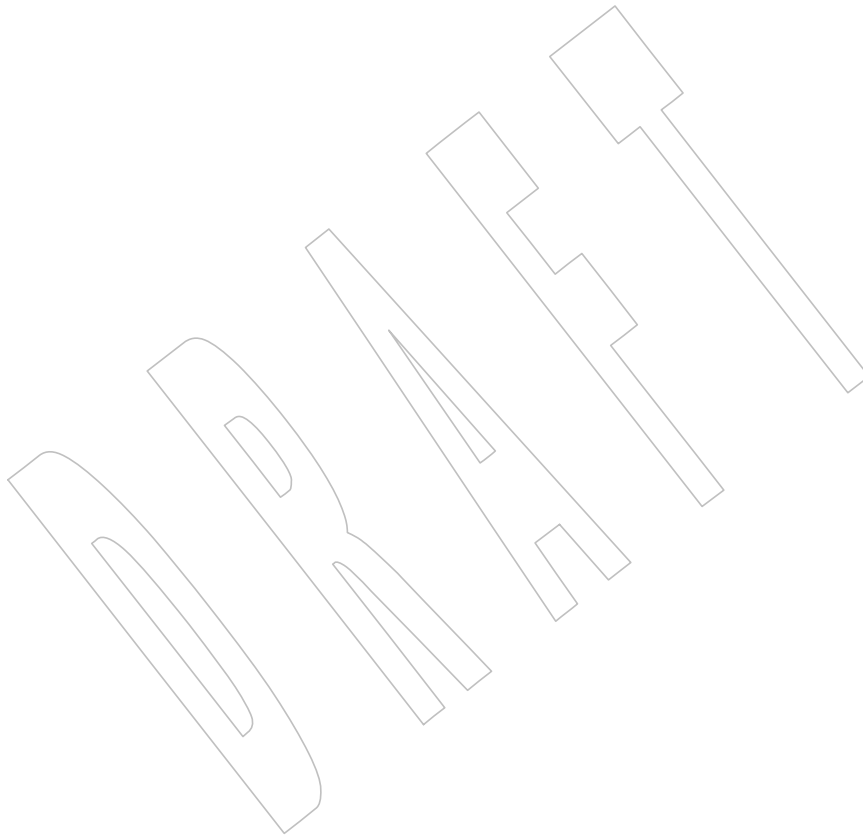
pthread_barrierattr_destroy()33793
33794
33795
33796
33797
33798**CHANGE HISTORY**

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.

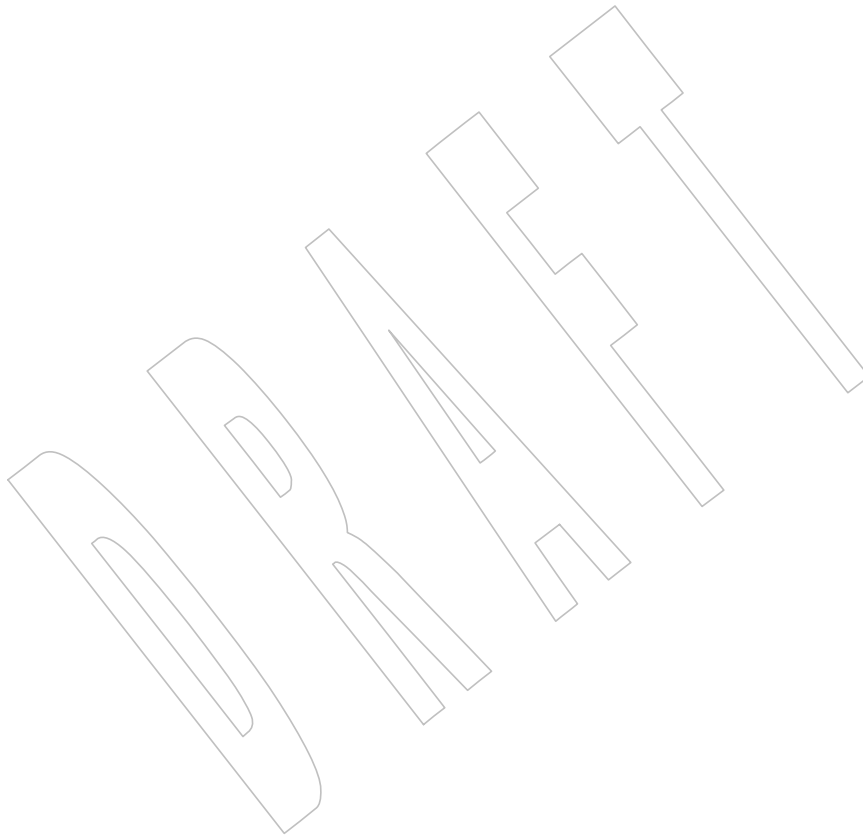
Issue 7

The `pthread_barrierattr_destroy()` and `pthread_barrierattr_init()` functions are moved from the Barriers option to the Base.



NAME

pthread_barrierattr_getpshared, pthread_barrierattr_setpshar



pthread_barrierattr_getpshared()*System Interfaces*

33835

EXAMPLES

33836

None.

33837

APPLICATION USAGE

33838

The *pthread_barrierattr_getpshared()* and *pthread_barrierattr_setpshared()* functions are part of the Thread Process-Shared Synchronization option and need not be provided on all implementations.

33839

33840

33841

RATIONALE

33842

None.

33843

FUTURE DIRECTIONS

33844

None.

33845

SEE ALSO

33846

pthread_barrier_destroy(), *pthread_barrierattr_destroy()*, *pthread_barrierattr_init()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

33847

33848

CHANGE HISTORY

33849

First released in Issue 6. Derived from IEEE Std 1003.1j-2000

33850

Issue 7

33851

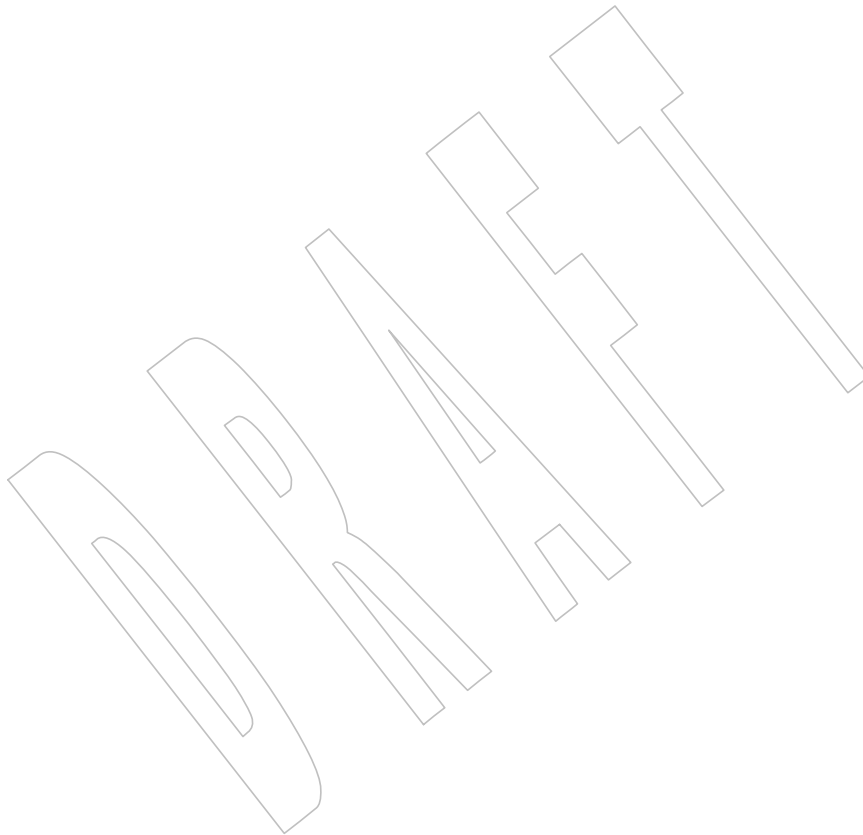
The *pthread_barrierattr_getpshared()* and *pthread_barrierattr_setpshared()* functions are moved from the Barriers option.

33852

33853 **NAME**
33854 pthread_barrierattr_init — initialize the barrier attributes object

33855 **SYNOPSIS**
33856 #include <pthread.h>
33857 int pthread_barrierattr_init(pthread_barrierattr_t *attr);

33858 **DESCRIPTION**
33859 Refer to *pthread_barrierattr_destroy()*.



pthread_barrierattr_setpshared()*System Interfaces*

33860 **NAME**
33861 pthread_barrierattr_setpshared — set the process-shared attribute of the barrier attributes object

SYNOPSIS

```
33863 TSH #include <pthread.h>  
33864 int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,  
33865 int pshared);
```

DESCRIPTION

33866 Refer to [pthread_barrierattr_getpshared\(\)](#).
33867

33868 **NAME**
 33869 pthread_cancel — cancel execution of a thread

33870 **SYNOPSIS**
 33871 #include <pthread.h>
 33872 int pthread_cancel(pthread_t thread);

33873 **DESCRIPTION**
 33874 The *pthread_cancel()* function shall request that *thread* be canceled. The target thread's
 33875 cancelability state and type determines when the cancellation takes effect. When the cancellation
 33876 is acted on, the cancellation cleanup handlers for *thread* shall be called. When the last
 33877 cancellation cleanup handler returns, the thread-specific data destructor functions shall be called
 33878 for *thread*. When the last destructor function returns, *thread* shall be terminated.

33879 The cancellation processing in the target thread shall run asynchronously with respect to the
 33880 calling thread returning from *pthread_cancel()*.

33881 **RETURN VALUE**
 33882 If successful, the *pthread_cancel()* function shall return zero; otherwise, an error number shall be
 33883 returned to indicate the error.

33884 **ERRORS**
 33885 The *pthread_cancel()* function may fail if:
 33886 [ESRCH] No thread could be found corresponding to that specified by the given thread
 33887 ID.
 33888 The *pthread_cancel()* function shall not return an error code of [EINTR].

33889 **EXAMPLES**
 33890 None.

33891 **APPLICATION USAGE**
 33892 None.

33893 **RATIONALE**
 33894 Two alternative functions were considered for sending the cancellation notification to a thread.
 33895 One would be to define a new SIGCANCEL signal that had the cancellation semantics when
 33896 delivered; the other was to define the new *pthread_cancel()* function, which would trigger the
 33897 cancellation semantics.

33898 The advantage of a new signal was that so much of the delivery criteria were identical to that
 33899 used when trying to deliver a signal that making cancellation notification a signal was seen as
 33900 consistent. Indeed, many implementations implement cancellation using a special signal. On the
 33901 other hand, there would be no signal functions that could be used with this signal except
 33902 *pthread_kill()*, and the behavior of the delivered cancellation signal would be unlike any
 33903 previously existing defined signal.

33904 The benefits of a special function include the recognition that this signal would be defined
 33905 because of the similar delivery criteria and that this is the only common behavior between a
 33906 cancellation request and a signal. In addition, the cancellation delivery mechanism does not
 33907 have to be implemented as a signal. There are also strong, if not stronger, parallels with
 33908 language exception mechanisms than with signals that are potentially obscured if the delivery
 33909 mechanism is visibly closer to signals.

33910 In the end, it was considered that as there were so many exceptions to the use of the new signal
 33911 with existing signals functions it would be misleading. A special function has resolved this
 33912 problem. This function was carefully defined so that an implementation wishing to provide the

pthread_cancel()*System Interfaces*

33913 cancellation functions on top of signals could do so. The special function also means that
33914 implementations are not obliged to implement cancellation with signals.

FUTURE DIRECTIONS

33915 None.
33916

SEE ALSO

33917 *pthread_exit()*, *pthread_cond_timedwait()*, *pthread_join()*, *pthread_setcancelstate()*, the Base
33918 Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**
33919

CHANGE HISTORY

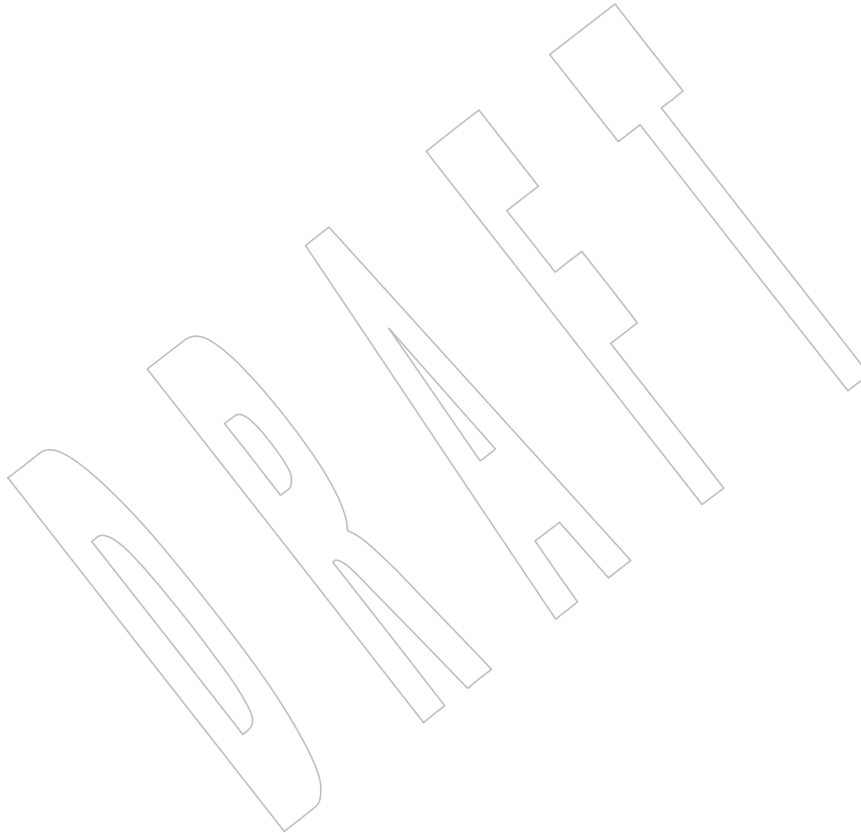
33920 First released in Issue 5. Included for alignment with the POSIX Threads Extension.
33921

Issue 6

33922 The *pthread_cancel()* function is marked as part of the Threads option.
33923

Issue 7

33924 The *pthread_cancel()* function is moved from the Threads option to the Base.
33925



33926 **NAME**

33927 pthread_cleanup_pop, pthread_cleanup_push — establish cancellation handlers

33928 **SYNOPSIS**

33929 #include <pthread.h>

33930 void pthread_cleanup_pop(int execute);

33931 void pthread_cleanup_push(void (*routine)(void*), void *arg);

33932 **DESCRIPTION**33933 The *pthread_cleanup_pop()* function shall remove the routine at the top of the calling thread's
33934 cancellation cleanup stack and optionally invoke it (if *execute* is non-zero).33935 The *pthread_cleanup_push()* function shall push the specified cancellation cleanup handler *routine*
33936 onto the calling thread's cancellation cleanup stack. The cancellation cleanup handler shall be
33937 popped from the cancellation cleanup stack and invoked with the argument *arg* when:

- 33938 • The thread exits (that is, calls *pthread_exit()*).
- 33939 • The thread acts upon a cancellation request.
- 33940 • The thread calls *pthread_cleanup_pop()* with a non-zero *execute* argument.

33941 These functions may be implemented as macros. The application shall ensure that they appear
33942 as statements, and in pairs within the same lexical scope (that is, the *pthread_cleanup_push()*
33943 macro may be thought to expand to a token list whose first token is '{' with
33944 *pthread_cleanup_pop()* expanding to a token list whose last token is the corresponding '}').

33945 The effect of calling *longjmp()* or *siglongjmp()* is undefined if there have been any calls to
33946 *pthread_cleanup_push()* or *pthread_cleanup_pop()* made without the matching call since the jump
33947 buffer was filled. The effect of calling *longjmp()* or *siglongjmp()* from inside a cancellation
33948 cleanup handler is also undefined unless the jump buffer was also filled in the cancellation
33949 cleanup handler.

33950 The effect of the use of **return**, **break**, **continue**, and **goto** to prematurely leave a code block
33951 described by a pair of *pthread_cleanup_push()* and *pthread_cleanup_pop()* functions calls is
33952 undefined.

33953 **RETURN VALUE**33954 The *pthread_cleanup_push()* and *pthread_cleanup_pop()* functions shall not return a value.33955 **ERRORS**

33956 No errors are defined.

33957 These functions shall not return an error code of [EINTR].

33958 **EXAMPLES**33959 The following is an example using thread primitives to implement a cancelable, writers-priority
33960 read-write lock:

```

33961 typedef struct {
33962     pthread_mutex_t lock;
33963     pthread_cond_t rcond,
33964     wcond;
33965     int lock_count; /* < 0 .. Held by writer. */
33966                   /* > 0 .. Held by lock_count readers. */
33967                   /* = 0 .. Held by nobody. */
33968     int waiting_writers; /* Count of waiting writers. */
33969 } rwlock;

```

```

33970     void
33971     waiting_reader_cleanup(void *arg)
33972     {
33973         rwlock *l;
33974
33975         l = (rwlock *) arg;
33976         pthread_mutex_unlock(&l->lock);
33977     }
33978
33979     void
33980     lock_for_read(rwlock *l)
33981     {
33982         pthread_mutex_lock(&l->lock);
33983         pthread_cleanup_push(waiting_reader_cleanup, l);
33984         while ((l->lock_count < 0) && (l->waiting_writers != 0))
33985             pthread_cond_wait(&l->rcond, &l->lock);
33986         l->lock_count++;
33987         /*
33988          * Note the pthread_cleanup_pop executes
33989          * waiting_reader_cleanup.
33990          */
33991         pthread_cleanup_pop(1);
33992     }
33993
33994     void
33995     release_read_lock(rwlock *l)
33996     {
33997         pthread_mutex_lock(&l->lock);
33998         if (--l->lock_count == 0)
33999             pthread_cond_signal(&l->wcond);
34000         pthread_mutex_unlock(l);
34001     }
34002
34003     void
34004     waiting_writer_cleanup(void *arg)
34005     {
34006         rwlock *l;
34007
34008         l = (rwlock *) arg;
34009         if ((--l->waiting_writers == 0) && (l->lock_count >= 0)) {
34010             /*
34011              * This only happens if we have been canceled.
34012              */
34013             pthread_cond_broadcast(&l->wcond);
34014         }
34015         pthread_mutex_unlock(&l->lock);
34016     }
34017
34018     void
34019     lock_for_write(rwlock *l)
34020     {
34021         pthread_mutex_lock(&l->lock);
34022         l->waiting_writers++;
34023         pthread_cleanup_push(waiting_writer_cleanup, l);
34024         while (l->lock_count != 0)
34025             pthread_cond_wait(&l->wcond, &l->lock);
34026         l->lock_count = -1;

```

```

34021     /*
34022     * Note the pthread_cleanup_pop executes
34023     * waiting_writer_cleanup.
34024     */
34025     pthread_cleanup_pop(1);
34026 }
34027
34028 void
34029 release_write_lock(rwlock *l)
34030 {
34031     pthread_mutex_lock(&l->lock);
34032     l->lock_count = 0;
34033     if (l->waiting_writers == 0)
34034         pthread_cond_broadcast(&l->rcond)
34035     else
34036         pthread_cond_signal(&l->wcond);
34037     pthread_mutex_unlock(&l->lock);
34038 }
34039 /*
34040 * This function is called to initialize the read/write lock.
34041 */
34042 void
34043 initialize_rwlock(rwlock *l)
34044 {
34045     pthread_mutex_init(&l->lock, pthread_mutexattr_default);
34046     pthread_cond_init(&l->wcond, pthread_condattr_default);
34047     pthread_cond_init(&l->rcond, pthread_condattr_default);
34048     l->lock_count = 0;
34049     l->waiting_writers = 0;
34050 }
34051
34052 reader_thread()
34053 {
34054     lock_for_read(&lock);
34055     pthread_cleanup_push(release_read_lock, &lock);
34056     /*
34057     * Thread has read lock.
34058     */
34059     pthread_cleanup_pop(1);
34060 }
34061
34062 writer_thread()
34063 {
34064     lock_for_write(&lock);
34065     pthread_cleanup_push(release_write_lock, &lock);
34066     /*
34067     * Thread has write lock.
34068     */
34069     pthread_cleanup_pop(1);
34070 }

```

APPLICATION USAGE

The two routines that push and pop cancellation cleanup handlers, *pthread_cleanup_push()* and *pthread_cleanup_pop()*, can be thought of as left and right parentheses. They always need to be matched.

RATIONALE

The restriction that the two routines that push and pop cancellation cleanup handlers, *pthread_cleanup_push()* and *pthread_cleanup_pop()*, have to appear in the same lexical scope allows for efficient macro or compiler implementations and efficient storage management. A sample implementation of these routines as macros might look like this:

```
#define pthread_cleanup_push(rtn,arg) { \
    struct _pthread_handler_rec __cleanup_handler, **__head; \
    __cleanup_handler.rtn = rtn; \
    __cleanup_handler.arg = arg; \
    (void) pthread_getspecific(_pthread_handler_key, &__head); \
    __cleanup_handler.next = *__head; \
    *__head = &__cleanup_handler;
#define pthread_cleanup_pop(ex) \
    *__head = __cleanup_handler.next; \
    if (ex) (*__cleanup_handler.rtn)(__cleanup_handler.arg); \
}
```

A more ambitious implementation of these routines might do even better by allowing the compiler to note that the cancellation cleanup handler is a constant and can be expanded inline.

This volume of IEEE Std 1003.1-200x currently leaves unspecified the effect of calling *longjmp()* from a signal handler executing in a POSIX System Interfaces function. If an implementation wants to allow this and give the programmer reasonable behavior, the *longjmp()* function has to call all cancellation cleanup handlers that have been pushed but not popped since the time *setjmp()* was called.

Consider a multi-threaded function called by a thread that uses signals. If a signal were delivered to a signal handler during the operation of *qsort()* and that handler were to call *longjmp()* (which, in turn, did *not* call the cancellation cleanup handlers), the helper threads created by the *qsort()* function would not be canceled. Instead, they would continue to execute and write into the argument array even though the array might have been popped off the stack.

Note that the specified cleanup handling mechanism is especially tied to the C language and, while the requirement for a uniform mechanism for expressing cleanup is language-independent, the mechanism used in other languages may be quite different. In addition, this mechanism is really only necessary due to the lack of a real exception mechanism in the C language, which would be the ideal solution.

There is no notion of a cancellation cleanup-safe function. If an application has no cancellation points in its signal handlers, blocks any signal whose handler may have cancellation points while calling async-unsafe functions, or disables cancellation while calling async-unsafe functions, all functions may be safely called from cancellation cleanup routines.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cancel(), *pthread_setcancelstate()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

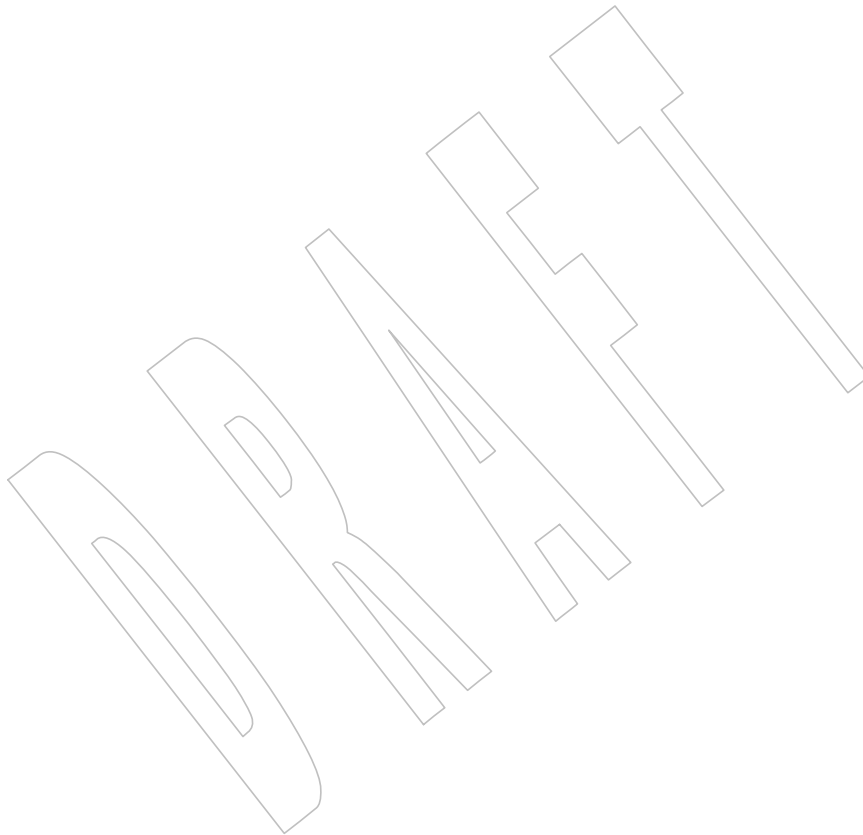
First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_cleanup_pop()* and *pthread_cleanup_push()* functions are marked as part of the Threads option.

The APPLICATION USAGE section is added.

- 34120 The normative text is updated to avoid use of the term “must” for application requirements.
- 34121 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/88 is applied, updating the
- 34122 DESCRIPTION to describe the consequences of prematurely leaving a code block defined by the
- 34123 *pthread_cleanup_push()* and *pthread_cleanup_pop()* functions.
- 34124 **Issue 7**
- 34125 The *pthread_cleanup_pop()* and *pthread_cleanup_push()* functions are moved from the Threads
- 34126 option to the Base.



34127 **NAME**

34128 pthread_cond_broadcast, pthread_cond_signal — broadcast or signal a condition

34129 **SYNOPSIS**

34130 #include <pthread.h>

34131 int pthread_cond_broadcast(pthread_cond_t *cond);

34132 int pthread_cond_signal(pthread_cond_t *cond);

34133 **DESCRIPTION**

34134 These functions shall unblock threads blocked on a condition variable.

34135 The *pthread_cond_broadcast()* function shall unblock all threads currently blocked on the
34136 specified condition variable *cond*.34137 The *pthread_cond_signal()* function shall unblock at least one of the threads that are blocked on
34138 the specified condition variable *cond* (if any threads are blocked on *cond*).34139 If more than one thread is blocked on a condition variable, the scheduling policy shall determine
34140 the order in which threads are unblocked. When each thread unblocked as a result of a
34141 *pthread_cond_broadcast()* or *pthread_cond_signal()* returns from its call to *pthread_cond_wait()* or
34142 *pthread_cond_timedwait()*, the thread shall own the mutex with which it called
34143 *pthread_cond_wait()* or *pthread_cond_timedwait()*. The thread(s) that are unblocked shall contend
34144 for the mutex according to the scheduling policy (if applicable), and as if each had called
34145 *pthread_mutex_lock()*.34146 The *pthread_cond_broadcast()* or *pthread_cond_signal()* functions may be called by a thread
34147 whether or not it currently owns the mutex that threads calling *pthread_cond_wait()* or
34148 *pthread_cond_timedwait()* have associated with the condition variable during their waits;
34149 however, if predictable scheduling behavior is required, then that mutex shall be locked by the
34150 thread calling *pthread_cond_broadcast()* or *pthread_cond_signal()*.34151 The *pthread_cond_broadcast()* and *pthread_cond_signal()* functions shall have no effect if there are
34152 no threads currently blocked on *cond*.34153 **RETURN VALUE**34154 If successful, the *pthread_cond_broadcast()* and *pthread_cond_signal()* functions shall return zero;
34155 otherwise, an error number shall be returned to indicate the error.34156 **ERRORS**34157 The *pthread_cond_broadcast()* and *pthread_cond_signal()* function may fail if:34158 [EINVAL] The value *cond* does not refer to an initialized condition variable.

34159 These functions shall not return an error code of [EINTR].

34160 **EXAMPLES**

34161 None.

34162 **APPLICATION USAGE**34163 The *pthread_cond_broadcast()* function is used whenever the shared-variable state has been
34164 changed in a way that more than one thread can proceed with its task. Consider a single
34165 producer/multiple consumer problem, where the producer can insert multiple items on a list
34166 that is accessed one item at a time by the consumers. By calling the *pthread_cond_broadcast()*
34167 function, the producer would notify all consumers that might be waiting, and thereby the
34168 application would receive more throughput on a multi-processor. In addition,
34169 *pthread_cond_broadcast()* makes it easier to implement a read-write lock. The
34170 *pthread_cond_broadcast()* function is needed in order to wake up all waiting readers when a
34171 writer releases its lock. Finally, the two-phase commit algorithm can use this broadcast function

34172 to notify all clients of an impending transaction commit.

34173 It is not safe to use the *pthread_cond_signal()* function in a signal handler that is invoked
34174 asynchronously. Even if it were safe, there would still be a race between the test of the Boolean
34175 *pthread_cond_wait()* that could not be efficiently eliminated.

34176 Mutexes and condition variables are thus not suitable for releasing a waiting thread by signaling
34177 from code running in a signal handler.

34178 RATIONALE

34179 Multiple Awakenings by Condition Signal

34180 On a multi-processor, it may be impossible for an implementation of *pthread_cond_signal()* to
34181 avoid the unblocking of more than one thread blocked on a condition variable. For example,
34182 consider the following partial implementation of *pthread_cond_wait()* and *pthread_cond_signal()*,
34183 executed by two threads in the order given. One thread is trying to wait on the condition
34184 variable, another is concurrently executing *pthread_cond_signal()*, while a third thread is already
34185 waiting.

```

34186 pthread_cond_wait(mutex, cond):
34187     value = cond->value; /* 1 */
34188     pthread_mutex_unlock(mutex); /* 2 */
34189     pthread_mutex_lock(cond->mutex); /* 10 */
34190     if (value == cond->value) { /* 11 */
34191         me->next_cond = cond->waiter;
34192         cond->waiter = me;
34193         pthread_mutex_unlock(cond->mutex);
34194         unable_to_run(me);
34195     } else
34196         pthread_mutex_unlock(cond->mutex); /* 12 */
34197     pthread_mutex_lock(mutex); /* 13 */

34198 pthread_cond_signal(cond):
34199     pthread_mutex_lock(cond->mutex); /* 3 */
34200     cond->value++; /* 4 */
34201     if (cond->waiter) { /* 5 */
34202         sleeper = cond->waiter; /* 6 */
34203         cond->waiter = sleeper->next_cond; /* 7 */
34204         able_to_run(sleeper); /* 8 */
34205     }
34206     pthread_mutex_unlock(cond->mutex); /* 9 */

```

34207 The effect is that more than one thread can return from its call to *pthread_cond_wait()* or
34208 *pthread_cond_timedwait()* as a result of one call to *pthread_cond_signal()*. This effect is called
34209 “spurious wakeup”. Note that the situation is self-correcting in that the number of threads that
34210 are so awakened is finite; for example, the next thread to call *pthread_cond_wait()* after the
34211 sequence of events above blocks.

34212 While this problem could be resolved, the loss of efficiency for a fringe condition that occurs
34213 only rarely is unacceptable, especially given that one has to check the predicate associated with a
34214 condition variable anyway. Correcting this problem would unnecessarily reduce the degree of
34215 concurrency in this basic building block for all higher-level synchronization operations.

34216 An added benefit of allowing spurious wakeups is that applications are forced to code a
34217 predicate-testing-loop around the condition wait. This also makes the application tolerate
34218 superfluous condition broadcasts or signals on the same condition variable that may be coded in
34219 some other part of the application. The resulting applications are thus more robust. Therefore,
34220 IEEE Std 1003.1-200x explicitly documents that spurious wakeups may occur.

34221
34222
34223
34224
34225
34226
34227
34228
34229
34230
34231
34232
34233
34234**FUTURE DIRECTIONS**

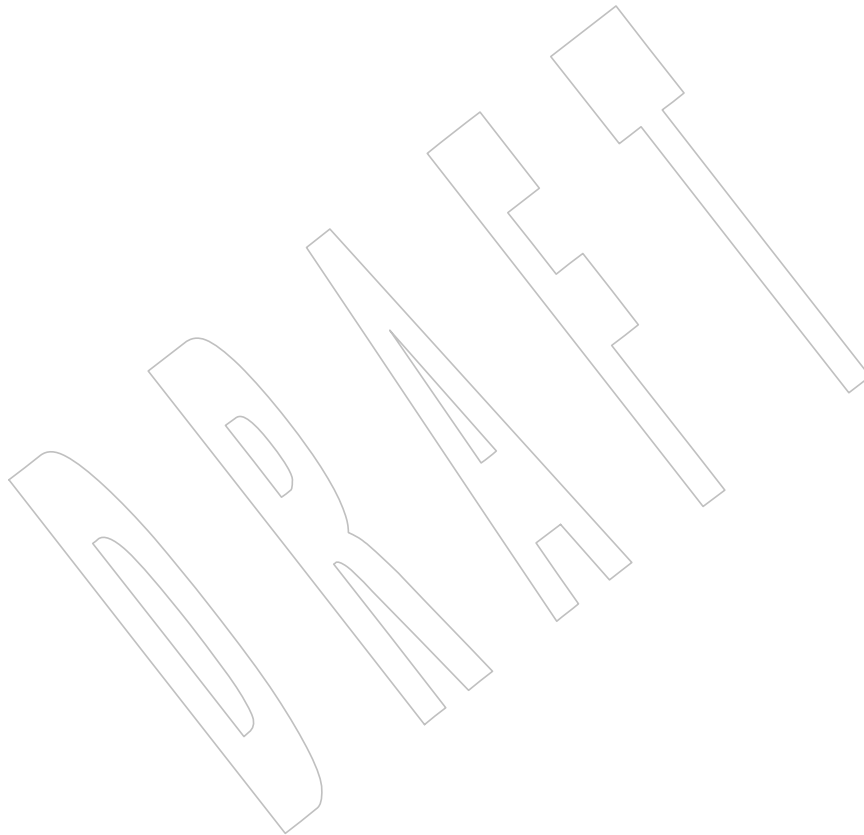
None.

SEE ALSO*pthread_cond_destroy()*, *pthread_cond_timedwait()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, <pthread.h>**CHANGE HISTORY**

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6The *pthread_cond_broadcast()* and *pthread_cond_signal()* functions are marked as part of the Threads option.

The APPLICATION USAGE section is added.

Issue 7The *pthread_cond_broadcast()* and *pthread_cond_signal()* functions are moved from the Threads option to the Base.

34235 **NAME**

34236 pthread_cond_destroy, pthread_cond_init — destroy and initialize condition variables

34237 **SYNOPSIS**

```
34238 #include <pthread.h>
34239
34239 int pthread_cond_destroy(pthread_cond_t *cond);
34240 int pthread_cond_init(pthread_cond_t *restrict cond,
34241     const pthread_condattr_t *restrict attr);
34242 pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

34243 **DESCRIPTION**

34244 The *pthread_cond_destroy()* function shall destroy the given condition variable specified by *cond*;
 34245 the object becomes, in effect, uninitialized. An implementation may cause *pthread_cond_destroy()*
 34246 to set the object referenced by *cond* to an invalid value. A destroyed condition variable object can
 34247 be reinitialized using *pthread_cond_init()*; the results of otherwise referencing the object after it
 34248 has been destroyed are undefined.

34249 It shall be safe to destroy an initialized condition variable upon which no threads are currently
 34250 blocked. Attempting to destroy a condition variable upon which other threads are currently
 34251 blocked results in undefined behavior.

34252 The *pthread_cond_init()* function shall initialize the condition variable referenced by *cond* with
 34253 attributes referenced by *attr*. If *attr* is NULL, the default condition variable attributes shall be
 34254 used; the effect is the same as passing the address of a default condition variable attributes
 34255 object. Upon successful initialization, the state of the condition variable shall become initialized.

34256 Only *cond* itself may be used for performing synchronization. The result of referring to copies of
 34257 *cond* in calls to *pthread_cond_wait()*, *pthread_cond_timedwait()*, *pthread_cond_signal()*,
 34258 *pthread_cond_broadcast()*, and *pthread_cond_destroy()* is undefined.

34259 Attempting to initialize an already initialized condition variable results in undefined behavior.

34260 In cases where default condition variable attributes are appropriate, the macro
 34261 PTHREAD_COND_INITIALIZER can be used to initialize condition variables that are statically
 34262 allocated. The effect shall be equivalent to dynamic initialization by a call to *pthread_cond_init()*
 34263 with parameter *attr* specified as NULL, except that no error checks are performed.

34264 **RETURN VALUE**

34265 If successful, the *pthread_cond_destroy()* and *pthread_cond_init()* functions shall return zero;
 34266 otherwise, an error number shall be returned to indicate the error.

34267 The [EBUSY] and [EINVAL] error checks, if implemented, shall act as if they were performed
 34268 immediately at the beginning of processing for the function and caused an error return prior to
 34269 modifying the state of the condition variable specified by *cond*.

34270 **ERRORS**

34271 The *pthread_cond_destroy()* function may fail if:

34272 [EBUSY] The implementation has detected an attempt to destroy the object referenced
 34273 by *cond* while it is referenced (for example, while being used in a
 34274 *pthread_cond_wait()* or *pthread_cond_timedwait()*) by another thread.

34275 [EINVAL] The value specified by *cond* is invalid.

34276 The *pthread_cond_init()* function shall fail if:

- 34277 [EAGAIN] The system lacked the necessary resources (other than memory) to initialize
34278 another condition variable.
- 34279 [ENOMEM] Insufficient memory exists to initialize the condition variable.
- 34280 The `pthread_cond_init()` function may fail if:
- 34281 [EBUSY] The implementation has detected an attempt to reinitialize the object
34282 referenced by `cond`, a previously initialized, but not yet destroyed, condition
34283 variable.
- 34284 [EINVAL] The value specified by `attr` is invalid.
- 34285 These functions shall not return an error code of [EINTR].

EXAMPLES

34286 A condition variable can be destroyed immediately after all the threads that are blocked on it are
34287 awakened. For example, consider the following code:

```

34289 struct list {
34290     pthread_mutex_t lm;
34291     ...
34292 }
34293 struct elt {
34294     key k;
34295     int busy;
34296     pthread_cond_t notbusy;
34297     ...
34298 }
34299 /* Find a list element and reserve it. */
34300 struct elt *
34301 list_find(struct list *lp, key k)
34302 {
34303     struct elt *ep;
34304     pthread_mutex_lock(&lp->lm);
34305     while ((ep = find_elt(l, k) != NULL) && ep->busy)
34306         pthread_cond_wait(&ep->notbusy, &lp->lm);
34307     if (ep != NULL)
34308         ep->busy = 1;
34309     pthread_mutex_unlock(&lp->lm);
34310     return(ep);
34311 }
34312 delete_elt(struct list *lp, struct elt *ep)
34313 {
34314     pthread_mutex_lock(&lp->lm);
34315     assert(ep->busy);
34316     ... remove ep from list ...
34317     ep->busy = 0; /* Paranoid. */
34318     (A) pthread_cond_broadcast(&ep->notbusy);
34319         pthread_mutex_unlock(&lp->lm);
34320     (B) pthread_cond_destroy(&rp->notbusy);
34321         free(ep);
34322 }

```

34323 In this example, the condition variable and its list element may be freed (line B) immediately
34324 after all threads waiting for it are awakened (line A), since the mutex and the code ensure that

34325 no other thread can touch the element to be deleted.

34326 **APPLICATION USAGE**

34327 None.

34328 **RATIONALE**

34329 See *pthread_mutex_init()*; a similar rationale applies to condition variables.

34330 **FUTURE DIRECTIONS**

34331 None.

34332 **SEE ALSO**

34333 *pthread_cond_broadcast()*, *pthread_cond_signal()*, *pthread_cond_timedwait()*, the Base Definitions
34334 volume of IEEE Std 1003.1-200x, <pthread.h>

34335 **CHANGE HISTORY**

34336 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34337 **Issue 6**

34338 The *pthread_cond_destroy()* and *pthread_cond_init()* functions are marked as part of the Threads
34339 option.

34340 IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION.

34341 The **restrict** keyword is added to the *pthread_cond_init()* prototype for alignment with the
34342 ISO/IEC 9899:1999 standard.

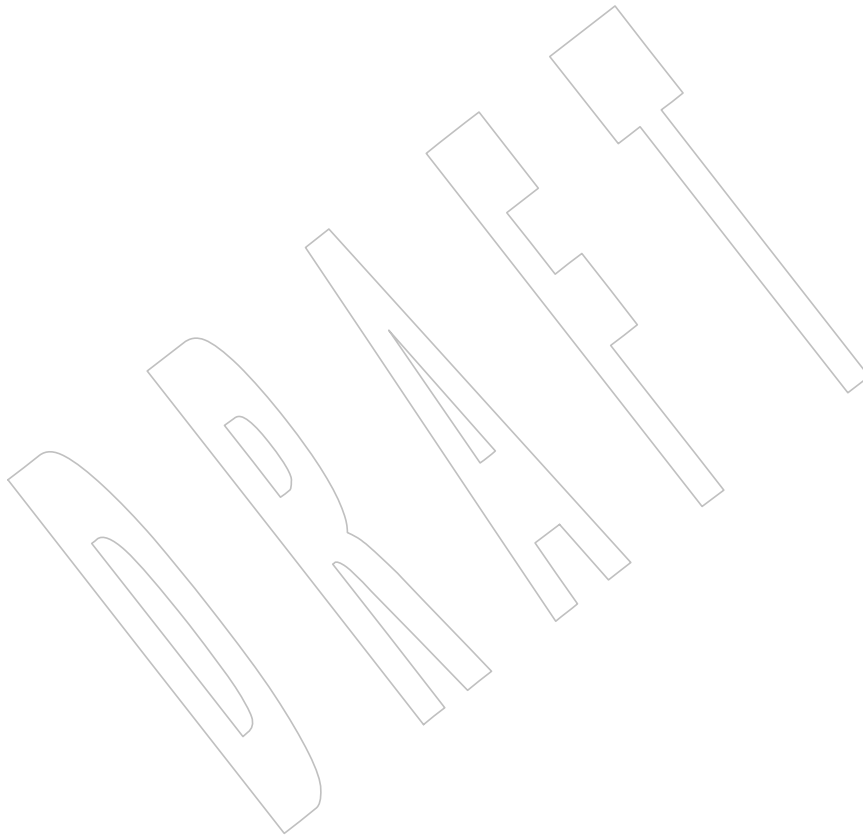
34343 **Issue 7**

34344 The *pthread_cond_destroy()* and *pthread_cond_init()* functions are moved from the Threads option
34345 to the Base.

34346 **NAME**
34347 pthread_cond_signal — signal a condition

34348 **SYNOPSIS**
34349 #include <pthread.h>
34350 int pthread_cond_signal(pthread_cond_t *cond);

34351 **DESCRIPTION**
34352 Refer to *pthread_cond_broadcast()*.



34353 **NAME**
 34354 pthread_cond_timedwait, pthread_cond_wait — wait on a condition

34355 **SYNOPSIS**
 34356 #include <pthread.h>
 34357 int pthread_cond_timedwait(pthread_cond_t *restrict cond,
 34358 pthread_mutex_t *restrict mutex,
 34359 const struct timespec *restrict abstime);
 34360 int pthread_cond_wait(pthread_cond_t *restrict cond,
 34361 pthread_mutex_t *restrict mutex);

34362 **DESCRIPTION**
 34363 The *pthread_cond_timedwait()* and *pthread_cond_wait()* functions shall block on a condition
 34364 variable. They shall be called with *mutex* locked by the calling thread or undefined behavior
 34365 results.

34366 These functions atomically release *mutex* and cause the calling thread to block on the condition
 34367 variable *cond*; atomically here means “atomically with respect to access by another thread to the
 34368 mutex and then the condition variable”. That is, if another thread is able to acquire the mutex
 34369 after the about-to-block thread has released it, then a subsequent call to *pthread_cond_broadcast()*
 34370 or *pthread_cond_signal()* in that thread shall behave as if it were issued after the about-to-block
 34371 thread has blocked.

34372 Upon successful return, the mutex shall have been locked and shall be owned by the calling
 34373 thread. If *mutex* is a robust mutex where an owner terminated while holding the lock and the
 34374 state is recoverable, the mutex shall be acquired even though the function returns an error code.

34375 When using condition variables there is always a Boolean predicate involving shared variables
 34376 associated with each condition wait that is true if the thread should proceed. Spurious wakeups
 34377 from the *pthread_cond_timedwait()* or *pthread_cond_wait()* functions may occur. Since the return
 34378 from *pthread_cond_timedwait()* or *pthread_cond_wait()* does not imply anything about the value of
 34379 this predicate, the predicate should be re-evaluated upon such return.

34380 When a thread waits on a condition variable, having specified a particular mutex to either the
 34381 *pthread_cond_timedwait()* or the *pthread_cond_wait()* operation, a dynamic binding is formed
 34382 between that mutex and condition variable that remains in effect as long as at least one thread is
 34383 blocked on the condition variable. During this time, the effect of an attempt by any thread to
 34384 wait on that condition variable using a different mutex is undefined. Once all waiting threads
 34385 have been unblocked (as by the *pthread_cond_broadcast()* operation), the next wait operation on
 34386 that condition variable shall form a new dynamic binding with the mutex specified by that wait
 34387 operation. Even though the dynamic binding between condition variable and mutex may be
 34388 removed or replaced between the time a thread is unblocked from a wait on the condition
 34389 variable and the time that it returns to the caller or begins cancellation cleanup, the unblocked
 34390 thread shall always re-acquire the mutex specified in the condition wait operation call from
 34391 which it is returning.

34392 A condition wait (whether timed or not) is a cancellation point. When the cancelability type of a
 34393 thread is set to *PTHREAD_CANCEL_DEFERRED*, a side effect of acting upon a cancellation
 34394 request while in a condition wait is that the mutex is (in effect) re-acquired before calling the first
 34395 cancellation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up
 34396 to the point of returning from the call to *pthread_cond_timedwait()* or *pthread_cond_wait()*, but at
 34397 that point notices the cancellation request and instead of returning to the caller of
 34398 *pthread_cond_timedwait()* or *pthread_cond_wait()*, starts the thread cancellation activities, which
 34399 includes calling cancellation cleanup handlers.

34400 A thread that has been unblocked because it has been canceled while blocked in a call to
 34401 *pthread_cond_timedwait()* or *pthread_cond_wait()* shall not consume any condition signal that may
 34402 be directed concurrently at the condition variable if there are other threads blocked on the
 34403 condition variable.

34404 The *pthread_cond_timedwait()* function shall be equivalent to *pthread_cond_wait()*, except that an
 34405 error is returned if the absolute time specified by *abstime* passes (that is, system time equals or
 34406 exceeds *abstime*) before the condition *cond* is signaled or broadcasted, or if the absolute time
 34407 specified by *abstime* has already been passed at the time of the call.

34408 The condition variable shall have a clock attribute which specifies the clock that shall be used to
 34409 measure the time specified by the *abstime* argument. When such timeouts occur,
 34410 *pthread_cond_timedwait()* shall nonetheless release and re-acquire the mutex referenced by *mutex*.
 34411 The *pthread_cond_timedwait()* function is also a cancellation point.

34412 If a signal is delivered to a thread waiting for a condition variable, upon return from the signal
 34413 handler the thread resumes waiting for the condition variable as if it was not interrupted, or it
 34414 shall return zero due to spurious wakeup.

34415 RETURN VALUE

34416 Except in the case of [ETIMEDOUT], all these error checks shall act as if they were performed
 34417 immediately at the beginning of processing for the function and shall cause an error return, in
 34418 effect, prior to modifying the state of the mutex specified by *mutex* or the condition variable
 34419 specified by *cond*.

34420 Upon successful completion, a value of zero shall be returned; otherwise, an error number shall
 34421 be returned to indicate the error.

34422 ERRORS

34423 The *pthread_cond_timedwait()* function shall fail if:

34424 [ETIMEDOUT] The time specified by *abstime* to *pthread_cond_timedwait()* has passed.

34425 [EINVAL] The *abstime* argument specified a nanosecond value less than zero or greater
 34426 than or equal to 1000 million.

34427 [ENOTRECOVERABLE]
 34428 The state protected by the mutex is not recoverable. The mutex is not locked.

34429 [EOWNERDEAD]
 34430 The mutex is a robust mutex and the process containing the previous owner
 34431 thread terminated while holding the mutex lock. The mutex lock has been
 34432 acquired and it is up to the new owner to make the state consistent.

34433 These functions may fail if:

34434 [EINVAL] The value specified by *cond* or *mutex* is invalid.

34435 [EOWNERDEAD]
 34436 The mutex is a robust mutex and the previous owning thread terminated
 34437 while holding the mutex lock. The mutex lock has been acquired and it is up
 34438 to the new owner to make the state consistent.

34439 [EPERM] The mutex was not owned by the current thread at the time of the call.

34440 These functions shall not return an error code of [EINTR].

EXAMPLES

34441
34442 None.

APPLICATION USAGE

34443 Applications that have assumed that non-zero return values are errors will need updating for
34444 use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting
34445 a currently inconsistent state is [EOWNERDEAD]. Applications that do not check the error
34446 returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If
34447 an application is supposed to work with normal and robust mutexes, it should check all return
34448 values for error conditions and if necessary take appropriate action.
34449

RATIONALE**Condition Wait Semantics**

34451
34452 It is important to note that when *pthread_cond_wait()* and *pthread_cond_timedwait()* return
34453 without error, the associated predicate may still be false. Similarly, when
34454 *pthread_cond_timedwait()* returns with the timeout error, the associated predicate may be true
34455 due to an unavoidable race between the expiration of the timeout and the predicate state change.

34456 The application needs to recheck the predicate on any return because it cannot be sure there is
34457 another thread waiting on the thread to handle the signal, and if there is not then the signal is
34458 lost. The burden is on the application to check the predicate.

34459 Some implementations, particularly on a multi-processor, may sometimes cause multiple
34460 threads to wake up when the condition variable is signaled simultaneously on different
34461 processors.

34462 In general, whenever a condition wait returns, the thread has to re-evaluate the predicate
34463 associated with the condition wait to determine whether it can safely proceed, should wait
34464 again, or should declare a timeout. A return from the wait does not imply that the associated
34465 predicate is either true or false.

34466 It is thus recommended that a condition wait be enclosed in the equivalent of a “while loop”
34467 that checks the predicate.

Timed Wait Semantics

34468
34469 An absolute time measure was chosen for specifying the timeout parameter for two reasons.
34470 First, a relative time measure can be easily implemented on top of a function that specifies
34471 absolute time, but there is a race condition associated with specifying an absolute timeout on top
34472 of a function that specifies relative timeouts. For example, assume that *clock_gettime()* returns
34473 the current time and *cond_relative_timed_wait()* uses relative timeouts:

```
34474 clock_gettime(CLOCK_REALTIME, &now)
34475 reltime = sleep_til_this_absolute_time -now;
34476 cond_relative_timed_wait(c, m, &reltime);
```

34477 If the thread is preempted between the first statement and the last statement, the thread blocks
34478 for too long. Blocking, however, is irrelevant if an absolute timeout is used. An absolute timeout
34479 also need not be recomputed if it is used multiple times in a loop, such as that enclosing a
34480 condition wait.

34481 For cases when the system clock is advanced discontinuously by an operator, it is expected that
34482 implementations process any timed wait expiring at an intervening time as if that time had
34483 actually occurred.

34484

Cancellation and Condition Wait

34485

34486

34487

34488

34489

34490

34491

A condition wait, whether timed or not, is a cancellation point. That is, the functions `pthread_cond_wait()` or `pthread_cond_timedwait()` are points where a pending (or concurrent) cancellation request is noticed. The reason for this is that an indefinite wait is possible at these points—whatever event is being waited for, even if the program is totally correct, might never occur; for example, some input data being awaited might never be sent. By making condition wait a cancellation point, the thread can be canceled and perform its cancellation cleanup handler even though it may be stuck in some indefinite wait.

34492

34493

34494

34495

34496

34497

34498

34499

34500

34501

A side effect of acting on a cancellation request while a thread is blocked on a condition variable is to re-acquire the mutex before calling any of the cancellation cleanup handlers. This is done in order to ensure that the cancellation cleanup handler is executed in the same state as the critical code that lies both before and after the call to the condition wait function. This rule is also required when interfacing to POSIX threads from languages, such as Ada or C++, which may choose to map cancellation onto a language exception; this rule ensures that each exception handler guarding a critical section can always safely depend upon the fact that the associated mutex has already been locked regardless of exactly where within the critical section the exception was raised. Without this rule, there would not be a uniform rule that exception handlers could follow regarding the lock, and so coding would become very cumbersome.

34502

34503

34504

Therefore, since *some* statement has to be made regarding the state of the lock when a cancellation is delivered during a wait, a definition has been chosen that makes application coding most convenient and error free.

34505

34506

34507

34508

34509

34510

When acting on a cancellation request while a thread is blocked on a condition variable, the implementation is required to ensure that the thread does not consume any condition signals directed at that condition variable if there are any other threads waiting on that condition variable. This rule is specified in order to avoid deadlock conditions that could occur if these two independent requests (one acting on a thread and the other acting on the condition variable) were not processed independently.

34511

Performance of Mutexes and Condition Variables

34512

34513

34514

Mutexes are expected to be locked only for a few instructions. This practice is almost automatically enforced by the desire of programmers to avoid long serial regions of execution (which would reduce total effective parallelism).

34515

34516

34517

34518

34519

34520

34521

34522

When using mutexes and condition variables, one tries to ensure that the usual case is to lock the mutex, access shared data, and unlock the mutex. Waiting on a condition variable should be a relatively rare situation. For example, when implementing a read-write lock, code that acquires a read-lock typically needs only to increment the count of readers (under mutual-exclusion) and return. The calling thread would actually wait on the condition variable only when there is already an active writer. So the efficiency of a synchronization operation is bounded by the cost of mutex lock/unlock and not by condition wait. Note that in the usual case there is no context switch.

34523

34524

34525

34526

This is not to say that the efficiency of condition waiting is unimportant. Since there needs to be at least one context switch per Ada rendezvous, the efficiency of waiting on a condition variable is important. The cost of waiting on a condition variable should be little more than the minimal cost for a context switch plus the time to unlock and lock the mutex.

34527

Features of Mutexes and Condition Variables34528
34529
34530
34531
34532
34533
34534
34535

It had been suggested that the mutex acquisition and release be decoupled from condition wait. This was rejected because it is the combined nature of the operation that, in fact, facilitates realtime implementations. Those implementations can atomically move a high-priority thread between the condition variable and the mutex in a manner that is transparent to the caller. This can prevent extra context switches and provide more deterministic acquisition of a mutex when the waiting thread is signaled. Thus, fairness and priority issues can be dealt with directly by the scheduling discipline. Furthermore, the current condition wait operation matches existing practice.

34536

Scheduling Behavior of Mutexes and Condition Variables34537
34538
34539
34540
34541

Synchronization primitives that attempt to interfere with scheduling policy by specifying an ordering rule are considered undesirable. Threads waiting on mutexes and condition variables are selected to proceed in an order dependent upon the scheduling policy rather than in some fixed order (for example, FIFO or priority). Thus, the scheduling policy determines which thread(s) are awakened and allowed to proceed.

34542

Timed Condition Wait34543
34544

The `pthread_cond_timedwait()` function allows an application to give up waiting for a particular condition after a given amount of time. An example of its use follows:

34545
34546
34547
34548
34549
34550
34551
34552
34553
34554

```
(void) pthread_mutex_lock(&t.mn);
    t.waiters++;
    clock_gettime(CLOCK_REALTIME, &ts);
    ts.tv_sec += 5;
    rc = 0;
    while (!mypredicate(&t) && rc == 0)
        rc = pthread_cond_timedwait(&t.cond, &t.mn, &ts);
    t.waiters--;
    if (rc == 0) setmystate(&t);
(void) pthread_mutex_unlock(&t.mn);
```

34555
34556
34557
34558
34559

By making the timeout parameter absolute, it does not need to be recomputed each time the program checks its blocking predicate. If the timeout was relative, it would have to be recomputed before each call. This would be especially difficult since such code would need to take into account the possibility of extra wakeups that result from extra broadcasts or signals on the condition variable that occur before either the predicate is true or the timeout is due.

34560

FUTURE DIRECTIONS

34561

None.

34562

SEE ALSO34563
34564

[*pthread_cond_signal\(\)*](#), [*pthread_cond_broadcast\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, [**<pthread.h>**](#)

34565

CHANGE HISTORY

34566

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34567

Issue 634568
34569

The `pthread_cond_timedwait()` and `pthread_cond_wait()` functions are marked as part of the Threads option.

34570
34571

The Open Group Corrigendum U021/9 is applied, correcting the prototype for the `pthread_cond_wait()` function.

34572
34573

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for the Clock Selection option.

- 34574 The ERRORS section has an additional case for [EPERM] in response to IEEE PASC
34575 Interpretation 1003.1c #28.
- 34576 The **restrict** keyword is added to the *pthread_cond_timedwait()* and *pthread_cond_wait()*
34577 prototypes for alignment with the ISO/IEC 9899:1999 standard.
- 34578 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/89 is applied, updating the
34579 DESCRIPTION for consistency with the *pthread_cond_destroy()* function that states it is safe to
34580 destroy an initialized condition variable upon which no threads are currently blocked.
- 34581 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/90 is applied, updating words in the
34582 DESCRIPTION from “the cancelability enable state” to “the cancelability type”.
- 34583 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/91 is applied, updating the ERRORS
34584 section to remove the error case related to *abstime* from the *pthread_cond_wait()* function, and to
34585 make the error case related to *abstime* mandatory for *pthread_cond_timedwait()* for consistency
34586 with other functions.
- 34587 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/92 is applied, adding a new paragraph to
34588 the RATIONALE section stating that an application should check the predicate on any return
34589 from this function.
- 34590 **Issue 7**
- 34591 SD5-XSH-ERN-44 is applied, changing the definition of the “shall fail” case of the [EINVAL]
34592 error.
- 34593 Changes are made from The Open Group Technical Standard, 2006, Extended API Set Part 3.
- 34594 The *pthread_cond_timedwait()* and *pthread_cond_wait()* functions are moved from the Threads
34595 option to the Base.

NAME

`pthread_condattr_destroy`, `pthread_condattr_init` — destroy and initialize the condition variable attributes object

SYNOPSIS

```
#include <pthread.h>

int pthread_condattr_destroy(pthread_condattr_t *attr);
int pthread_condattr_init(pthread_condattr_t *attr);
```

DESCRIPTION

The `pthread_condattr_destroy()` function shall destroy a condition variable attributes object; the object becomes, in effect, uninitialized. An implementation may cause `pthread_condattr_destroy()` to set the object referenced by `attr` to an invalid value. A destroyed `attr` attributes object can be reinitialized using `pthread_condattr_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

The `pthread_condattr_init()` function shall initialize a condition variable attributes object `attr` with the default value for all of the attributes defined by the implementation.

Results are undefined if `pthread_condattr_init()` is called specifying an already initialized `attr` attributes object.

After a condition variable attributes object has been used to initialize one or more condition variables, any

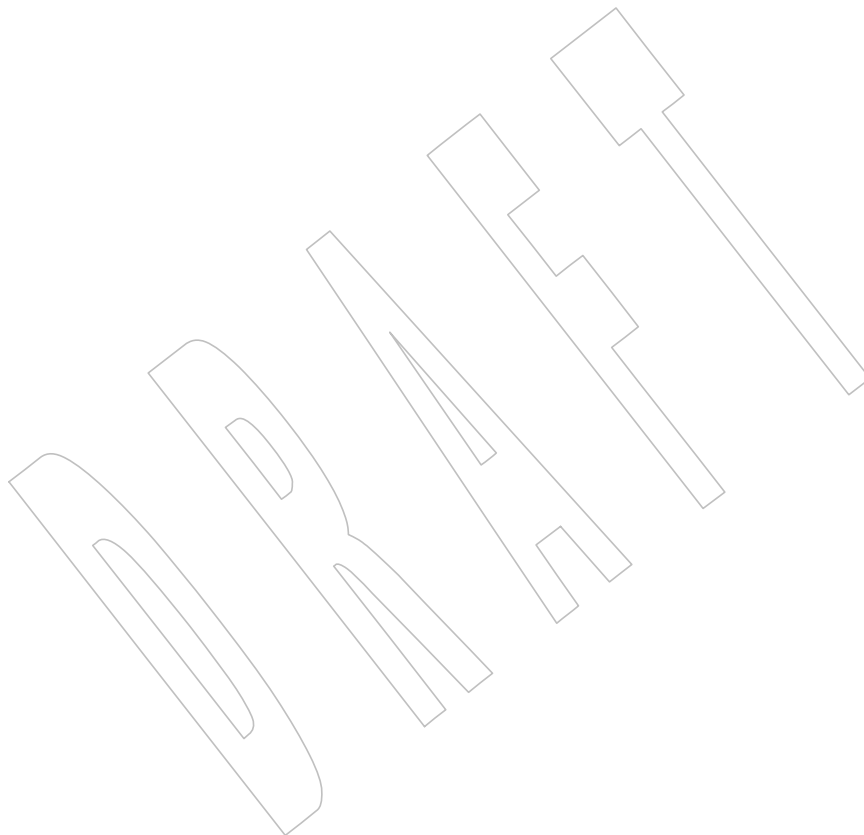
DRAFT

34637 **FUTURE DIRECTIONS**

34638 None.

34639 **SEE ALSO**34640 *pthread_attr_destroy()*, *pthread_cond_destroy()*, *pthread_condattr_getpshared()*, *pthread_create()*,
34641 *pthread_mutex_destroy()*, the Base Definitions volume of IEEE Std 1003.1-200x, <pthread.h>34642 **CHANGE HISTORY**

34643 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34644 **Issue 6**34645 The *pthread_condattr_destroy()* and *pthread_condattr_init()* functions are marked as part of the
34646 Threads option.34647 **Issue 7**34648 The *pthread_condattr_destroy()* and *pthread_condattr_init()* functions are moved from the Threads
34649 option to the Base.

34650 **NAME**

34651 pthread_condattr_getclock, pthread_condattr_setclock — get and set the clock selection
 34652 condition variable attribute

34653 **SYNOPSIS**

```
34654 #include <pthread.h>

34655 int pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
34656                             clockid_t *restrict clock_id);
34657 int pthread_condattr_setclock(pthread_condattr_t *attr,
34658                              clockid_t clock_id);
```

34659 **DESCRIPTION**

34660 The *pthread_condattr_getclock()* function shall obtain the value of the *clock* attribute from the
 34661 attributes object referenced by *attr*. The *pthread_condattr_setclock()* function shall set the *clock*
 34662 attribute in an initialized attributes object referenced by *attr*. If *pthread_condattr_setclock()* is
 34663 called with a *clock_id* argument that refers to a CPU-time clock, the call shall fail.

34664 The *clock* attribute is the clock ID of the clock that shall be used to measure the timeout service of
 34665 *pthread_cond_timedwait()*. The default value of the *clock* attribute shall refer to the system clock.

34666 **RETURN VALUE**

34667 If successful, the *pthread_condattr_getclock()* function shall return zero and store the value of the
 34668 clock attribute of *attr* into the object referenced by the *clock_id* argument. Otherwise, an error
 34669 number shall be returned to indicate the error.

34670 If successful, the *pthread_condattr_setclock()* function shall return zero; otherwise, an error
 34671 number shall be returned to indicate the error.

34672 **ERRORS**

34673 These functions may fail if:

34674 [EINVAL] The value specified by *attr* is invalid.

34675 The *pthread_condattr_setclock()* function may fail if:

34676 [EINVAL] The value specified by *clock_id* does not refer to a known clock, or is a CPU-
 34677 time clock.

34678 These functions shall not return an error code of [EINTR].

34679 **EXAMPLES**

34680 None.

34681 **APPLICATION USAGE**

34682 None.

34683 **RATIONALE**

34684 None.

34685 **FUTURE DIRECTIONS**

34686 None.

34687 **SEE ALSO**

34688 *pthread_cond_destroy()*, *pthread_cond_timedwait()*, *pthread_condattr_destroy()*,
 34689 *pthread_condattr_getpshared()* (on page 1107), *pthread_condattr_init()*,
 34690 *pthread_condattr_setpshared()* (on page 1111), *pthread_create()*, *pthread_mutex_init()*, the Base
 34691 Definitions volume of IEEE Std 1003.1-200x, <pthread.h>

pthread_condattr_getclock()*System Interfaces*

34692

CHANGE HISTORY

34693

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

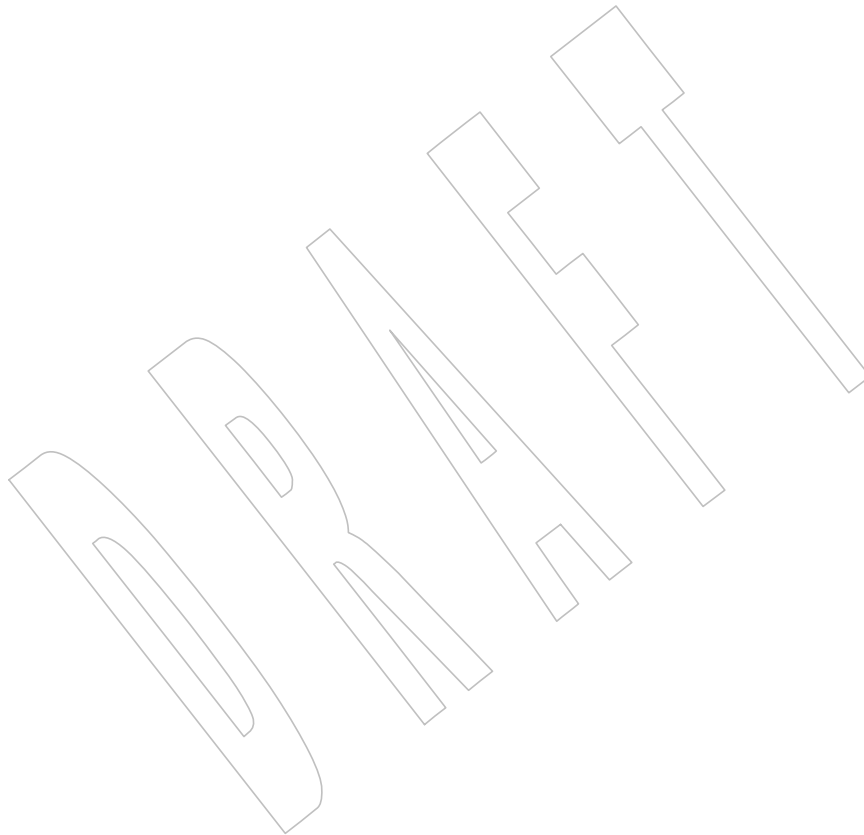
34694

Issue 7

34695

The *pthread_condattr_getclock()* and *pthread_condattr_setclock()* functions are moved from the Clock Selection option to the Base.

34696



34697 **NAME**

34698 pthread_condattr_getpshared, pthread_condattr_setpshared — get and set the process-shared
 34699 condition variable attributes

34700 **SYNOPSIS**

```
34701 TSH #include <pthread.h>
34702
34702 int pthread_condattr_getpshared(const pthread_condattr_t *restrict attr,
34703 int *restrict pshared);
34704 int pthread_condattr_setpshared(pthread_condattr_t *attr,
34705 int pshared);
```

34706 **DESCRIPTION**

34707 The *pthread_condattr_getpshared()* function shall obtain the value of the *process-shared* attribute
 34708 from the attributes object referenced by *attr*. The *pthread_condattr_setpshared()* function shall set
 34709 the *process-shared* attribute in an initialized attributes object referenced by *attr*.

34710 The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a condition
 34711 variable to be operated upon by any thread that has access to the memory where the condition
 34712 variable is allocated, even if the condition variable is allocated in memory that is shared by
 34713 multiple processes. If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the
 34714 condition variable shall only be operated upon by threads created within the same process as the
 34715 thread that initialized the condition variable; if threads of differing processes attempt to operate
 34716 on such a condition variable, the behavior is undefined. The default value of the attribute is
 34717 PTHREAD_PROCESS_PRIVATE.

34718 **RETURN VALUE**

34719 If successful, the *pthread_condattr_setpshared()* function shall return zero; otherwise, an error
 34720 number shall be returned to indicate the error.

34721 If successful, the *pthread_condattr_getpshared()* function shall return zero and store the value of
 34722 the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise,
 34723 an error number shall be returned to indicate the error.

34724 **ERRORS**

34725 The *pthread_condattr_getpshared()* and *pthread_condattr_setpshared()* functions may fail if:

34726 [EINVAL] The value specified by *attr* is invalid.

34727 The *pthread_condattr_setpshared()* function may fail if:

34728 [EINVAL] The new value specified for the attribute is outside the range of legal values
 34729 for that attribute.

34730 These functions shall not return an error code of [EINTR].

34731 **EXAMPLES**

34732 None.

34733 **APPLICATION USAGE**

34734 None.

34735 **RATIONALE**

34736 None.

pthread_condattr_getpshared()

34737

FUTURE DIRECTIONS

34738

None.

34739

SEE ALSO

34740

pthread_create(), *pthread_cond_destroy()*, *pthread_condattr_destroy()*, *pthread_mutex_destroy()*, the Base Definitions volume of IEEE Std 1003.1-200x, <pthread.h>

34741

34742

CHANGE HISTORY

34743

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34744

Issue 6

34745

The *pthread_condattr_getpshared()* and *pthread_condattr_setpshared()* functions are marked as part of the Threads and Thread Process-Shared Synchronization options.

34746

34747

The **restrict** keyword is added to the *pthread_condattr_getpshared()* prototype for alignment with the ISO/IEC 9899:1999 standard.

34748

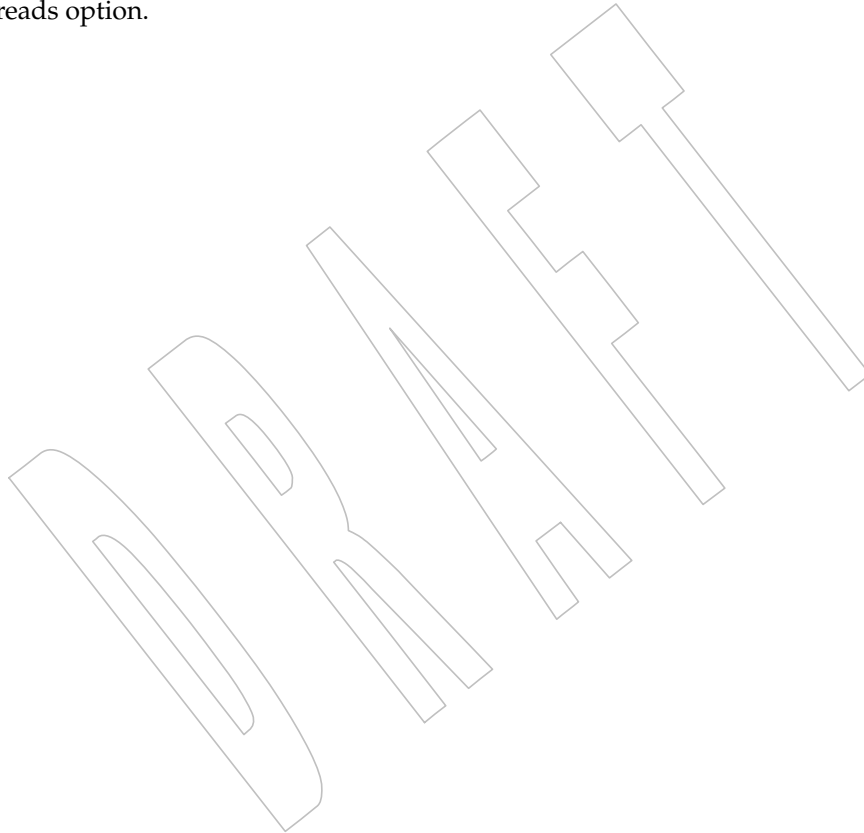
34749

Issue 7

34750

The *pthread_condattr_getpshared()* and *pthread_condattr_setpshared()* functions are moved from the Threads option.

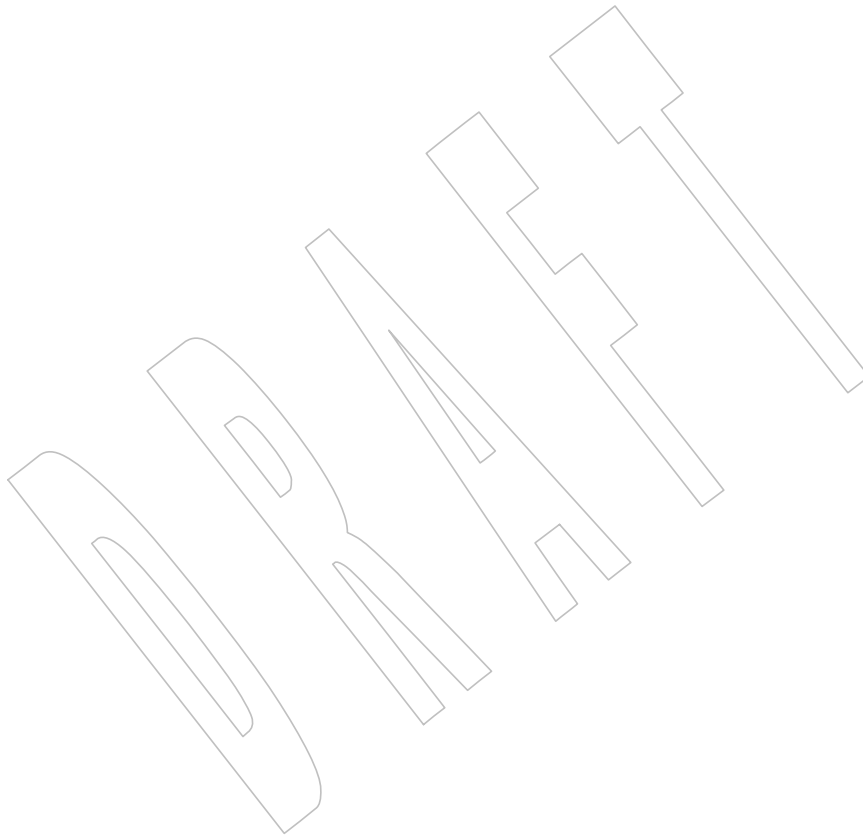
34751



34752 **NAME**
34753 pthread_condattr_init — initialize the condition variable attributes object

34754 **SYNOPSIS**
34755 #include <pthread.h>
34756 int pthread_condattr_init(pthread_condattr_t *attr);

34757 **DESCRIPTION**
34758 Refer to *pthread_condattr_destroy()*.

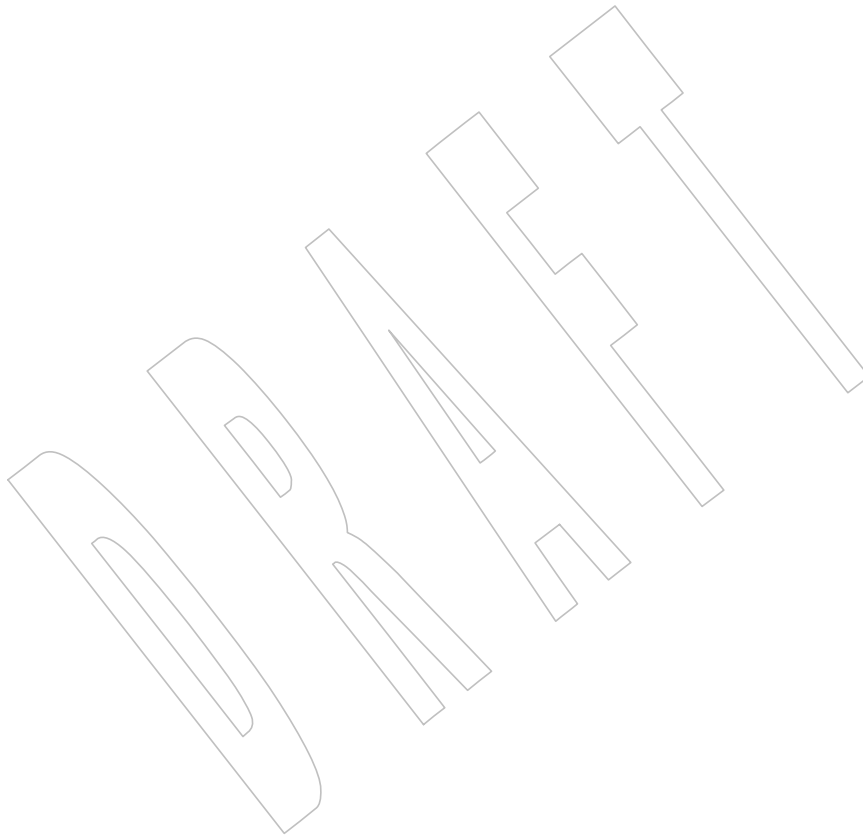


34759 **NAME**
34760 pthread_condattr_setclock — set the clock selection condition variable attribute

34761 **SYNOPSIS**
34762 #include <pthread.h>

34763 int pthread_condattr_setclock(pthread_condattr_t *attr,
34764 clockid_t clock_id);

34765 **DESCRIPTION**
34766 Refer to *pthread_condattr_getclock()*.



34767 **NAME**
34768 pthread_condattr_setpshared — set the process-shared condition variable attribute

34769 **SYNOPSIS**

34770 TSH #include <pthread.h>
34771 int pthread_condattr_setpshared(pthread_condattr_t *attr,
34772 int pshared);

34773 **DESCRIPTION**

34774 Refer to *pthread_condattr_getpshared()*.

34775 **NAME**
 34776 pthread_create — thread creation

34777 **SYNOPSIS**
 34778 #include <pthread.h>
 34779 int pthread_create(pthread_t *restrict thread,
 34780 const pthread_attr_t *restrict attr,
 34781 void *(*start_routine)(void*), void *restrict arg);

34782 **DESCRIPTION**
 34783 The *pthread_create()* function shall create a new thread, with attributes specified by *attr*, within a
 34784 process. If *attr* is NULL, the default attributes shall be used. If the attributes specified by *attr* are
 34785 modified later, the thread's attributes shall not be affected. Upon successful completion,
 34786 *pthread_create()* shall store the ID of the created thread in the location referenced by *thread*.

34787 The thread is created executing *start_routine* with *arg* as its sole argument. If the *start_routine*
 34788 returns, the effect shall be as if there was an implicit call to *pthread_exit()* using the return value
 34789 of *start_routine* as the exit status. Note that the thread in which *main()* was originally invoked
 34790 differs from this. When it returns from *main()*, the effect shall be as if there was an implicit call to
 34791 *exit()* using the return value of *main()* as the exit status.

34792 The signal state of the new thread shall be initialized as follows:

- 34793 • The signal mask shall be inherited from the creating thread.
- 34794 • The set of signals pending for the new thread shall be empty.

34795 XSI The alternate stack shall not be inherited.

34796 The floating-point environment shall be inherited from the creating thread.

34797 If *pthread_create()* fails, no new thread is created and the contents of the location referenced by
 34798 *thread* are undefined.

34799 TCT If `_POSIX_THREAD_CPUTIME` is defined, the new thread shall have a CPU-time clock
 34800 accessible, and the initial value of this clock shall be set to zero.

34801 **RETURN VALUE**
 34802 If successful, the *pthread_create()* function shall return zero; otherwise, an error number shall be
 34803 returned to indicate the error.

34804 **ERRORS**
 34805 The *pthread_create()* function shall fail if:

34806 [EAGAIN] The system lacked the necessary resources to create another thread, or the
 34807 system-imposed limit on the total number of threads in a process
 34808 {PTHREAD_THREADS_MAX} would be exceeded.

34809 [EPERM] The caller does not have appropriate permission to set the required scheduling
 34810 parameters or scheduling policy.

34811 The *pthread_create()* function may fail if:

34812 [EINVAL] The attributes specified by *attr* are invalid.

34813 The *pthread_create()* function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

There is no requirement on the implementation that the ID of the created thread be available before the newly created thread starts executing. The calling thread can obtain the ID of the created thread through the return value of the *pthread_create()* function, and the newly created thread can obtain its ID by a call to *pthread_self()*.

RATIONALE

A suggested alternative to *pthread_create()* would be to define two separate operations: create and start. Some applications would find such behavior more natural. Ada, in particular, separates the “creation” of a task from its “activation”.

Splitting the operation was rejected by the standard developers for many reasons:

- The number of calls required to start a thread would increase from one to two and thus place an additional burden on applications that do not require the additional synchronization. The second call, however, could be avoided by the additional complication of a start-up state attribute.
- An extra state would be introduced: “created but not started”. This would require the standard to specify the behavior of the thread operations when the target has not yet started executing.
- For those applications that require such behavior, it is possible to simulate the two separate steps with the facilities that are currently provided. The *start_routine()* can synchronize by waiting on a condition variable that is signaled by the start operation.

An Ada implementor can choose to create the thread at either of two points in the Ada program: when the task object is created, or when the task is activated (generally at a “begin”). If the first approach is adopted, the *start_routine()* needs to wait on a condition variable to receive the order

pthread_create()*System Interfaces*

- 34861
- 34862
- 34863
- For many implementations, the entire stack of the calling thread would need to be duplicated, since in many architectures there is no way to determine the size of the calling frame.
- 34864
- Efficiency is reduced since at least some part of the stack has to be copied, even though in most cases the thread never needs the copied context, since it merely calls the desired start routine.
- 34865
- 34866

FUTURE DIRECTIONS

None.

SEE ALSO

34869

34870

34871

fork(), *pthread_exit()*, *pthread_join()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, <**pthread.h**>

CHANGE HISTORY

34872

34873

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

34874

34875

The *pthread_create()* function is marked as part of the Threads option.

34876

34877

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- 34878
- The [EPERM] mandatory error condition is added.

34879

The thread CPU-time clock semantics are added for alignment with IEEE Std 1003.1d-1999.

34880

34881

The **restrict** keyword is added to the *pthread_create()* prototype for alignment with the ISO/IEC 9899:1999 standard.

34882

34883

The DESCRIPTION is updated to make it explicit that the floating-point environment is inherited from the creating thread.

34884

34885

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/44 is applied, adding text that the alternate stack is not inherited.

34886

34887

34888

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/93 is applied, updating the ERRORS section to remove the mandatory [EINVAL] error (“The value specified by *attr* is invalid”), and adding the optional [EINVAL] error (“The attributes specified by *attr* are invalid”).

34889

34890

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/94 is applied, adding the APPLICATION USAGE section.

Issue 7

34891

34892

The *pthread_create()* function is moved from the Threads option to the Base.

34893 **NAME**
 34894 pthread_detach — detach a thread

34895 **SYNOPSIS**
 34896 #include <pthread.h>

34897 int pthread_detach(pthread_t thread);

34898 **DESCRIPTION**
 34899 The *pthread_detach()* function shall indicate to the implementation that storage for the thread
 34900 *thread* can be reclaimed when that thread terminates. If *thread* has not terminated,
 34901 *pthread_detach()* shall not cause it to terminate. The effect of multiple *pthread_detach()* calls on the
 34902 same target thread is unspecified.

34903 **RETURN VALUE**
 34904 If the call succeeds, *pthread_detach()* shall return 0; otherwise, an error number shall be returned
 34905 to indicate the error.

34906 **ERRORS**
 34907 The *pthread_detach()* function may fail if:

34908	[EINVAL]	The implementation has detected that the value specified by <i>thread</i> does not refer to a joinable thread.
34909		
34910	[ESRCH]	No thread could be found corresponding to that specified by the given thread ID.
34911		

34912 The *pthread_detach()* function shall not return an error code of [EINTR].

34913 **EXAMPLES**
 34914 None.

34915 **APPLICATION USAGE**
 34916 None.

34917 **RATIONALE**
 34918 The *pthread_join()* or *pthread_detach()* functions should eventually be called for every thread that
 34919 is created so that storage associated with the thread may be reclaimed.

34920 It has been suggested that a “detach” function is not necessary; the *detachstate* thread creation
 34921 attribute is sufficient, since a thread need never be dynamically detached. However, need arises
 34922 in at least two cases:

- 34923 1. In a cancellation handler for a *pthread_join()* it is nearly essential to have a
 34924 *pthread_detach()* function in order to detach the thread on which *pthread_join()* was
 34925 waiting. Without it, it would be necessary to have the handler do another *pthread_join()* to
 34926 attempt to detach the thread, which would both delay the cancellation processing for an
 34927 unbounded period and introduce a new call to *pthread_join()*, which might itself need a
 34928 cancellation handler. A dynamic detach is nearly essential in this case.
- 34929 2. In order to detach the “initial thread” (as may be desirable in processes that set up server
 34930 threads).

34931 **FUTURE DIRECTIONS**
 34932 None.

pthread_detach()

34933

SEE ALSO

34934

pthread_join(), the Base Definitions volume of IEEE Std 1003.1-200x, <pthread.h>

34935

CHANGE HISTORY

34936

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34937

Issue 6

34938

The *pthread_detach()* function is marked as part of the Threads option.

34939

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/95 is applied, updating the ERRORS section so that the [EINVAL] and [ESRCH] error cases become optional.

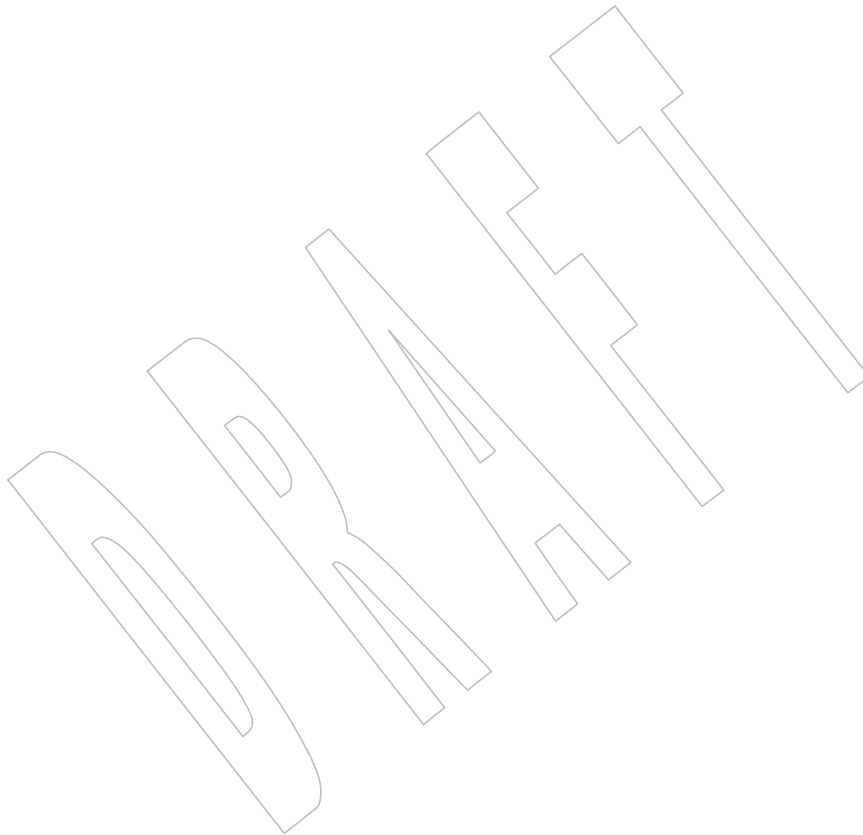
34940

34941

Issue 7

34942

The *pthread_detach()* function is moved from the Threads option to the Base.



34943 **NAME**

34944 pthread_equal — compare thread IDs

34945 **SYNOPSIS**

34946 #include <pthread.h>

34947 int pthread_equal(pthread_t t1, pthread_t t2);

34948 **DESCRIPTION**34949 This function shall compare the thread IDs *t1* and *t2*.34950 **RETURN VALUE**34951 The *pthread_equal()* function shall return a non-zero value if *t1* and *t2* are equal; otherwise, zero
34952 shall be returned.34953 If either *t1* or *t2* are not valid thread IDs, the behavior is undefined.34954 **ERRORS**

34955 No errors are defined.

34956 The *pthread_equal()* function shall not return an error code of [EINTR].34957 **EXAMPLES**

34958 None.

34959 **APPLICATION USAGE**

34960 None.

34961 **RATIONALE**34962 Implementations may choose to define a thread ID as a structure. This allows additional
34963 flexibility and robustness over using an **int**. For example, a thread ID could include a sequence
34964 number that allows detection of “dangling IDs” (copies of a thread ID that has been detached).
34965 Since the C language does not support comparison on structure types, the *pthread_equal()*
34966 function is provided to compare thread IDs.34967 **FUTURE DIRECTIONS**

34968 None.

34969 **SEE ALSO**34970 *pthread_create()*, *pthread_self()*, the Base Definitions volume of IEEE Std 1003.1-200x, <pthread.h>34971 **CHANGE HISTORY**

34972 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34973 **Issue 6**34974 The *pthread_equal()* function is marked as part of the Threads option.34975 **Issue 7**34976 The *pthread_equal()* function is moved from the Threads option to the Base.

34977 **NAME**

34978 pthread_exit — thread termination

34979 **SYNOPSIS**

34980 #include <pthread.h>

34981 void pthread_exit(void *value_ptr);

34982 **DESCRIPTION**

34983 The *pthread_exit()* function shall terminate the calling thread and make the value *value_ptr*
 34984 available to any successful join with the terminating thread. Any cancellation cleanup handlers
 34985 that have been pushed and not yet popped shall be popped in the reverse order that they were
 34986 pushed and then executed. After all cancellation cleanup handlers have been executed, if the
 34987 thread has any thread-specific data, appropriate destructor functions shall be called in an
 34988 unspecified order. Thread termination does not release any application visible process resources,
 34989 including, but not limited to, mutexes and file descriptors, nor does it perform any process-level
 34990 cleanup actions, including, but not limited to, calling any *atexit()* routines that may exist.

34991 An implicit call to *pthread_exit()* is made when a thread other than the thread in which *main()*
 34992 was first invoked returns from the start routine that was used to create it. The function's return
 34993 value shall serve as the thread's exit status.

34994 The behavior of *pthread_exit()* is undefined if called from a cancellation cleanup handler or
 34995 destructor function that was invoked as a result of either an implicit or explicit call to
 34996 *pthread_exit()*.

34997 After a thread has terminated, the result of access to local (auto) variables of the thread is
 34998 undefined. Thus, references to local variables of the exiting thread should not be used for the
 34999 *pthread_exit()* *value_ptr* parameter value.

35000 The process shall exit with an exit status of 0 after the last thread has been terminated. The
 35001 behavior shall be as if the implementation called *exit()* with a zero argument at thread
 35002 termination time.

35003 **RETURN VALUE**35004 The *pthread_exit()* function cannot return to its caller.35005 **ERRORS**

35006 No errors are defined.

35007 **EXAMPLES**

35008 None.

35009 **APPLICATION USAGE**

35010 None.

35011 **RATIONALE**

35012 The normal mechanism by which a thread terminates is to return from the routine that was
 35013 specified in the *pthread_create()* call that started it. The *pthread_exit()* function provides the
 35014 capability for a thread to terminate without requiring a return from the start routine of that
 35015 thread, thereby providing a function analogous to *exit()*.

35016 Regardless of the method of thread termination, any cancellation cleanup handlers that have
 35017 been pushed and not yet popped are executed, and the destructors for any existing thread-
 35018 specific data are executed. This volume of IEEE Std 1003.1-200x requires that cancellation
 35019 cleanup handlers be popped and called in order. After all cancellation cleanup handlers have
 35020 been executed, thread-specific data destructors are called, in an unspecified order, for each item
 35021 of thread-specific data that exists in the thread. This ordering is necessary because cancellation

35022 cleanup handlers may rely on thread-specific data.

35023 As the meaning of the status is determined by the application (except when the thread has been
35024 canceled, in which case it is PTHREAD_CANCELED), the implementation has no idea what an
35025 illegal status value is, which is why no address error checking is done.

35026 **FUTURE DIRECTIONS**

35027 None.

35028 **SEE ALSO**

35029 *exit()*, *pthread_create()*, *pthread_join()*, the Base Definitions volume of IEEE Std 1003.1-200x,
35030 **<pthread.h>**

35031 **CHANGE HISTORY**

35032 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

35033 **Issue 6**

35034 The *pthread_exit()* function is marked as part of the Threads option.

35035 **Issue 7**

35036 The *pthread_exit()* function is moved from the Threads option to the Base.

DRAFT

35037 **NAME**

35038 pthread_getconcurrency, pthread_setconcurrency — get and set the level of concurrency

35039 **SYNOPSIS**

```
35040 OB XSI #include <pthread.h>
35041 int pthread_getconcurrency(void);
35042 int pthread_setconcurrency(int new_level);
```

35043 **DESCRIPTION**

35044 Unbound threads in a process may or may not be required to be simultaneously active. By
 35045 default, the threads implementation ensures that a sufficient number of threads are active so that
 35046 the process can continue to make progress. While this conserves system resources, it may not
 35047 produce the most effective level of concurrency.

35048 The *pthread_setconcurrency()* function allows an application to inform the threads
 35049 implementation of its desired concurrency level, *new_level*. The actual level of concurrency
 35050 provided by the implementation as a result of this function call is unspecified.

35051 If *new_level* is zero, it causes the implementation to maintain the concurrency level at its
 35052 discretion as if *pthread_setconcurrency()* had never been called.

35053 The *pthread_getconcurrency()* function shall return the value set by a previous call to the
 35054 *pthread_setconcurrency()* function. If the *pthread_setconcurrency()* function was not previously
 35055 called, this function shall return zero to indicate that the implementation is maintaining the
 35056 concurrency level.

35057 A call to *pthread_setconcurrency()* shall inform the implementation of its desired concurrency
 35058 level. The implementation shall use this as a hint, not a requirement.

35059 If an implementation does not support multiplexing of user threads on top of several kernel-
 35060 scheduled entities, the *pthread_setconcurrency()* and *pthread_getconcurrency()* functions are
 35061 provided for source code compatibility but they shall have no effect when called. To maintain
 35062 the function semantics, the *new_level* parameter is saved when *pthread_setconcurrency()* is called
 35063 so that a subsequent call to *pthread_getconcurrency()* shall return the same value.

35064 **RETURN VALUE**

35065 If successful, the *pthread_setconcurrency()* function shall return zero; otherwise, an error number
 35066 shall be returned to indicate the error.

35067 The *pthread_getconcurrency()* function shall always return the concurrency level set by a previous
 35068 call to *pthread_setconcurrency()*. If the *pthread_setconcurrency()* function has never been called,
 35069 *pthread_getconcurrency()* shall return zero.

35070 **ERRORS**

35071 The *pthread_setconcurrency()* function shall fail if:

- | | | |
|-------|----------|---|
| 35072 | [EINVAL] | The value specified by <i>new_level</i> is negative. |
| 35073 | [EAGAIN] | The value specified by <i>new_level</i> would cause a system resource to be exceeded. |

35074
 35075 These functions shall not return an error code of [EINTR].

35076
35077
35078
35079
35080
35081
35082
35083
35084
35085
35086
35087
35088
35089
35090
35091

EXAMPLES

None.

APPLICATION USAGE

Application developers should note that an implementation can always ignore any calls to *pthread_setconcurrency()* and return a constant for *pthread_getconcurrency()*. For this reason, it is not recommended that portable applications use this function.

RATIONALE

None.

FUTURE DIRECTIONS

These functions may be removed in a future version.

SEE ALSO

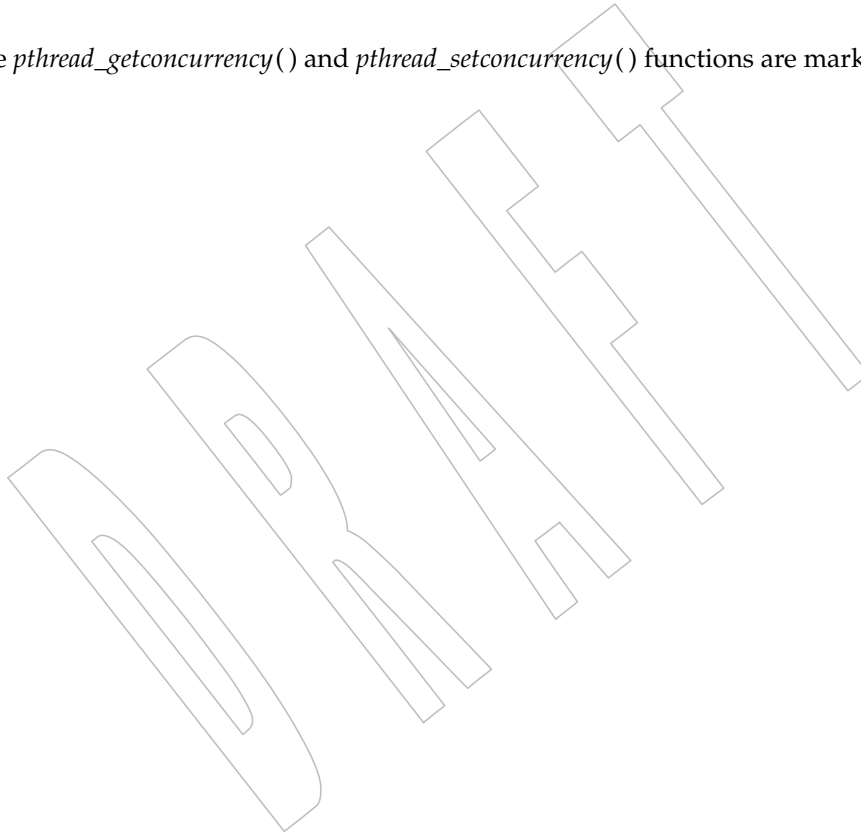
The Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 7

The *pthread_getconcurrency()* and *pthread_setconcurrency()* functions are marked obsolescent.



35092 **NAME**

35093 pthread_getcpuclockid — access a thread CPU-time clock (**ADVANCED REALTIME**
 35094 **THREADS**)

35095 **SYNOPSIS**

```
35096 TCT #include <pthread.h>
35097 #include <time.h>
35098 int pthread_getcpuclockid(pthread_t thread_id, clockid_t *clock_id);
```

35099 **DESCRIPTION**

35100 The *pthread_getcpuclockid()* function shall return in *clock_id* the clock ID of the CPU-time clock of
 35101 the thread specified by *thread_id*, if the thread specified by *thread_id* exists.

35102 **RETURN VALUE**

35103 Upon successful completion, *pthread_getcpuclockid()* shall return zero; otherwise, an error
 35104 number shall be returned to indicate the error.

35105 **ERRORS**

35106 The *pthread_getcpuclockid()* function may fail if:

35107 [ESRCH] The value specified by *thread_id* does not refer to an existing thread.

35108 **EXAMPLES**

35109 None.

35110 **APPLICATION USAGE**

35111 The *pthread_getcpuclockid()* function is part of the Thread CPU-Time Clocks option and need not
 35112 be provided on all implementations.

35113 **RATIONALE**

35114 None.

35115 **FUTURE DIRECTIONS**

35116 None.

35117 **SEE ALSO**

35118 *clock_getcpuclockid()*, *clock_getres()*, *timer_create()*, the Base Definitions volume of
 35119 IEEE Std 1003.1-200x, **<pthread.h>**, **<time.h>**

35120 **CHANGE HISTORY**

35121 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

35122 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

35123 **Issue 7**

35124 The *pthread_getcpuclockid()* function is moved from the Threads option.

35125 **NAME**

35126 pthread_getschedparam, pthread_setschedparam — dynamic thread scheduling parameters
 35127 access (**REALTIME THREADS**)

35128 **SYNOPSIS**

```
35129 TPS #include <pthread.h>
35130
35131 int pthread_getschedparam(pthread_t thread, int *restrict policy,
35132                          struct sched_param *restrict param);
35133 int pthread_setschedparam(pthread_t thread, int policy,
35134                          const struct sched_param *param);
```

35134 **DESCRIPTION**

35135 The *pthread_getschedparam()* and *pthread_setschedparam()* functions shall, respectively, get and set
 35136 the scheduling policy and parameters of individual threads within a multi-threaded process to
 35137 be retrieved and set. For SCHED_FIFO and SCHED_RR, the only required member of the
 35138 **sched_param** structure is the priority *sched_priority*. For SCHED_OTHER, the affected
 35139 scheduling parameters are implementation-defined.

35140 The *pthread_getschedparam()* function shall retrieve the scheduling policy and scheduling
 35141 parameters for the thread whose thread ID is given by *thread* and shall store those values in
 35142 *policy* and *param*, respectively. The priority value returned from *pthread_getschedparam()* shall be
 35143 the value specified by the most recent *pthread_setschedparam()*, *pthread_setschedprio()*, or
 35144 *pthread_create()* call affecting the target thread. It shall not reflect any temporary adjustments to
 35145 its priority as a result of any priority inheritance or ceiling functions. The *pthread_setschedparam()*
 35146 function shall set the scheduling policy and associated scheduling parameters for the thread
 35147 whose thread ID is given by *thread* to the policy and associated parameters provided in *policy*
 35148 and *param*, respectively.

35149 The *policy* parameter may have the value SCHED_OTHER, SCHED_FIFO, or SCHED_RR. The
 35150 scheduling parameters for the SCHED_OTHER policy are implementation-defined. The
 35151 SCHED_FIFO and SCHED_RR policies shall have a single scheduling parameter, *priority*.

35152 TSP If **_POSIX_THREAD_SPORADIC_SERVER** is defined, then the *policy* argument may have the
 35153 value SCHED_SPORADIC, with the exception for the *pthread_setschedparam()* function that if the
 35154 scheduling policy was not SCHED_SPORADIC at the time of the call, it is implementation-
 35155 defined whether the function is supported; in other words, the implementation need not allow
 35156 the application to dynamically change the scheduling policy to SCHED_SPORADIC. The
 35157 sporadic server scheduling policy has the associated parameters *sched_ss_low_priority*,
 35158 *sched_ss_repl_period*, *sched_ss_init_budget*, *sched_priority*, and *sched_ss_max_repl*. The specified
 35159 *sched_ss_repl_period* shall be greater than or equal to the specified *sched_ss_init_budget* for the
 35160 function to succeed; if it is not, then the function shall fail. The value of *sched_ss_max_repl* shall
 35161 be within the inclusive range [1,{SS_REPL_MAX}] for the function to succeed; if not, the function
 35162 shall fail.

35163 If the *pthread_setschedparam()* function fails, the scheduling parameters shall not be changed for
 35164 the target thread.

35165 **RETURN VALUE**

35166 If successful, the *pthread_getschedparam()* and *pthread_setschedparam()* functions shall return zero;
 35167 otherwise, an error number shall be returned to indicate the error.

ERRORS35168
35169
35170
35171
35172
35173
35174
35175
35176
35177
35178
35179
35180
35181
35182
35183
35184
35185
35186
35187
35188
35189
35190
35191
35192
35193
35194
35195
35196
35197
35198
35199
35200
35201
35202
35203
35204
35205
35206
35207
35208
35209

The *pthread_getschedparam()* function may fail if:

[ESRCH] The value specified by *thread* does not refer to an existing thread.

The *pthread_setschedparam()* function may fail if:

[EINVAL] The value specified by *policy* or one of the scheduling parameters associated with the scheduling policy *policy* is invalid.

[ENOTSUP] An attempt was made to set the policy or scheduling parameters to an unsupported value.

TSP [ENOTSUP] An attempt was made to dynamically change the scheduling policy to SCHED_SPORADIC, and the implementation does not support this change.

[EPERM] The caller does not have the appropriate permission to set either the scheduling parameters or the scheduling policy of the specified thread.

[EPERM] The implementation does not allow the application to modify one of the parameters to the value specified.

[ESRCH] The value specified by *thread* does not refer to an existing thread.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_setschedprio(), *sched_getparam()*, *sched_getscheduler()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<pthread.h>`, `<sched.h>`

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_getschedparam()* and *pthread_setschedparam()* functions are marked as part of the Threads and Thread Execution Scheduling options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The Open Group Corrigendum U026/2 is applied, correcting the prototype for the *pthread_setschedparam()* function so that its second argument is of type **int**.

The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the *pthread_getschedparam()* prototype for alignment with the ISO/IEC 9899:1999 standard.

The Open Group Corrigendum U047/1 is applied.

IEEE PASC Interpretation 1003.1 #96 is applied, noting that priority values can also be set by a call to the *pthread_setschedprio()* function.

35210
35211
35212**Issue 7**

The *pthread_getschedparam()* and *pthread_setschedparam()* functions are moved from the Threads option.



35213 **NAME**

35214 pthread_getspecific, pthread_setspecific — thread-specific data management

35215 **SYNOPSIS**

35216 #include <pthread.h>

35217 void *pthread_getspecific(pthread_key_t key);

35218 int pthread_setspecific(pthread_key_t key, const void *value);

35219 **DESCRIPTION**35220 The *pthread_getspecific()* function shall return the value currently bound to the specified *key* on
35221 behalf of the calling thread.35222 The *pthread_setspecific()* function shall associate a thread-specific *value* with a *key* obtained via a
35223 previous call to *pthread_key_create()*. Different threads may bind different values to the same
35224 key. These values are typically pointers to blocks of dynamically allocated memory that have
35225 been reserved for use by the calling thread.35226 The effect of calling *pthread_getspecific()* or *pthread_setspecific()* with a *key* value not obtained
35227 from *pthread_key_create()* or after *key* has been deleted with *pthread_key_delete()* is undefined.35228 Both *pthread_getspecific()* and *pthread_setspecific()* may be called from a thread-specific data
35229 destructor function. A call to *pthread_getspecific()* for the thread-specific data key being
35230 destroyed shall return the value NULL, unless the value is changed (after the destructor starts)
35231 by a call to *pthread_setspecific()*. Calling *pthread_setspecific()* from a thread-specific data
35232 destructor routine may result either in lost storage (after at least
35233 PTHREAD_DESTRUCTOR_ITERATIONS attempts at destruction) or in an infinite loop.

35234 Both functions may be implemented as macros.

35235 **RETURN VALUE**35236 The *pthread_getspecific()* function shall return the thread-specific data value associated with the
35237 given *key*. If no thread-specific data value is associated with *key*, then the value NULL shall be
35238 returned.35239 If successful, the *pthread_setspecific()* function shall return zero; otherwise, an error number shall
35240 be returned to indicate the error.35241 **ERRORS**35242 No errors are returned from *pthread_getspecific()*.35243 The *pthread_setspecific()* function shall fail if:

35244 [ENOMEM] Insufficient memory exists to associate the non-NULL value with the key.

35245 The *pthread_setspecific()* function may fail if:

35246 [EINVAL] The key value is invalid.

35247 The *pthread_setspecific()* function shall not return an error code of [EINTR].

35248
35249
35250
35251
35252
35253
35254
35255
35256
35257
35258
35259
35260
35261
35262
35263
35264
35265
35266
35267
35268
35269
35270
35271
35272
35273

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Performance and ease-of-use of *pthread_getspecific()* are critical for functions that rely on maintaining state in thread-specific data. Since no errors are required to be detected by it, and since the only error that could be detected is the use of an invalid key, the function to *pthread_getspecific()* has been designed to favor speed and simplicity over error reporting.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_key_create(), the Base Definitions volume of IEEE Std 1003.1-200x, <pthread.h>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_getspecific()* and *pthread_setspecific()* functions are marked as part of the Threads option.

IEEE PASC Interpretation 1003.1c #3 (Part 6) is applied, updating the DESCRIPTION.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/96 is applied, updating the ERRORS section so that the [ENOMEM] error case is changed from “to associate the value with the key” to “to associate the non-NULL value with the key”.

Issue 7

Austin Group Interpretation 1003.1-2001 #063 is applied, updating the ERRORS section.

The *pthread_getspecific()* and *pthread_setspecific()* functions are moved from the Threads option to the Base.

35274 **NAME**
 35275 pthread_join — wait for thread termination

35276 **SYNOPSIS**
 35277 #include <pthread.h>
 35278 int pthread_join(pthread_t thread, void **value_ptr);

35279 **DESCRIPTION**
 35280 The *pthread_join()* function shall suspend execution of the calling thread until the target *thread*
 35281 terminates, unless the target *thread* has already terminated. On return from a successful
 35282 *pthread_join()* call with a non-NULL *value_ptr* argument, the value passed to *pthread_exit()* by
 35283 the terminating thread shall be made available in the location referenced by *value_ptr*. When a
 35284 *pthread_join()* returns successfully, the target thread has been terminated. The results of multiple
 35285 simultaneous calls to *pthread_join()* specifying the same target thread are undefined. If the
 35286 thread calling *pthread_join()* is canceled, then the target thread shall not be detached.

35287 It is unspecified whether a thread that has exited but remains unjoined counts against
 35288 {PTHREAD_THREADS_MAX}.

35289 **RETURN VALUE**
 35290 If successful, the *pthread_join()* function shall return zero; otherwise, an error number shall be
 35291 returned to indicate the error.

35292 **ERRORS**
 35293 The *pthread_join()* function shall fail if:
 35294 [ESRCH] No thread could be found corresponding to that specified by the given thread
 35295 ID.

35296 The *pthread_join()* function may fail if:
 35297 [EDEADLK] A deadlock was detected or the value of *thread* specifies the calling thread.
 35298 [EINVAL] The value specified by *thread* does not refer to a joinable thread.

35299 The *pthread_join()* function shall not return an error code of [EINTR].

35300 **EXAMPLES**
 35301 An example of thread creation and deletion follows:

```
35302 typedef struct {
35303     int *ar;
35304     long n;
35305 } subarray;
35306
35307 void *
35308 incer(void *arg)
35309 {
35310     long i;
35311     for (i = 0; i < ((subarray *)arg)->n; i++)
35312         ((subarray *)arg)->ar[i]++;
35313 }
35314
35315 int main(void)
35316 {
35317     int ar[1000000];
35318     pthread_t th1, th2;
35319     subarray sb1, sb2;
```

```

35318         sb1.ar = &ar[0];
35319         sb1.n  = 500000;
35320         (void) pthread_create(&th1, NULL, incer, &sb1);
35321
35322         sb2.ar = &ar[500000];
35323         sb2.n  = 500000;
35324         (void) pthread_create(&th2, NULL, incer, &sb2);
35325
35326         (void) pthread_join(th1, NULL);
35327         (void) pthread_join(th2, NULL);
35328         return 0;
35329     }

```

APPLICATION USAGE

None.

RATIONALE

The *pthread_join()* function is a convenience that has proven useful in multi-threaded applications. It is true that a programmer could simulate this function if it were not provided by passing extra state as part of the argument to the *start_routine()*. The terminating thread would set a flag to indicate termination and broadcast a condition that is part of that state; a joining thread would wait on that condition variable. While such a technique would allow a thread to wait on more complex conditions (for example, waiting for multiple threads to terminate), waiting on individual thread termination is considered widely useful. Also, including the *pthread_join()* function in no way precludes a programmer from coding such complex waits. Thus, while not a primitive, including *pthread_join()* in this volume of IEEE Std 1003.1-200x was considered valuable.

The *pthread_join()* function provides a simple mechanism allowing an application to wait for a thread to terminate. After the thread terminates, the application may then choose to clean up resources that were used by the thread. For instance, after *pthread_join()* returns, any application-provided stack storage could be reclaimed.

The *pthread_join()* or *pthread_detach()* function should eventually be called for every thread that is created with the *detachstate* attribute set to `PTHREAD_CREATE_JOINABLE` so that storage associated with the thread may be reclaimed.

The interaction between *pthread_join()* and cancellation is well-defined for the following reasons:

- The *pthread_join()* function, like all other non-async-cancel-safe functions, can only be called with deferred cancelability type.
- Cancellation cannot occur in the disabled cancelability state.

Thus, only the default cancelability state need be considered. As specified, either the *pthread_join()* call is canceled, or it succeeds, but not both. The difference is obvious to the application, since either a cancellation handler is run or *pthread_join()* returns. There are no race conditions since *pthread_join()* was called in the deferred cancelability state.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_create(), *wait()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, `<pthread.h>`

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

pthread_join()

35363

Issue 6

35364

The *pthread_join()* function is marked as part of the Threads option.

35365

35366

35367

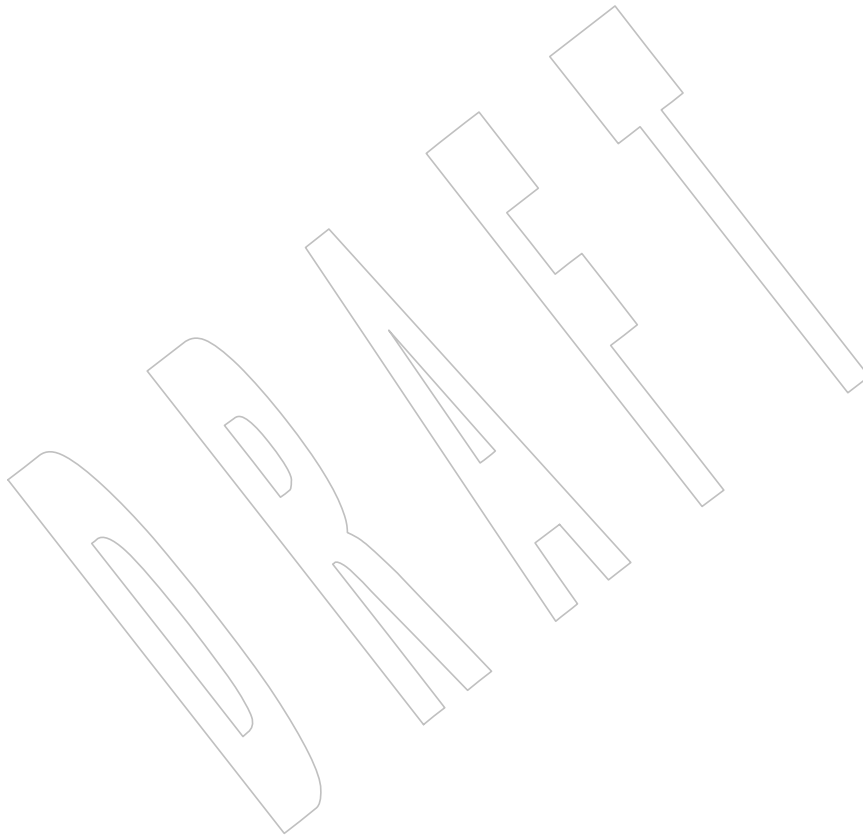
IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/97 is applied, updating the ERRORS section so that the [EINVAL] error is made optional and the words “the implementation has detected” are removed from it.

35368

Issue 7

35369

The *pthread_join()* function is moved from the Threads option to the Base.



35370 **NAME**
 35371 pthread_key_create — thread-specific data key creation

35372 **SYNOPSIS**
 35373 #include <pthread.h>

35374 int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));

35375 **DESCRIPTION**

35376 The *pthread_key_create()* function shall create a thread-specific data key visible to all threads in
 35377 the process. Key values provided by *pthread_key_create()* are opaque objects used to locate
 35378 thread-specific data. Although the same key value may be used by different threads, the values
 35379 bound to the key by *pthread_setspecific()* are maintained on a per-thread basis and persist for the
 35380 life of the calling thread.

35381 Upon key creation, the value NULL shall be associated with the new key in all active threads.
 35382 Upon thread creation, the value NULL shall be associated with all defined keys in the new
 35383 thread.

35384 An optional destructor function may be associated with each key value. At thread exit, if a key
 35385 value has a non-NULL destructor pointer, and the thread has a non-NULL value associated with
 35386 that key, the value of the key is set to NULL, and then the function pointed to is called with the
 35387 previously associated value as its sole argument. The order of destructor calls is unspecified if
 35388 more than one destructor exists for a thread when it exits.

35389 If, after all the destructors have been called for all non-NULL values with associated destructors,
 35390 there are still some non-NULL values with associated destructors, then the process is repeated.
 35391 If, after at least {PTHREAD_DESTRUCTOR_ITERATIONS} iterations of destructor calls for
 35392 outstanding non-NULL values, there are still some non-NULL values with associated
 35393 destructors, implementations may stop calling destructors, or they may continue calling
 35394 destructors until no non-NULL values with associated destructors exist, even though this might
 35395 result in an infinite loop.

35396 **RETURN VALUE**

35397 If successful, the *pthread_key_create()* function shall store the newly created key value at *key and
 35398 shall return zero. Otherwise, an error number shall be returned to indicate the error.

35399 **ERRORS**

35400 The *pthread_key_create()* function shall fail if:

35401 [EAGAIN] The system lacked the necessary resources to create another thread-specific
 35402 data key, or the system-imposed limit on the total number of keys per process
 35403 {PTHREAD_KEYS_MAX} has been exceeded.

35404 [ENOMEM] Insufficient memory exists to create the key.

35405 The *pthread_key_create()* function shall not return an error code of [EINTR].

35406 **EXAMPLES**

35407 The following example demonstrates a function that initializes a thread-specific data key when it
 35408 is first called, and associates a thread-specific object with each calling thread, initializing this
 35409 object when necessary.

```
35410 static pthread_key_t key;
35411 static pthread_once_t key_once = PTHREAD_ONCE_INIT;

35412 static void
35413 make_key()
35414 {
```

```

35415         (void) pthread_key_create(&key, NULL);
35416     }
35417 func()
35418 {
35419     void *ptr;
35420
35421     (void) pthread_once(&key_once, make_key);
35422     if ((ptr = pthread_getspecific(key)) == NULL) {
35423         ptr = malloc(OBJECT_SIZE);
35424         ...
35425         (void) pthread_setspecific(key, ptr);
35426     }
35427     ...
35428 }

```

Note that the key has to be initialized before *pthread_getspecific()* or *pthread_setspecific()* can be used. The *pthread_key_create()* call could either be explicitly made in a module initialization routine, or it can be done implicitly by the first call to a module as in this example. Any attempt to use the key before it is initialized is a programming error, making the code below incorrect.

```

35432 static pthread_key_t key;
35433 func()
35434 {
35435     void *ptr;
35436
35437     /* KEY NOT INITIALIZED!!! THIS WON'T WORK!!! */
35438     if ((ptr = pthread_getspecific(key)) == NULL &&
35439         pthread_setspecific(key, NULL) != 0) {
35440         pthread_key_create(&key, NULL);
35441         ...
35442     }
35443 }

```

APPLICATION USAGE

None.

RATIONALE

Destructor Functions

Normally, the value bound to a key on behalf of a particular thread is a pointer to storage allocated dynamically on behalf of the calling thread. The destructor functions specified with *pthread_key_create()* are intended to be used to free this storage when the thread exits. Thread cancellation cleanup handlers cannot be used for this purpose because thread-specific data may persist outside the lexical scope in which the cancellation cleanup handlers operate.

If the value associated with a key needs to be updated during the lifetime of the thread, it may be necessary to release the storage associated with the old value before the new value is bound. Although the *pthread_setspecific()* function could do this automatically, this feature is not needed often enough to justify the added complexity. Instead, the programmer is responsible for freeing the stale storage:

```

35457 pthread_getspecific(key, &old);
35458 new = allocate();
35459 destructor(old);
35460 pthread_setspecific(key, new);

```

Note: The above example could leak storage if run with asynchronous cancellation enabled. No such problems occur in the default cancellation state if no cancellation points occur between the get and set.

There is no notion of a destructor-safe function. If an application does not call *pthread_exit()* from a signal handler, or if it blocks any signal whose handler may call *pthread_exit()* while calling async-unsafe functions, all functions may be safely called from destructors.

Non-Idempotent Data Key Creation

There were requests to make *pthread_key_create()* idempotent with respect to a given *key* address parameter. This would allow applications to call *pthread_key_create()* multiple times for a given *key* address and be guaranteed that only one key would be created. Doing so would require the key value to be previously initialized (possibly at compile time) to a known null value and would require that implicit mutual-exclusion be performed based on the address and contents of the *key* parameter in order to guarantee that exactly one key would be created.

Unfortunately, the implicit mutual-exclusion would not be limited to only *pthread_key_create()*. On many implementations, implicit mutual-exclusion would also have to be performed by *pthread_getspecific()* and *pthread_setspecific()* in order to guard against using incompletely stored or not-yet-visible key values. This could significantly increase the cost of important operations, particularly *pthread_getspecific()*.

Thus, this proposal was rejected. The *pthread_key_create()* function performs no implicit synchronization. It is the responsibility of the programmer to ensure that it is called exactly once per key before use of the key. Several straightforward mechanisms can already be used to accomplish this, including calling explicit module initialization functions, using mutexes, and using *pthread_once()*. This places no significant burden on the programmer, introduces no possibly confusing *ad hoc* implicit synchronization mechanism, and potentially allows commonly used thread-specific data operations to be more efficient.

FUTURE DIRECTIONS

None.

SEE ALSO

[*pthread_getspecific\(\)*](#), [*pthread_key_delete\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, [`<pthread.h>`](#)

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_key_create()* function is marked as part of the Threads option.

IEEE PASC Interpretation 1003.1c #8 is applied, updating the DESCRIPTION.

Issue 7

The *pthread_key_create()* function is moved from the Threads option to the Base.

35497 **NAME**

35498 pthread_key_delete — thread-specific data key deletion

35499 **SYNOPSIS**

35500 #include <pthread.h>

35501 int pthread_key_delete(pthread_key_t key);

35502 **DESCRIPTION**

35503 The *pthread_key_delete()* function shall delete a thread-specific data key previously returned by
 35504 *pthread_key_create()*. The thread-specific data values associated with *key* need not be NULL at
 35505 the time *pthread_key_delete()* is called. It is the responsibility of the application to free any
 35506 application storage or perform any cleanup actions for data structures related to the deleted key
 35507 or associated thread-specific data in any threads; this cleanup can be done either before or after
 35508 *pthread_key_delete()* is called. Any attempt to use *key* following the call to *pthread_key_delete()*
 35509 results in undefined behavior.

35510 The *pthread_key_delete()* function shall be callable from within destructor functions. No
 35511 destructor functions shall be invoked by *pthread_key_delete()*. Any destructor function that may
 35512 have been associated with *key* shall no longer be called upon thread exit.

35513 **RETURN VALUE**

35514 If successful, the *pthread_key_delete()* function shall return zero; otherwise, an error number shall
 35515 be returned to indicate the error.

35516 **ERRORS**35517 The *pthread_key_delete()* function may fail if:35518 [EINVAL] The *key* value is invalid.35519 The *pthread_key_delete()* function shall not return an error code of [EINTR].35520 **EXAMPLES**

35521 None.

35522 **APPLICATION USAGE**

35523 None.

35524 **RATIONALE**

35525 A thread-specific data key deletion function has been included in order to allow the resources
 35526 associated with an unused thread-specific data key to be freed. Unused thread-specific data keys
 35527 can arise, among other scenarios, when a dynamically loaded module that allocated a key is
 35528 unloaded.

35529 Conforming applications are responsible for performing any cleanup actions needed for data
 35530 structures associated with the key to be deleted, including data referenced by thread-specific
 35531 data values. No such cleanup is done by *pthread_key_delete()*. In particular, destructor functions
 35532 are not called. There are several reasons for this division of responsibility:

- 35533 1. The associated destructor functions used to free thread-specific data at thread exit time
 35534 are only guaranteed to work correctly when called in the thread that allocated the thread-
 35535 specific data. (Destructors themselves may utilize thread-specific data.) Thus, they cannot
 35536 be used to free thread-specific data in other threads at key deletion time. Attempting to
 35537 have them called by other threads at key deletion time would require other threads to be
 35538 asynchronously interrupted. But since interrupted threads could be in an arbitrary state,
 35539 including holding locks necessary for the destructor to run, this approach would fail. In
 35540 general, there is no safe mechanism whereby an implementation could free thread-
 35541 specific data at key deletion time.

35542 2. Even if there were a means of safely freeing thread-specific data associated with keys to
 35543 be deleted, doing so would require that implementations be able to enumerate the
 35544 threads with non-NULL data and potentially keep them from creating more thread-
 35545 specific data while the key deletion is occurring. This special case could cause extra
 35546 synchronization in the normal case, which would otherwise be unnecessary.

35547 For an application to know that it is safe to delete a key, it has to know that all the threads that
 35548 might potentially ever use the key do not attempt to use it again. For example, it could know
 35549 this if all the client threads have called a cleanup procedure declaring that they are through with
 35550 the module that is being shut down, perhaps by setting a reference count to zero.

FUTURE DIRECTIONS

35551 None.
 35552

SEE ALSO

35553 *pthread_key_create()*, the Base Definitions volume of IEEE Std 1003.1-200x, <pthread.h>
 35554

CHANGE HISTORY

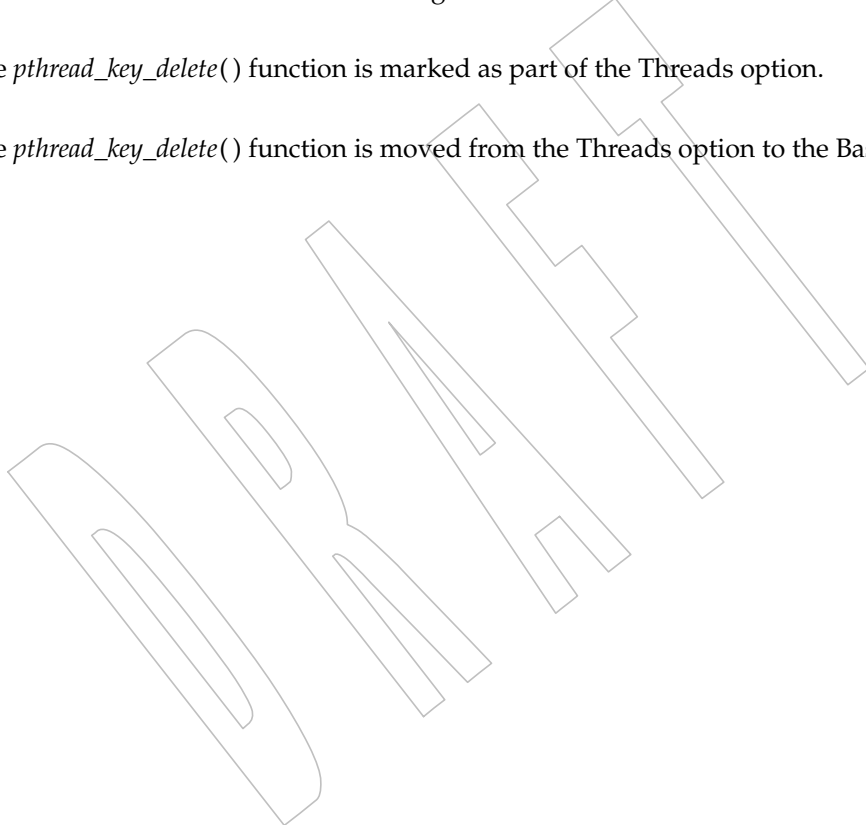
35555 First released in Issue 5. Included for alignment with the POSIX Threads Extension.
 35556

Issue 6

35557 The *pthread_key_delete()* function is marked as part of the Threads option.
 35558

Issue 7

35559 The *pthread_key_delete()* function is moved from the Threads option to the Base.
 35560



35561 **NAME**

35562 pthread_kill — send a signal to a thread

35563 **SYNOPSIS**

```
35564 CX #include <signal.h>
35565 int pthread_kill(pthread_t thread, int sig);
```

35566 **DESCRIPTION**35567 The *pthread_kill()* function shall request that a signal be delivered to the specified thread.35568 As in *kill()*, if *sig* is zero, error checking shall be performed but no signal shall actually be sent.35569 **RETURN VALUE**35570 Upon successful completion, the function shall return a value of zero. Otherwise, the function
35571 shall return an error number. If the *pthread_kill()* function fails, no signal shall be sent.35572 **ERRORS**35573 The *pthread_kill()* function shall fail if:35574 [ESRCH] No thread could be found corresponding to that specified by the given thread
35575 ID.35576 [EINVAL] The value of the *sig* argument is an invalid or unsupported signal number.35577 The *pthread_kill()* function shall not return an error code of [EINTR].35578 **EXAMPLES**

35579 None.

35580 **APPLICATION USAGE**35581 The *pthread_kill()* function provides a mechanism for asynchronously directing a signal at a
35582 thread in the calling process. This could be used, for example, by one thread to affect broadcast
35583 delivery of a signal to a set of threads.35584 Note that *pthread_kill()* only causes the signal to be handled in the context of the given thread;
35585 the signal action (termination or stopping) affects the process as a whole.35586 **RATIONALE**

35587 None.

35588 **FUTURE DIRECTIONS**

35589 None.

35590 **SEE ALSO**35591 *kill()*, *pthread_self()*, *raise()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**35592 **CHANGE HISTORY**

35593 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

35594 **Issue 6**35595 The *pthread_kill()* function is marked as part of the Threads option.

35596 The APPLICATION USAGE section is added.

35597 **Issue 7**35598 The *pthread_kill()* function is moved from the Threads option to the Base.

NAME

pthread_mutex_consistent — mark state protected by robust mutex as consistent

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_consistent(pthread_mutex_t *mutex);
```

DESCRIPTION

If *mutex* is a robust mutex in an inconsistent state, the *pthread_mutex_consistent()* function can be used to mark the state protected by the mutex referenced by *mutex* as consistent again.

If an owner of a robust mutex terminates while holding the mutex, the mutex becomes inconsistent and the next thread that acquires the mutex lock shall be notified of the state by the return value [EOWNERDEAD]. In this case, the mutex does not become normally usable again until the state is marked consistent.

If the thread which acquired the mutex lock with the return value [EOWNERDEAD] terminates before calling either *pthread_mutex_consistent()* or *pthread_mutex_unlock()*, the next thread that acquires the mutex lock shall be notified about the state of the mutex by the return value [EOWNERDEAD].

RETURN VALUE

Upon successful completion, the *pthread_mutex_consistent()* function shall return zero. Otherwise, an error value shall be returned to indicate the error.

ERRORS

The *pthread_mutex_consistent()* function shall fail if:

[EINVAL] The mutex object referenced by *mutex* is not robust or does not protect an inconsistent state.

The *pthread_mutex_consistent()* function may fail if:

[EINVAL] The value *mutex* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_mutex_consistent()* function is only responsible for notifying the implementation that the state protected by the mutex has been recovered and that normal operations with the mutex can be resumed. It is the responsibility of the application to recover the state so it can be reused. If the application is not able to perform the recovery, it can notify the implementation that the situation is unrecoverable by a call to *pthread_mutex_unlock()* without a prior call to *pthread_mutex_consistent()*, in which case subsequent threads that attempt to lock the mutex will fail to acquire the lock and be returned [ENOTRECOVERABLE].

RATIONALE

None.

FUTURE DIRECTIONS

None.

pthread_mutex_consistent()*System Interfaces*

35639

SEE ALSO

35640

pthread_mutex_lock(), *pthread_mutexattr_getrobust()*, the Base Definitions volume of IEEE Std 1003.1-200x, <pthread.h>

35641

35642

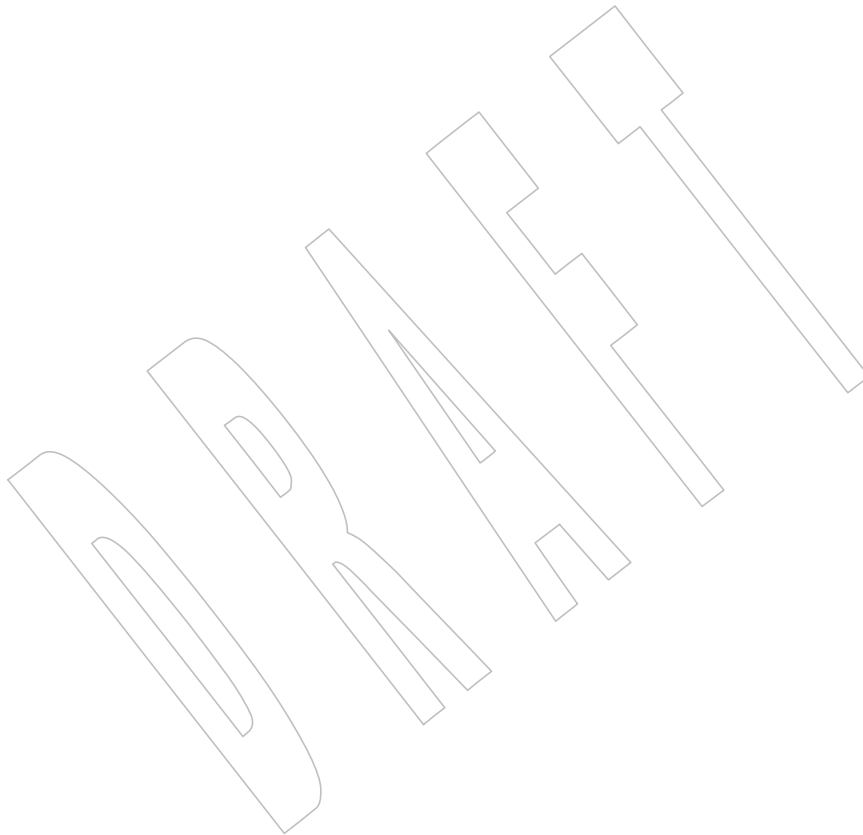
CHANGE HISTORY

35643

First released in Issue 7.

35644

The *pthread_mutex_consistent()* function is moved from the Threads option to the Base.



35645 **NAME**

35646 pthread_mutex_destroy, pthread_mutex_init — destroy and initialize a mutex

35647 **SYNOPSIS**

```
35648 #include <pthread.h>
35649
35649 int pthread_mutex_destroy(pthread_mutex_t *mutex);
35650 int pthread_mutex_init(pthread_mutex_t *restrict mutex,
35651 const pthread_mutexattr_t *restrict attr);
35652 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

35653 **DESCRIPTION**

35654 The *pthread_mutex_destroy()* function shall destroy the mutex object referenced by *mutex*; the
 35655 mutex object becomes, in effect, uninitialized. An implementation may cause
 35656 *pthread_mutex_destroy()* to set the object referenced by *mutex* to an invalid value. A destroyed
 35657 mutex object can be reinitialized using *pthread_mutex_init()*; the results of otherwise referencing
 35658 the object after it has been destroyed are undefined.

35659 It shall be safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked
 35660 mutex results in undefined behavior.

35661 The *pthread_mutex_init()* function shall initialize the mutex referenced by *mutex* with attributes
 35662 specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect shall be the
 35663 same as passing the address of a default mutex attributes object. Upon successful initialization,
 35664 the state of the mutex becomes initialized and unlocked.

35665 Only *mutex* itself may be used for performing synchronization. The result of referring to copies
 35666 of *mutex* in calls to *pthread_mutex_lock()*, *pthread_mutex_trylock()*, *pthread_mutex_unlock()*, and
 35667 *pthread_mutex_destroy()* is undefined.

35668 Attempting to initialize an already initialized mutex results in undefined behavior.

35669 In cases where default mutex attributes are appropriate, the macro
 35670 PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes that are statically allocated.
 35671 The effect shall be equivalent to dynamic initialization by a call to *pthread_mutex_init()* with
 35672 parameter *attr* specified as NULL, except that no error checks are performed.

35673 **RETURN VALUE**

35674 If successful, the *pthread_mutex_destroy()* and *pthread_mutex_init()* functions shall return zero;
 35675 otherwise, an error number shall be returned to indicate the error.

35676 The [EBUSY] and [EINVAL] error checks, if implemented, act as if they were performed
 35677 immediately at the beginning of processing for the function and shall cause an error return prior
 35678 to modifying the state of the mutex specified by *mutex*.

35679 **ERRORS**

35680 The *pthread_mutex_destroy()* function may fail if:

35681 [EBUSY] The implementation has detected an attempt to destroy the object referenced
 35682 by *mutex* while it is locked or referenced (for example, while being used in a
 35683 *pthread_cond_timedwait()* or *pthread_cond_wait()*) by another thread.

35684 [EINVAL] The value specified by *mutex* is invalid.

35685 The *pthread_mutex_init()* function shall fail if:

35686 [EAGAIN] The system lacked the necessary resources (other than memory) to initialize
 35687 another mutex.

- 35688 [ENOMEM] Insufficient memory exists to initialize the mutex.
- 35689 [EPERM] The caller does not have the privilege to perform the operation.
- 35690 The *pthread_mutex_init()* function may fail if:
- 35691 [EBUSY] The implementation has detected an attempt to reinitialize the object
35692 referenced by *mutex*, a previously initialized, but not yet destroyed, mutex.
- 35693 [EINVAL] The value specified by *attr* is invalid.
- 35694 [EINVAL] The attributes object referenced by *attr* has the robust mutex attribute set
35695 without the process-shared attribute being set.
- 35696 These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE**Alternate Implementations Possible**

This volume of IEEE Std 1003.1-200x supports several alternative implementations of mutexes. An implementation may store the lock directly in the object of type **pthread_mutex_t**. Alternatively, an implementation may store the lock in the heap and merely store a pointer, handle, or unique ID in the mutex object. Either implementation has advantages or may be required on certain hardware configurations. So that portable code can be written that is invariant to this choice, this volume of IEEE Std 1003.1-200x does not define assignment or equality for this type, and it uses the term “initialize” to reinforce the (more restrictive) notion that the lock may actually reside in the mutex object itself.

Note that this precludes an over-specification of the type of the mutex or condition variable and motivates the opaqueness of the type.

An implementation is permitted, but not required, to have *pthread_mutex_destroy()* store an illegal value into the mutex. This may help detect erroneous programs that try to lock (or otherwise reference) a mutex that has already been destroyed.

Tradeoff Between Error Checks and Performance Supported

Many of the error checks were made optional in order to let implementations trade off performance *versus* degree of error checking according to the needs of their specific applications and execution environment. As a general rule, errors or conditions caused by the system (such as insufficient memory) always need to be reported, but errors due to an erroneously coded application (such as failing to provide adequate synchronization to prevent a mutex from being deleted while in use) are made optional.

A wide range of implementations is thus made possible. For example, an implementation intended for application debugging may implement all of the error checks, but an implementation running a single, provably correct application under very tight performance constraints in an embedded computer might implement minimal checks. An implementation might even be provided in two versions, similar to the options that compilers provide: a full-checking, but slower version; and a limited-checking, but faster version. To forbid this optionality would be a disservice to users.

By carefully limiting the use of “undefined behavior” only to things that an erroneous (badly coded) application might do, and by defining that resource-not-available errors are mandatory, this volume of IEEE Std 1003.1-200x ensures that a fully-conforming application is portable

35733 across the full range of implementations, while not forcing all implementations to add overhead
 35734 to check for numerous things that a correct program never does.

35735 Why No Limits are Defined

35736 Defining symbols for the maximum number of mutexes and condition variables was considered
 35737 but rejected because the number of these objects may change dynamically. Furthermore, many
 35738 implementations place these objects into application memory; thus, there is no explicit
 35739 maximum.

35740 Static Initializers for Mutexes and Condition Variables

35741 Providing for static initialization of statically allocated synchronization objects allows modules
 35742 with private static synchronization variables to avoid runtime initialization tests and overhead.
 35743 Furthermore, it simplifies the coding of self-initializing modules. Such modules are common in
 35744 C libraries, where for various reasons the design calls for self-initialization instead of requiring
 35745 an explicit module initialization function to be called. An example use of static initialization
 35746 follows.

35747 Without static initialization, a self-initializing routine *foo()* might look as follows:

```
35748 static pthread_once_t foo_once = PTHREAD_ONCE_INIT;
35749 static pthread_mutex_t foo_mutex;

35750 void foo_init()
35751 {
35752     pthread_mutex_init(&foo_mutex, NULL);
35753 }

35754 void foo()
35755 {
35756     pthread_once(&foo_once, foo_init);
35757     pthread_mutex_lock(&foo_mutex);
35758     /* Do work. */
35759     pthread_mutex_unlock(&foo_mutex);
35760 }
```

35761 With static initialization, the same routine could be coded as follows:

```
35762 static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;

35763 void foo()
35764 {
35765     pthread_mutex_lock(&foo_mutex);
35766     /* Do work. */
35767     pthread_mutex_unlock(&foo_mutex);
35768 }
```

35769 Note that the static initialization both eliminates the need for the initialization test inside
 35770 *pthread_once()* and the fetch of *&foo_mutex* to learn the address to be passed to
 35771 *pthread_mutex_lock()* or *pthread_mutex_unlock()*.

35772 Thus, the C code written to initialize static objects is simpler on all systems and is also faster on a
 35773 large class of systems; those where the (entire) synchronization object can be stored in
 35774 application memory.

35775 Yet the locking performance question is likely to be raised for machines that require mutexes to
 35776 be allocated out of special memory. Such machines actually have to have mutexes and possibly
 35777 condition variables contain pointers to the actual hardware locks. For static initialization to work
 35778 on such machines, *pthread_mutex_lock()* also has to test whether or not the pointer to the actual

35779 lock has been allocated. If it has not, *pthread_mutex_lock()* has to initialize it before use. The
 35780 reservation of such resources can be made when the program is loaded, and hence return codes
 35781 have not been added to mutex locking and condition variable waiting to indicate failure to
 35782 complete initialization.

35783 This runtime test in *pthread_mutex_lock()* would at first seem to be extra work; an extra test is
 35784 required to see whether the pointer has been initialized. On most machines this would actually
 35785 be implemented as a fetch of the pointer, testing the pointer against zero, and then using the
 35786 pointer if it has already been initialized. While the test might seem to add extra work, the extra
 35787 effort of testing a register is usually negligible since no extra memory references are actually
 35788 done. As more and more machines provide caches, the real expenses are memory references, not
 35789 instructions executed.

35790 Alternatively, depending on the machine architecture, there are often ways to eliminate *all*
 35791 overhead in the most important case: on the lock operations that occur *after* the lock has been
 35792 initialized. This can be done by shifting more overhead to the less frequent operation:
 35793 initialization. Since out-of-line mutex allocation also means that an address has to be
 35794 dereferenced to find the actual lock, one technique that is widely applicable is to have static
 35795 initialization store a bogus value for that address; in particular, an address that causes a machine
 35796 fault to occur. When such a fault occurs upon the first attempt to lock such a mutex, validity
 35797 checks can be done, and then the correct address for the actual lock can be filled in. Subsequent
 35798 lock operations incur no extra overhead since they do not “fault”. This is merely one technique
 35799 that can be used to support static initialization, while not adversely affecting the performance of
 35800 lock acquisition. No doubt there are other techniques that are highly machine-dependent.

35801 The locking overhead for machines doing out-of-line mutex allocation is thus similar for
 35802 modules being implicitly initialized, where it is improved for those doing mutex allocation
 35803 entirely inline. The inline case is thus made much faster, and the out-of-line case is not
 35804 significantly worse.

35805 Besides the issue of locking performance for such machines, a concern is raised that it is possible
 35806 that threads would serialize contending for initialization locks when attempting to finish
 35807 initializing statically allocated mutexes. (Such finishing would typically involve taking an
 35808 internal lock, allocating a structure, storing a pointer to the structure in the mutex, and releasing
 35809 the internal lock.) First, many implementations would reduce such serialization by hashing on
 35810 the mutex address. Second, such serialization can only occur a bounded number of times. In
 35811 particular, it can happen at most as many times as there are statically allocated synchronization
 35812 objects. Dynamically allocated objects would still be initialized via *pthread_mutex_init()* or
 35813 *pthread_cond_init()*.

35814 Finally, if none of the above optimization techniques for out-of-line allocation yields sufficient
 35815 performance for an application on some implementation, the application can avoid static
 35816 initialization altogether by explicitly initializing all synchronization objects with the
 35817 corresponding *pthread_*_init()* functions, which are supported by all implementations. An
 35818 implementation can also document the tradeoffs and advise which initialization technique is
 35819 more efficient for that particular implementation.

35820 **Destroying Mutexes**

35821 A mutex can be destroyed immediately after it is unlocked. For example, consider the following
 35822 code:

```
35823 struct obj {
35824     pthread_mutex_t om;
35825     int refcnt;
35826     ...
35827 };
```

```

35828     obj_done(struct obj *op)
35829     {
35830         pthread_mutex_lock(&op->om);
35831         if (--op->refcnt == 0) {
35832             pthread_mutex_unlock(&op->om);
35833         (A)     pthread_mutex_destroy(&op->om);
35834         (B)     free(op);
35835         } else
35836         (C)     pthread_mutex_unlock(&op->om);
35837     }

```

35838 In this case *obj* is reference counted and *obj_done()* is called whenever a reference to the object is
35839 dropped. Implementations are required to allow an object to be destroyed and freed and
35840 potentially unmapped (for example, lines A and B) immediately after the object is unlocked (line
35841 C).

35842 **Robust Mutexes**

35843 Implementations are required to provide robust mutexes for mutexes with the process-shared
35844 attribute set to PTHREAD_PROCESS_SHARED. Implementations are allowed, but not required,
35845 to provide robust mutexes when the process-shared attribute is set to
35846 PTHREAD_PROCESS_PRIVATE.

35847 **FUTURE DIRECTIONS**

35848 None.

35849 **SEE ALSO**

35850 [pthread_mutex_getprioceiling\(\)](#), [pthread_mutexattr_getrobust\(\)](#), [pthread_mutex_lock\(\)](#),
35851 [pthread_mutex_timedlock\(\)](#), [pthread_mutexattr_getpshared\(\)](#), the Base Definitions volume of
35852 IEEE Std 1003.1-200x, [<pthread.h>](#)

35853 **CHANGE HISTORY**

35854 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

35855 **Issue 6**

35856 The [pthread_mutex_destroy\(\)](#) and [pthread_mutex_init\(\)](#) functions are marked as part of the
35857 Threads option.

35858 The [pthread_mutex_timedlock\(\)](#) function is added to the SEE ALSO section for alignment with
35859 IEEE Std 1003.1d-1999.

35860 IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION.

35861 The **restrict** keyword is added to the [pthread_mutex_init\(\)](#) prototype for alignment with the
35862 ISO/IEC 9899: 1999 standard.

35863 **Issue 7**

35864 Changes are made from The Open Group Technical Standard, 2006, Extended API Set Part 3.

35865 The [pthread_mutex_destroy\(\)](#) and [pthread_mutex_init\(\)](#) functions are moved from the Threads
35866 option to the Base.

35867 **NAME**

35868 pthread_mutex_getprioceiling, pthread_mutex_setprioceiling — get and set the priority ceiling
 35869 of a mutex (**REALTIME THREADS**)

35870 **SYNOPSIS**

```
35871 RPP|TPP #include <pthread.h>
35872
35873 int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict mutex,
35874 int *restrict prioceiling);
35875
35876 int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
35877 int prioceiling, int *restrict old_ceiling);
```

35876 **DESCRIPTION**

35877 The *pthread_mutex_getprioceiling()* function shall return the current priority ceiling of the mutex.

35878 The *pthread_mutex_setprioceiling()* function shall either lock the mutex if it is unlocked, or block
 35879 until it can successfully lock the mutex, then it shall change the mutex's priority ceiling and
 35880 release the mutex. When the change is successful, the previous value of the priority ceiling shall
 35881 be returned in *old_ceiling*. The process of locking the mutex need not adhere to the priority
 35882 protect protocol.

35883 If *pthread_mutex_setprioceiling()* is called while holding the mutex, the result is undefined unless
 35884 the mutex is of type PTHREAD_MUTEX_RECURSIVE.

35885 If the *pthread_mutex_setprioceiling()* function fails, the mutex priority ceiling shall not be
 35886 changed.

35887 **RETURN VALUE**

35888 If successful, the *pthread_mutex_getprioceiling()* and *pthread_mutex_setprioceiling()* functions shall
 35889 return zero; otherwise, an error number shall be returned to indicate the error.

35890 **ERRORS**

35891 These functions shall fail if:

35892 [EINVAL] The protocol attribute of *mutex* is PTHREAD_PRIO_NONE.

35893 These functions may fail if:

35894 [EDEADLK] The current thread already owns the mutex.

35895 [EINVAL] The priority requested by *prioceiling* is out of range.

35896 [EINVAL] The value specified by *mutex* does not refer to a currently existing mutex.

35897 [EPERM] The caller does not have the privilege to perform the operation.

35898 These functions shall not return an error code of [EINTR].

35899 **EXAMPLES**

35900 None.

35901 **APPLICATION USAGE**

35902 None.

35903 **RATIONALE**

35904 None.

35905
35906
35907
35908
35909
35910
35911
35912
35913
35914
35915
35916
35917
35918
35919
35920
35921
35922
35923
35924
35925
35926
35927
35928
35929

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_destroy(), *pthread_mutex_lock()*, *pthread_mutex_timedlock()*, the Base Definitions volume of IEEE Std 1003.1-200x, <pthread.h>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

Issue 6

The *pthread_mutex_getprioceiling()* and *pthread_mutex_setprioceiling()* functions are marked as part of the Threads and Thread Priority Protection options.

The [ENOSYS] error conditions have been removed.

The *pthread_mutex_timedlock()* function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the *pthread_mutex_getprioceiling()* and *pthread_mutex_setprioceiling()* prototypes for alignment with the ISO/IEC 9899:1999 standard.

Issue 7

SD5-XSH-ERN-39 is applied.

Austin Group Interpretation 1003.1-2001 #052 is applied, adding [EDEADLK] as a “may fail” error.

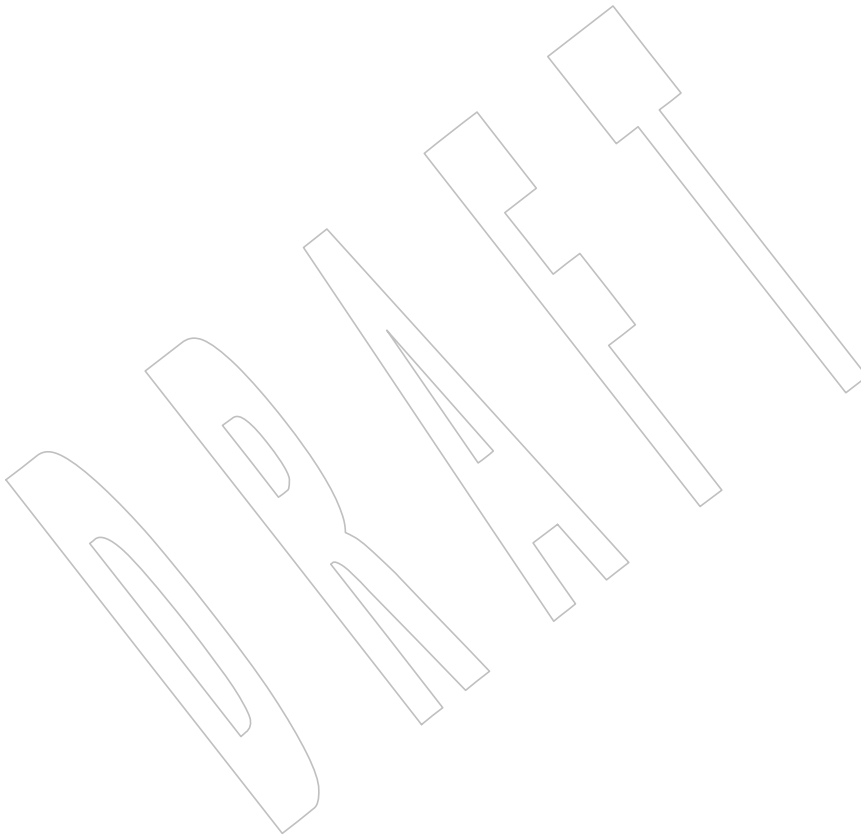
SD5-XSH-ERN-158 is applied, updating the ERRORS section to include a “shall fail” error case for when the protocol attribute of *mutex* is PTHREAD_PRIO_NONE.

The *pthread_mutex_getprioceiling()* and *pthread_mutex_setprioceiling()* functions are moved from the Threads option to require support of either the Robust Mutex Priority Protection option or the Non-Robust Mutex Priority Protection option.

35930 **NAME**
35931 pthread_mutex_init — destroy and initialize a mutex

35932 **SYNOPSIS**
35933 #include <pthread.h>
35934 int pthread_mutex_init(pthread_mutex_t *restrict mutex,
35935 const pthread_mutexattr_t *restrict attr);
35936 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

35937 **DESCRIPTION**
35938 Refer to *pthread_mutex_destroy()*.



35939 **NAME**

35940 pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a
35941 mutex

35942 **SYNOPSIS**

```
35943 #include <pthread.h>

35944 int pthread_mutex_lock(pthread_mutex_t *mutex);
35945 int pthread_mutex_trylock(pthread_mutex_t *mutex);
35946 int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

35947 **DESCRIPTION**

35948 The mutex object referenced by *mutex* shall be locked by calling *pthread_mutex_lock()*. If the
35949 mutex is already locked, the calling thread shall block until the mutex becomes available. This
35950 operation shall return with the mutex object referenced by *mutex* in the locked state with the
35951 calling thread as its owner.

35952 If the mutex type is PTHREAD_MUTEX_NORMAL, deadlock detection shall not be provided.
35953 Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it
35954 has not locked or a mutex which is unlocked, undefined behavior results.

35955 If the mutex type is PTHREAD_MUTEX_ERRORCHECK, then error checking shall be provided.
35956 If a thread attempts to relock a mutex that it has already locked, an error shall be returned. If a
35957 thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error
35958 shall be returned.

35959 If the mutex type is PTHREAD_MUTEX_RECURSIVE, then the mutex shall maintain the
35960 concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock
35961 count shall be set to one. Every time a thread relocks this mutex, the lock count shall be
35962 incremented by one. Each time the thread unlocks the mutex, the lock count shall be
35963 decremented by one. When the lock count reaches zero, the mutex shall become available for
35964 other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex
35965 which is unlocked, an error shall be returned.

35966 If the mutex type is PTHREAD_MUTEX_DEFAULT, attempting to recursively lock the mutex
35967 results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling
35968 thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in
35969 undefined behavior.

35970 The *pthread_mutex_trylock()* function shall be equivalent to *pthread_mutex_lock()*, except that if
35971 the mutex object referenced by *mutex* is currently locked (by any thread, including the current
35972 thread), the call shall return immediately. If the mutex type is PTHREAD_MUTEX_RECURSIVE
35973 and the mutex is currently owned by the calling thread, the mutex lock count shall be
35974 incremented by one and the *pthread_mutex_trylock()* function shall immediately return success.

35975 The *pthread_mutex_unlock()* function shall release the mutex object referenced by *mutex*. The
35976 manner in which a mutex is released is dependent upon the mutex's type attribute. If there are
35977 threads blocked on the mutex object referenced by *mutex* when *pthread_mutex_unlock()* is called,
35978 resulting in the mutex becoming available, the scheduling policy shall determine which thread
35979 shall acquire the mutex.

35980 (In the case of PTHREAD_MUTEX_RECURSIVE mutexes, the mutex shall become available
35981 when the count reaches zero and the calling thread no longer has any locks on this mutex.)

35982 If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the
35983 thread shall resume waiting for the mutex as if it was not interrupted.

35984 If *mutex* is a robust mutex and the process containing the owning thread terminated while

pthread_mutex_lock()

System Interfaces

35985 holding the mutex lock, a call to *pthread_mutex_lock()* shall return the error value
 35986 [EOWNERDEAD]. If *mutex* is a robust mutex and the owning thread terminated while holding
 35987 the mutex lock, a call to *pthread_mutex_lock()* may return the error value [EOWNERDEAD] even
 35988 if the process in which the owning thread resides has not terminated. In these cases, the mutex is
 35989 locked by the thread but the state it protects is marked as inconsistent. The application should
 35990 ensure that the state is made consistent for reuse and when that is complete call
 35991 *pthread_mutex_consistent()*. If the application is unable to recover the state, it should unlock the
 35992 mutex without a prior call to *pthread_mutex_consistent()*, after which the mutex is marked
 35993 permanently unusable.

RETURN VALUE

35994 If successful, the *pthread_mutex_lock()* and *pthread_mutex_unlock()* functions shall return zero;
 35995 otherwise, an error number shall be returned to indicate the error.

35997 The *pthread_mutex_trylock()* function shall return zero if a lock on the mutex object referenced by
 35998 *mutex* is acquired. Otherwise, an error number is returned to indicate the error.

ERRORS

35999 The *pthread_mutex_lock()* and *pthread_mutex_trylock()* functions shall fail if:

36000 RPP|TPP [EINVAL] The *mutex* was created with the protocol attribute having the value
 36001 PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than
 36002 the mutex's current priority ceiling.
 36003

36004 [ENOTRECOVERABLE]

36005 The state protected by the mutex is not recoverable. The mutex is not locked.

36006 [EOWNERDEAD]

36007 The mutex is a robust mutex and the process containing the previous owning
 36008 thread terminated while holding the mutex lock. The mutex lock has been
 36009 acquired and it is up to the new owner to make the state consistent.

36010 The *pthread_mutex_trylock()* function shall fail if:

36011 [EBUSY] The *mutex* could not be acquired because it was already locked.

36012 The *pthread_mutex_unlock()* function shall fail if:

36013 [EPERM] The current thread does not own the mutex and the mutex is a robust mutex.

36014 The *pthread_mutex_lock()*, *pthread_mutex_trylock()*, and *pthread_mutex_unlock()* functions may fail
 36015 if:

36016 [EINVAL] The value specified by *mutex* does not refer to an initialized mutex object.

36017 [EAGAIN] The mutex could not be acquired because the maximum number of recursive
 36018 locks for *mutex* has been exceeded.

36019 The *pthread_mutex_lock()* and *pthread_mutex_trylock()* functions may fail if:

36020 [EOWNERDEAD]

36021 The mutex is a robust mutex and the previous owning thread terminated
 36022 while holding the mutex lock. The mutex lock has been acquired and it is up
 36023 to the new owner to make the state consistent.

36024 The *pthread_mutex_lock()* function may fail if:

36025 [EDEADLK] A deadlock condition was detected or the current thread already owns the
 36026 mutex.

36027 The *pthread_mutex_unlock()* function may fail if:

36028 [EPERM] The current thread does not own the mutex.

36029 These functions shall not return an error code of [EINTR].

EXAMPLES

36030 None.

APPLICATION USAGE

36033 Applications that have assumed that non-zero return values are errors will need updating for
 36034 use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting
 36035 a currently inconsistent state is [EOWNERDEAD]. Applications that do not check the error
 36036 returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If
 36037 an application is supposed to work with normal and robust mutexes it should check all return
 36038 values for error conditions and if necessary take appropriate action.

RATIONALE

36039 Mutex objects are intended to serve as a low-level primitive from which other thread
 36040 synchronization functions can be built. As such, the implementation of mutexes should be as
 36041 efficient as possible, and this has ramifications on the features available at the interface.
 36042

36043 The mutex functions and the particular default settings of the mutex attributes have been
 36044 motivated by the desire to not preclude fast, inlined implementations of mutex locking and
 36045 unlocking.

36046 Since most attributes only need to be checked when a thread is going to be blocked, the use of
 36047 attributes does not slow the (common) mutex-locking case.

36048 Likewise, while being able to extract the thread ID of the owner of a mutex might be desirable, it
 36049 would require storing the current thread ID when each mutex is locked, and this could incur
 36050 unacceptable levels of overhead. Similar arguments apply to a *mutex_tryunlock* operation.

36051 For further rationale on the extended mutex types, see the Rationale (Informative) volume of
 36052 IEEE Std 1003.1-200x.

FUTURE DIRECTIONS

36053 None.

SEE ALSO

36054
 36055 *pthread_mutex_consistent()*, *pthread_mutex_destroy()*, *pthread_mutex_timedlock()*,
 36056 *pthread_mutexattr_getrobust()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10,
 36057 Memory Synchronization, <pthread.h>

CHANGE HISTORY

36059 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

36061 The *pthread_mutex_lock()*, *pthread_mutex_trylock()*, and *pthread_mutex_unlock()* functions are
 36062 marked as part of the Threads option.
 36063

36064 The following new requirements on POSIX implementations derive from alignment with the
 36065 Single UNIX Specification:

- The behavior when attempting to relock a mutex is defined.

36067 The *pthread_mutex_timedlock()* function is added to the SEE ALSO section for alignment with
 36068 IEEE Std 1003.1d-1999.

36069 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/98 is applied, updating the ERRORS
 36070 section so that the [EDEADLK] error includes detection of a deadlock condition. The
 36071 RATIONALE section is also reworded to take into account non-XSI-conformant systems.

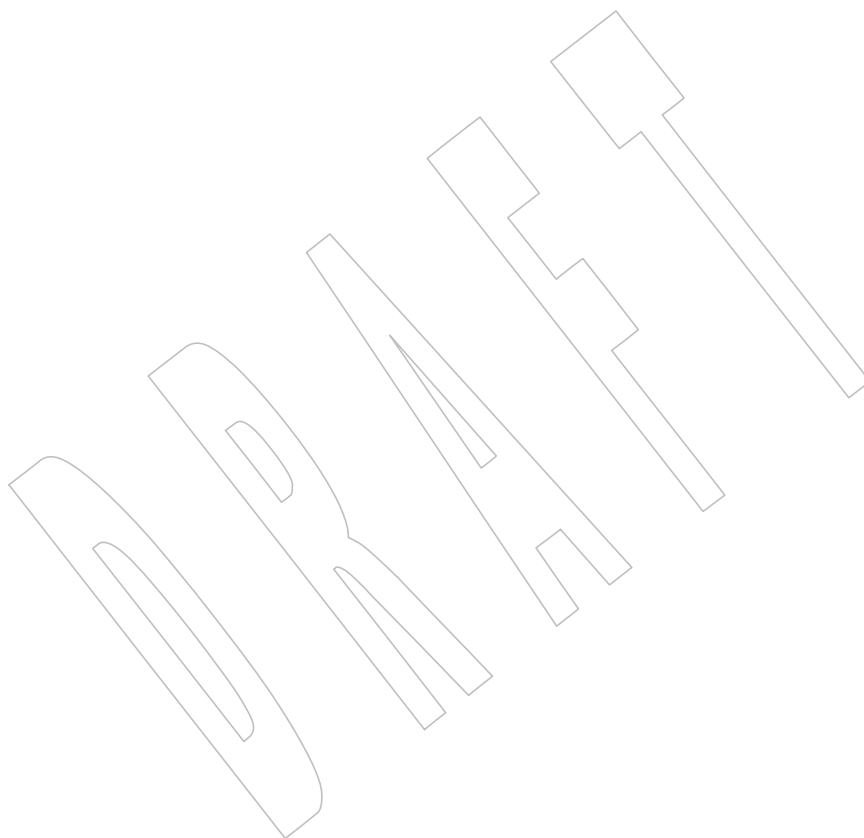
pthread_mutex_lock()36072 **Issue 7**

36073 SD5-XSH-ERN-43 is applied, marking the “shall fail” case of the [EINVAL] error as dependent
36074 on the Thread Priority Protection option.

36075 Changes are made from The Open Group Technical Standard, 2006, Extended API Set Part 3.

36076 The *pthread_mutex_lock()*, *pthread_mutex_trylock()*, and *pthread_mutex_unlock()* functions are
36077 moved from the Threads option to the Base.

36078 The PTHREAD_MUTEX_NORMAL, PTHREAD_MUTEX_ERRORCHECK,
36079 PTHREAD_MUTEX_RECURSIVE, and PTHREAD_MUTEX_DEFAULT extended mutex types
36080 are moved from the XSI option to the Base.



36081 **NAME**

36082 pthread_mutex_setprioceiling — change the priority ceiling of a mutex (**REALTIME**
36083 **THREADS**)

36084 **SYNOPSIS**

```
36085 RPP|TPP #include <pthread.h>  
36086 int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,  
36087 int prioceiling, int *restrict old_ceiling);
```

36088 **DESCRIPTION**

36089 Refer to [pthread_mutex_getprioceiling\(\)](#).

36090 **NAME**

36091 pthread_mutex_timedlock — lock a mutex

36092 **SYNOPSIS**

36093 #include <pthread.h>

36094 #include <time.h>

```
36095 int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
36096                             const struct timespec *restrict abs_timeout);
```

36097 **DESCRIPTION**

36098 The *pthread_mutex_timedlock()* function shall lock the mutex object referenced by *mutex*. If the
 36099 mutex is already locked, the calling thread shall block until the mutex becomes available as in
 36100 the *pthread_mutex_lock()* function. If the mutex cannot be locked without waiting for another
 36101 thread to unlock the mutex, this wait shall be terminated when the specified timeout expires.

36102 The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by
 36103 the clock on which timeouts are based (that is, when the value of that clock equals or exceeds
 36104 *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time
 36105 of the call.

36106 The timeout shall be based on the CLOCK_REALTIME clock. The resolution of the timeout shall
 36107 be the resolution of the clock on which it is based. The **timespec** data type is defined in the
 36108 <**time.h**> header.

36109 Under no circumstance shall the function fail with a timeout if the mutex can be locked
 36110 immediately. The validity of the *abs_timeout* parameter need not be checked if the mutex can be
 36111 locked immediately.

36112 RPI | TPI As a consequence of the priority inheritance rules (for mutexes initialized with the
 36113 PRIO_INHERIT protocol), if a timed mutex wait is terminated because its timeout expires, the
 36114 priority of the owner of the mutex shall be adjusted as necessary to reflect the fact that this
 36115 thread is no longer among the threads waiting for the mutex.

36116 If *mutex* is a robust mutex and the process containing the owning thread terminated while
 36117 holding the mutex lock, a call to *pthread_mutex_timedlock()* shall return the error value
 36118 [EOWNERDEAD]. If *mutex* is a robust mutex and the owning thread terminated while holding
 36119 the mutex lock, a call to *pthread_mutex_timedlock()* may return the error value [EOWNERDEAD]
 36120 even if the process in which the owning thread resides has not terminated. In these cases, the
 36121 mutex is locked by the thread but the state it protects is marked as inconsistent. The application
 36122 should ensure that the state is made consistent for reuse and when that is complete call
 36123 *pthread_mutex_consistent()*. If the application is unable to recover the state, it should unlock the
 36124 mutex without a prior call to *pthread_mutex_consistent()*, after which the mutex is marked
 36125 permanently unusable.

36126 **RETURN VALUE**

36127 If successful, the *pthread_mutex_timedlock()* function shall return zero; otherwise, an error
 36128 number shall be returned to indicate the error.

36129 **ERRORS**

36130 The *pthread_mutex_timedlock()* function shall fail if:

36131 [EINVAL] The mutex was created with the protocol attribute having the value
 36132 PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than
 36133 the mutex' current priority ceiling.

- 36134 [EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter
 36135 specified a nanoseconds field value less than zero or greater than or equal to
 36136 1 000 million.
- 36137 [ENOTRECOVERABLE]
 36138 The state protected by the mutex is not recoverable. The mutex is not locked.
- 36139 [EOWNERDEAD]
 36140 The mutex is a robust mutex and the process containing the previous owning
 36141 thread terminated while holding the mutex lock. The mutex lock has been
 36142 acquired and it is up to the new owner to make the state consistent.
- 36143 [ETIMEDOUT] The mutex could not be locked before the specified timeout expired.
- 36144 The *pthread_mutex_timedlock()* function may fail if:
- 36145 [EINVAL] The value specified by *mutex* does not refer to an initialized mutex object.
- 36146 [EAGAIN] The mutex could not be acquired because the maximum number of recursive
 36147 locks for *mutex* has been exceeded.
- 36148 [EDEADLK] A deadlock condition was detected or the current thread already owns the
 36149 mutex.
- 36150 [EOWNERDEAD]
 36151 The mutex is a robust mutex and the previous owning thread terminated
 36152 while holding the mutex lock. The mutex lock has been acquired and it is up
 36153 to the new owner to make the state consistent.
- 36154 This function shall not return an error code of [EINTR].

EXAMPLES

36155 None.
 36156

APPLICATION USAGE

36157 Applications that have assumed that non-zero return values are errors will need updating for
 36158 use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting
 36159 a currently inconsistent state is [EOWNERDEAD]. Applications that do not check the error
 36160 returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If
 36161 an application is supposed to work with normal and robust mutexes, it should check all return
 36162 values for error conditions and if necessary take appropriate action.
 36163

RATIONALE

36164 None.
 36165

FUTURE DIRECTIONS

36166 None.
 36167

SEE ALSO

36168 [pthread_mutex_destroy\(\)](#), [pthread_mutex_lock\(\)](#), [pthread_mutex_trylock\(\)](#), [time\(\)](#), the Base
 36169 Definitions volume of IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization,
 36170 [<pthread.h>](#), [<time.h>](#)
 36171

CHANGE HISTORY

36172 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.
 36173

36174 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/99 is applied, marking the last paragraph
 36175 in the DESCRIPTION as part of the Thread Priority Inheritance option.

36176 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/100 is applied, updating the ERRORS
 36177 section so that the [EDEADLK] error includes detection of a deadlock condition.

pthread_mutex_timedlock()

36178

Issue 7

36179

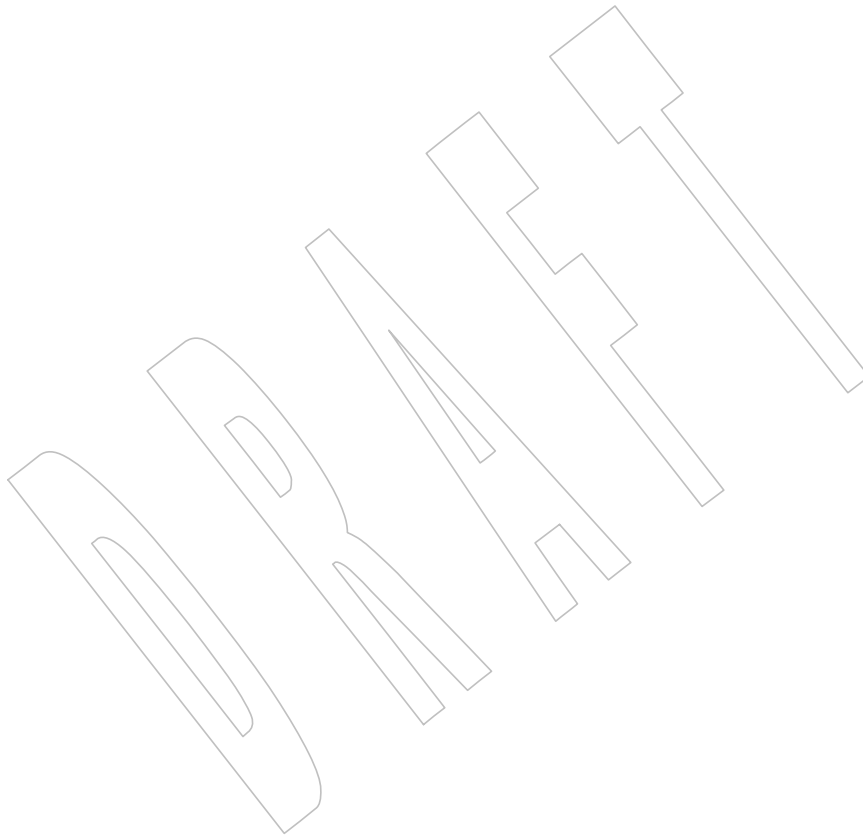
Changes are made from The Open Group Technical Standard, 2006, Extended API Set Part 3.

36180

The *pthread_mutex_timedlock()* function is moved from the Timeouts option to the Base.

36181

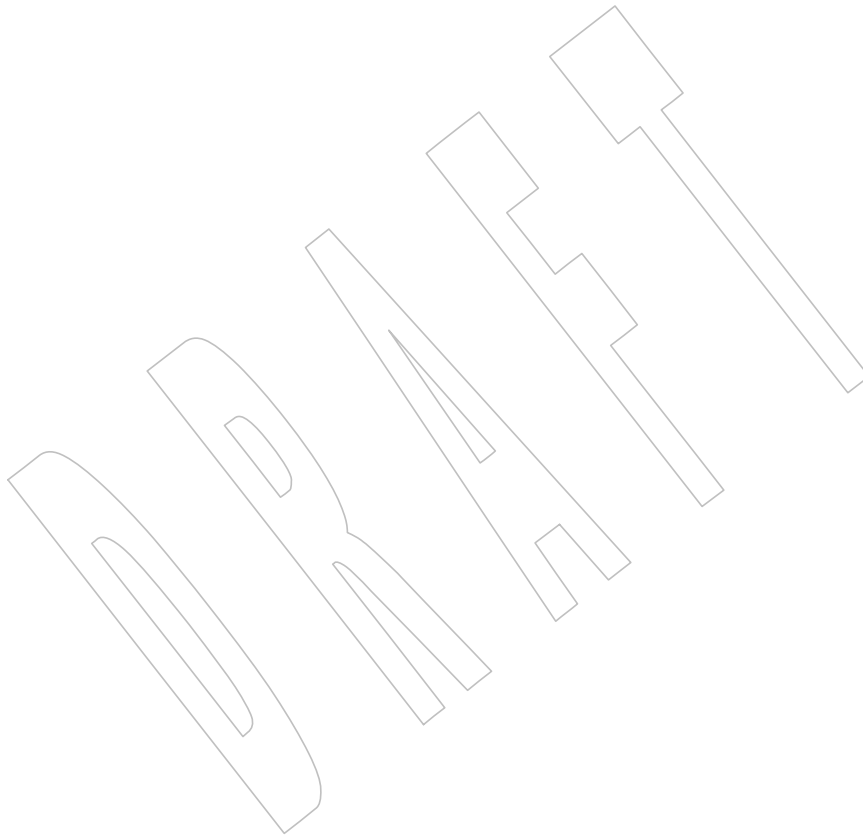
Functionality relating to the Timers option is moved to the Base.



36182 **NAME**
36183 pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a mutex

36184 **SYNOPSIS**
36185 #include <pthread.h>
36186 int pthread_mutex_trylock(pthread_mutex_t *mutex);
36187 int pthread_mutex_unlock(pthread_mutex_t *mutex);

36188 **DESCRIPTION**
36189 Refer to *pthread_mutex_lock()*.



36190 **NAME**

36191 pthread_mutexattr_destroy, pthread_mutexattr_init — destroy and initialize the mutex
 36192 attributes object

36193 **SYNOPSIS**

36194 #include <pthread.h>

36195 int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
 36196 int pthread_mutexattr_init(pthread_mutexattr_t *attr);

36197 **DESCRIPTION**

36198 The *pthread_mutexattr_destroy()* function shall destroy a mutex attributes object; the object
 36199 becomes, in effect, uninitialized. An implementation may cause *pthread_mutexattr_destroy()* to
 36200 set the object referenced by *attr* to an invalid value. A destroyed *attr* attributes object can be
 36201 reinitialized using *pthread_mutexattr_init()*; the results of otherwise referencing the object after it
 36202 has been destroyed are undefined.

36203 The *pthread_mutexattr_init()* function shall initialize a mutex attributes object *attr* with the
 36204 default value for all of the attributes defined by the implementation.

36205 Results are undefined if *pthread_mutexattr_init()* is called specifying an already initialized *attr*
 36206 attributes object.

36207 After a mutex attributes object has been used to initialize one or more mutexes, any function
 36208 affecting the attributes object (including destruction) shall not affect any previously initialized
 36209 mutexes.

36210 **RETURN VALUE**

36211 Upon successful completion, *pthread_mutexattr_destroy()* and *pthread_mutexattr_init()* shall
 36212 return zero; otherwise, an error number shall be returned to indicate the error.

36213 **ERRORS**

36214 The *pthread_mutexattr_destroy()* function may fail if:

36215 [EINVAL] The value specified by *attr* is invalid.

36216 The *pthread_mutexattr_init()* function shall fail if:

36217 [ENOMEM] Insufficient memory exists to initialize the mutex attributes object.

36218 These functions shall not return an error code of [EINTR].

36219 **EXAMPLES**

36220 None.

36221 **APPLICATION USAGE**

36222 None.

36223 **RATIONALE**

36224 See *pthread_attr_init()* for a general explanation of attributes. Attributes objects allow
 36225 implementations to experiment with useful extensions and permit extension of this volume of
 36226 IEEE Std 1003.1-200x without changing the existing functions. Thus, they provide for future
 36227 extensibility of this volume of IEEE Std 1003.1-200x and reduce the temptation to standardize
 36228 prematurely on semantics that are not yet widely implemented or understood.

36229 Examples of possible additional mutex attributes that have been discussed are *spin_only*,
 36230 *limited_spin*, *no_spin*, *recursive*, and *metered*. (To explain what the latter attributes might mean:
 36231 recursive mutexes would allow for multiple re-locking by the current owner; metered mutexes
 36232 would transparently keep records of queue length, wait time, and so on.) Since there is not yet
 36233 wide agreement on the usefulness of these resulting from shared implementation and usage

36234 experience, they are not yet specified in this volume of IEEE Std 1003.1-200x. Mutex attributes
 36235 objects, however, make it possible to test out these concepts for possible standardization at a
 36236 later time.

36237 **Mutex Attributes and Performance**

36238 Care has been taken to ensure that the default values of the mutex attributes have been defined
 36239 such that mutexes initialized with the defaults have simple enough semantics so that the locking
 36240 and unlocking can be done with the equivalent of a test-and-set instruction (plus possibly a few
 36241 other basic instructions).

36242 There is at least one implementation method that can be used to reduce the cost of testing at
 36243 lock-time if a mutex has non-default attributes. One such method that an implementation can
 36244 employ (and this can be made fully transparent to fully conforming POSIX applications) is to
 36245 secretly pre-lock any mutexes that are initialized to non-default attributes. Any later attempt to
 36246 lock such a mutex causes the implementation to branch to the “slow path” as if the mutex were
 36247 unavailable; then, on the slow path, the implementation can do the “real work” to lock a non-
 36248 default mutex. The underlying unlock operation is more complicated since the implementation
 36249 never really wants to release the pre-lock on this kind of mutex. This illustrates that, depending
 36250 on the hardware, there may be certain optimizations that can be used so that whatever mutex
 36251 attributes are considered “most frequently used” can be processed most efficiently.

36252 **Process Shared Memory and Synchronization**

36253 The existence of memory mapping functions in this volume of IEEE Std 1003.1-200x leads to the
 36254 possibility that an application may allocate the synchronization objects from this section in
 36255 memory that is accessed by multiple processes (and therefore, by threads of multiple processes).

36256 In order to permit such usage, while at the same time keeping the usual case (that is, usage
 36257 within a single process) efficient, a *process-shared* option has been defined.

36258 If an implementation supports the `_POSIX_THREAD_PROCESS_SHARED` option, then the
 36259 *process-shared* attribute can be used to indicate that mutexes or condition variables may be
 36260 accessed by threads of multiple processes.

36261 The default setting of `PTHREAD_PROCESS_PRIVATE` has been chosen for the *process-shared*
 36262 attribute so that the most efficient forms of these synchronization objects are created by default.

36263 Synchronization variables that are initialized with the `PTHREAD_PROCESS_PRIVATE` *process-*
 36264 *shared* attribute may only be operated on by threads in the process that initialized them.
 36265 Synchronization variables that are initialized with the `PTHREAD_PROCESS_SHARED` *process-*
 36266 *shared* attribute may be operated on by any thread in any process that has access to it. In
 36267 particular, these processes may exist beyond the lifetime of the initializing process. For example,
 36268 the following code implements a simple counting semaphore in a mapped file that may be used
 36269 by many processes.

```
36270 /* sem.h */
36271 struct semaphore {
36272     pthread_mutex_t lock;
36273     pthread_cond_t nonzero;
36274     unsigned count;
36275 };
36276 typedef struct semaphore semaphore_t;
36277
36278 semaphore_t *semaphore_create(char *semaphore_name);
36279 semaphore_t *semaphore_open(char *semaphore_name);
36280 void semaphore_post(semaphore_t *semap);
36281 void semaphore_wait(semaphore_t *semap);
36282 void semaphore_close(semaphore_t *semap);
```

```

36282     /* sem.c */
36283     #include <sys/types.h>
36284     #include <sys/stat.h>
36285     #include <sys/mman.h>
36286     #include <fcntl.h>
36287     #include <pthread.h>
36288     #include "sem.h"

36289     semaphore_t *
36290     semaphore_create(char *semaphore_name)
36291     {
36292     int fd;
36293     semaphore_t *semap;
36294     pthread_mutexattr_t psharedm;
36295     pthread_condattr_t psharedc;

36296     fd = open(semaphore_name, O_RDWR | O_CREAT | O_EXCL, 0666);
36297     if (fd < 0)
36298         return (NULL);
36299     (void) ftruncate(fd, sizeof(semaphore_t));
36300     (void) pthread_mutexattr_init(&psharedm);
36301     (void) pthread_mutexattr_setpshared(&psharedm,
36302         PTHREAD_PROCESS_SHARED);
36303     (void) pthread_condattr_init(&psharedc);
36304     (void) pthread_condattr_setpshared(&psharedc,
36305         PTHREAD_PROCESS_SHARED);
36306     semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
36307         PROT_READ | PROT_WRITE, MAP_SHARED,
36308         fd, 0);
36309     close (fd);
36310     (void) pthread_mutex_init(&semap->lock, &psharedm);
36311     (void) pthread_cond_init(&semap->nonzero, &psharedc);
36312     semap->count = 0;
36313     return (semap);
36314     }

36315     semaphore_t *
36316     semaphore_open(char *semaphore_name)
36317     {
36318     int fd;
36319     semaphore_t *semap;

36320     fd = open(semaphore_name, O_RDWR, 0666);
36321     if (fd < 0)
36322         return (NULL);
36323     semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
36324         PROT_READ | PROT_WRITE, MAP_SHARED,
36325         fd, 0);
36326     close (fd);
36327     return (semap);
36328     }

36329     void
36330     semaphore_post(semaphore_t *semap)
36331     {
36332     pthread_mutex_lock(&semap->lock);
36333     if (semap->count == 0)

```

```

36334         pthread_cond_signal(&semapx->nonzero);
36335     semap->count++;
36336     pthread_mutex_unlock(&semap->lock);
36337 }
36338
36339 void
36340 semaphore_wait(semaphore_t *semap)
36341 {
36342     pthread_mutex_lock(&semap->lock);
36343     while (semap->count == 0)
36344         pthread_cond_wait(&semap->nonzero, &semap->lock);
36345     semap->count--;
36346     pthread_mutex_unlock(&semap->lock);
36347 }

```

```

36347 void
36348 semaphore_close(semaphore_t *semap)
36349 {
36350     munmap((void *) semap, sizeof(semaphore_t));
36351 }

```

36352 The following code is for three separate processes that create, post, and wait on a semaphore in
36353 the file **/tmp/semaphore**. Once the file is created, the post and wait programs increment and
36354 decrement the counting semaphore (waiting and waking as required) even though they did not
36355 initialize the semaphore.

```

36356 /* create.c */
36357 #include "pthread.h"
36358 #include "sem.h"
36359
36360 int
36361 main()
36362 {
36363     semaphore_t *semap;
36364     semap = semaphore_create("/tmp/semaphore");
36365     if (semap == NULL)
36366         exit(1);
36367     semaphore_close(semap);
36368     return (0);
36369 }
36370
36371 /* post.c */
36372 #include "pthread.h"
36373 #include "sem.h"
36374
36375 int
36376 main()
36377 {
36378     semaphore_t *semap;
36379
36380     semap = semaphore_open("/tmp/semaphore");
36381     if (semap == NULL)
36382         exit(1);
36383     semaphore_post(semap);
36384     semaphore_close(semap);
36385     return (0);
36386 }

```

```

36383     /* wait */
36384     #include "pthread.h"
36385     #include "sem.h"
36386
36387     int
36388     main()
36389     {
36390         semaphore_t *semap;
36391
36392         semap = semaphore_open("/tmp/semaphore");
36393         if (semap == NULL)
36394             exit(1);
36395         semaphore_wait(semap);
36396         semaphore_close(semap);
36397         return (0);
36398     }

```

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_destroy(), *pthread_create()*, *pthread_mutex_destroy()*, *pthread_mutexattr_destroy()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**pthread.h**>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_mutexattr_destroy()* and *pthread_mutexattr_init()* functions are marked as part of the Threads option.

IEEE PASC Interpretation 1003.1c #27 is applied, updating the ERRORS section.

Issue 7

The *pthread_mutexattr_destroy()* and *pthread_mutexattr_init()* functions are moved from the Threads option to the Base.

36411 **NAME**

36412 pthread_mutexattr_getprioceiling, pthread_mutexattr_setprioceiling — get and set the
 36413 prioceiling attribute of the mutex attributes object (**REALTIME THREADS**)

36414 **SYNOPSIS**

```
36415 RPP|TPP #include <pthread.h>
36416
36417 int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *
36418     restrict attr, int *restrict prioceiling);
36419 int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
36420     int prioceiling);
```

36420 **DESCRIPTION**

36421 The *pthread_mutexattr_getprioceiling()* and *pthread_mutexattr_setprioceiling()* functions,
 36422 respectively, shall get and set the priority ceiling attribute of a mutex attributes object pointed to
 36423 by *attr* which was previously created by the function *pthread_mutexattr_init()*.

36424 The *prioceiling* attribute contains the priority ceiling of initialized mutexes. The values of
 36425 *prioceiling* are within the maximum range of priorities defined by SCHED_FIFO.

36426 The *prioceiling* attribute defines the priority ceiling of initialized mutexes, which is the minimum
 36427 priority level at which the critical section guarded by the mutex is executed. In order to avoid
 36428 priority inversion, the priority ceiling of the mutex shall be set to a priority higher than or equal
 36429 to the highest priority of all the threads that may lock that mutex. The values of *prioceiling* are
 36430 within the maximum range of priorities defined under the SCHED_FIFO scheduling policy.

36431 **RETURN VALUE**

36432 Upon successful completion, the *pthread_mutexattr_getprioceiling()* and
 36433 *pthread_mutexattr_setprioceiling()* functions shall return zero; otherwise, an error number shall be
 36434 returned to indicate the error.

36435 **ERRORS**

36436 The *pthread_mutexattr_getprioceiling()* and *pthread_mutexattr_setprioceiling()* functions may fail if:

36437 [EINVAL] The value specified by *attr* or *prioceiling* is invalid.

36438 [EPERM] The caller does not have the privilege to perform the operation.

36439 These functions shall not return an error code of [EINTR].

36440 **EXAMPLES**

36441 None.

36442 **APPLICATION USAGE**

36443 None.

36444 **RATIONALE**

36445 None.

36446 **FUTURE DIRECTIONS**

36447 None.

36448 **SEE ALSO**

36449 *pthread_cond_destroy()*, *pthread_create()*, *pthread_mutex_destroy()*, the Base Definitions volume of
 36450 IEEE Std 1003.1-200x, <pthread.h>

36451
36452
36453
36454
36455
36456
36457
36458
36459
36460
36461
36462
36463
36464
36465
36466

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

Issue 6

The *pthread_mutexattr_getprioceiling()* and *pthread_mutexattr_setprioceiling()* functions are marked as part of the Threads and Thread Priority Protection options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Priority Protection option.

The [ENOTSUP] error condition has been removed since these functions do not have a *protocol* argument.

The **restrict** keyword is added to the *pthread_mutexattr_getprioceiling()* prototype for alignment with the ISO/IEC 9899:1999 standard.

Issue 7

The *pthread_mutexattr_getprioceiling()* and *pthread_mutexattr_setprioceiling()* functions are moved from the Threads option to require support of either the Robust Mutex Priority Protection option or the Non-Robust Mutex Priority Protection option.

36467 **NAME**

36468 pthread_mutexattr_getprotocol, pthread_mutexattr_setprotocol — get and set the protocol
 36469 attribute of the mutex attributes object (**REALTIME THREADS**)

36470 **SYNOPSIS**

```
36471 MC1 #include <pthread.h>
36472
36473 int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *
36474 restrict attr, int *restrict protocol);
36475 int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
36476 int protocol);
```

36476 **DESCRIPTION**

36477 The *pthread_mutexattr_getprotocol()* and *pthread_mutexattr_setprotocol()* functions, respectively,
 36478 shall get and set the protocol attribute of a mutex attributes object pointed to by *attr* which was
 36479 previously created by the function *pthread_mutexattr_init()*.

36480 The *protocol* attribute defines the protocol to be followed in utilizing mutexes. The value of
 36481 *protocol* may be one of:

36482	RPI TPI	PTHREAD_PRIO_INHERIT
36483	MC1	PTHREAD_PRIO_NONE
36484	RPP TPP	PTHREAD_PRIO_PROTECT

36485 which are defined in the **<pthread.h>** header. The default value of the attribute shall be
 36486 PTHREAD_PRIO_NONE.

36487 When a thread owns a mutex with the PTHREAD_PRIO_NONE *protocol* attribute, its priority
 36488 and scheduling shall not be affected by its mutex ownership.

36489 RPI When a thread is blocking higher priority threads because of owning one or more robust
 36490 mutexes with the PTHREAD_PRIO_INHERIT *protocol* attribute, it shall execute at the higher of
 36491 its priority or the priority of the highest priority thread waiting on any of the robust mutexes
 36492 owned by this thread and initialized with this protocol.

36493 TPI When a thread is blocking higher priority threads because of owning one or more non-robust
 36494 mutexes with the PTHREAD_PRIO_INHERIT *protocol* attribute, it shall execute at the higher of
 36495 its priority or the priority of the highest priority thread waiting on any of the non-robust
 36496 mutexes owned by this thread and initialized with this protocol.

36497 RPP When a thread owns one or more robust mutexes initialized with the
 36498 PTHREAD_PRIO_PROTECT protocol, it shall execute at the higher of its priority or the highest
 36499 of the priority ceilings of all the robust mutexes owned by this thread and initialized with this
 36500 attribute, regardless of whether other threads are blocked on any of these robust mutexes or not.

36501 TPP When a thread owns one or more non-robust mutexes initialized with the
 36502 PTHREAD_PRIO_PROTECT protocol, it shall execute at the higher of its priority or the highest
 36503 of the priority ceilings of all the non-robust mutexes owned by this thread and initialized with
 36504 this attribute, regardless of whether other threads are blocked on any of these non-robust
 36505 mutexes or not.

36506 While a thread is holding a mutex which has been initialized with the
 36507 PTHREAD_PRIO_INHERIT or PTHREAD_PRIO_PROTECT protocol attributes, it shall not be
 36508 subject to being moved to the tail of the scheduling queue at its priority in the event that its
 36509 original priority is changed, such as by a call to *sched_setparam()*. Likewise, when a thread

pthread_mutexattr_getprotocol()

System Interfaces

36510 unlocks a mutex that has been initialized with the PTHREAD_PRIO_INHERIT or
 36511 PTHREAD_PRIO_PROTECT protocol attributes, it shall not be subject to being moved to the tail
 36512 of the scheduling queue at its priority in the event that its original priority is changed.

36513 If a thread simultaneously owns several mutexes initialized with different protocols, it shall
 36514 execute at the highest of the priorities that it would have obtained by each of these protocols.

36515 RPI | TPI When a thread makes a call to *pthread_mutex_lock()*, the mutex was initialized with the protocol
 36516 attribute having the value PTHREAD_PRIO_INHERIT, when the calling thread is blocked
 36517 because the mutex is owned by another thread, that owner thread shall inherit the priority level
 36518 of the calling thread as long as it continues to own the mutex. The implementation shall update
 36519 its execution priority to the maximum of its assigned priority and all its inherited priorities.
 36520 Furthermore, if this owner thread itself becomes blocked on another mutex, the same priority
 36521 inheritance effect shall be propagated to this other owner thread, in a recursive manner.

RETURN VALUE

36522 Upon successful completion, the *pthread_mutexattr_getprotocol()* and
 36523 *pthread_mutexattr_setprotocol()* functions shall return zero; otherwise, an error number shall be
 36524 returned to indicate the error.
 36525

ERRORS

36526 The *pthread_mutexattr_setprotocol()* function shall fail if:

36527 [ENOTSUP] The value specified by *protocol* is an unsupported value.

36528 The *pthread_mutexattr_getprotocol()* and *pthread_mutexattr_setprotocol()* functions may fail if:

36529 [EINVAL] The value specified by *attr* or *protocol* is invalid.

36530 [EPERM] The caller does not have the privilege to perform the operation.

36531 These functions shall not return an error code of [EINTR].
 36532

EXAMPLES

36533 None.
 36534

APPLICATION USAGE

36535 None.
 36536

RATIONALE

36537 None.
 36538

FUTURE DIRECTIONS

36539 None.
 36540

SEE ALSO

36541 *pthread_cond_destroy()*, *pthread_create()*, *pthread_mutex_destroy()*, the Base Definitions volume of
 36542 IEEE Std 1003.1-200x, <pthread.h>
 36543

CHANGE HISTORY

36544 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

36545 Marked as part of the Realtime Threads Feature Group.
 36546

Issue 6

36547 The *pthread_mutexattr_getprotocol()* and *pthread_mutexattr_setprotocol()* functions are marked as
 36548 part of the Threads option and either the Thread Priority Protection or Thread Priority
 36549 Inheritance options.
 36550

36551 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 36552 implementation does not support the Thread Priority Protection or Thread Priority Inheritance
 36553 options.

36554 The **restrict** keyword is added to the *pthread_mutexattr_getprotocol()* prototype for alignment

36555

with the ISO/IEC 9899:1999 standard.

36556

Issue 7

36557

36558

36559

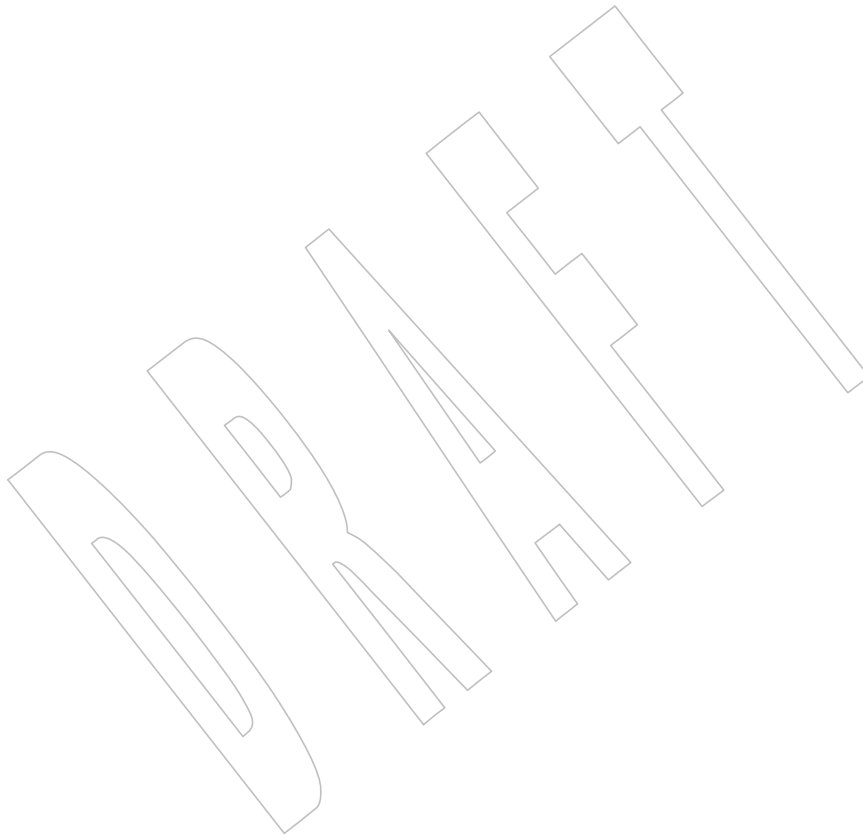
36560

The *pthread_mutexattr_getprotocol()* and *pthread_mutexattr_setprotocol()* functions are moved from the Threads option to require support of either the Non-Robust Mutex Priority Protection option or the Non-Robust Mutex Priority Inheritance option or the Robust Mutex Priority Protection option or the Robust Mutex Priority Inheritance option.

36561

36562

SD5-XSH-ERN-135 is applied, updating the DESCRIPTION to define a default value for the *protocol* attribute.



36563 **NAME**

36564 pthread_mutexattr_getpshared, pthread_mutexattr_setpshared — get and set the process-shared
 36565 attribute

36566 **SYNOPSIS**

```
36567 TSH #include <pthread.h>
36568
36568 int pthread_mutexattr_getpshared(const pthread_mutexattr_t *
36569 restrict attr, int *restrict pshared);
36570 int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
36571 int pshared);
```

36572 **DESCRIPTION**

36573 The *pthread_mutexattr_getpshared()* function shall obtain the value of the *process-shared* attribute
 36574 from the attributes object referenced by *attr*. The *pthread_mutexattr_setpshared()* function shall
 36575 set the *process-shared* attribute in an initialized attributes object referenced by *attr*.

36576 The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a mutex to be
 36577 operated upon by any thread that has access to the memory where the mutex is allocated, even if
 36578 the mutex is allocated in memory that is shared by multiple processes. If the *process-shared*
 36579 attribute is PTHREAD_PROCESS_PRIVATE, the mutex shall only be operated upon by threads
 36580 created within the same process as the thread that initialized the mutex; if threads of differing
 36581 processes attempt to operate on such a mutex, the behavior is undefined. The default value of
 36582 the attribute shall be PTHREAD_PROCESS_PRIVATE.

36583 **RETURN VALUE**

36584 Upon successful completion, *pthread_mutexattr_setpshared()* shall return zero; otherwise, an error
 36585 number shall be returned to indicate the error.

36586 Upon successful completion, *pthread_mutexattr_getpshared()* shall return zero and store the value
 36587 of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter.
 36588 Otherwise, an error number shall be returned to indicate the error.

36589 **ERRORS**

36590 The *pthread_mutexattr_getpshared()* and *pthread_mutexattr_setpshared()* functions may fail if:

36591 [EINVAL] The value specified by *attr* is invalid.

36592 The *pthread_mutexattr_setpshared()* function may fail if:

36593 [EINVAL] The new value specified for the attribute is outside the range of legal values
 36594 for that attribute.

36595 These functions shall not return an error code of [EINTR].

36596 **EXAMPLES**

36597 None.

36598 **APPLICATION USAGE**

36599 None.

36600 **RATIONALE**

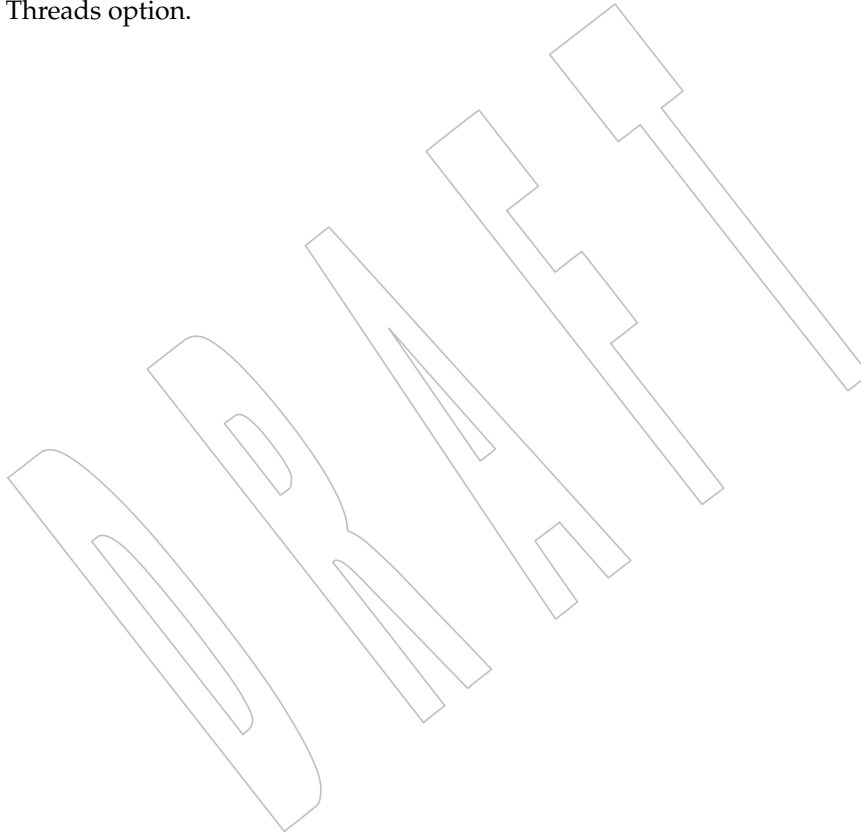
36601 None.

36602 **FUTURE DIRECTIONS**

36603 None.

36604 **SEE ALSO**36605 *pthread_cond_destroy()*, *pthread_create()*, *pthread_mutex_destroy()*, *pthread_mutexattr_destroy()*, the
36606 Base Definitions volume of IEEE Std 1003.1-200x, <pthread.h>36607 **CHANGE HISTORY**

36608 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

36609 **Issue 6**36610 The *pthread_mutexattr_getpshared()* and *pthread_mutexattr_setpshared()* functions are marked as
36611 part of the Threads and Thread Process-Shared Synchronization options.36612 The **restrict** keyword is added to the *pthread_mutexattr_getpshared()* prototype for alignment
36613 with the ISO/IEC 9899:1999 standard.36614 **Issue 7**36615 The *pthread_mutexattr_getpshared()* and *pthread_mutexattr_setpshared()* functions are moved from
36616 the Threads option.

36617 **NAME**

36618 pthread_mutexattr_getrobust, pthread_mutexattr_setrobust — get and set the mutex robust
 36619 attribute

36620 **SYNOPSIS**

```
36621 #include <pthread.h>
36622
36622 int pthread_mutexattr_getrobust(const pthread_mutexattr_t *restrict
36623     attr, int *restrict robust);
36624 int pthread_mutexattr_setrobust(pthread_mutexattr_t *attr,
36625     int robust);
```

36626 **DESCRIPTION**

36627 The *pthread_mutexattr_getrobust()* and *pthread_mutexattr_setrobust()* functions, respectively, shall
 36628 get and set the mutex *robust* attribute. This attribute is set in the *robust* parameter. Valid values
 36629 for *robust* include:

36630 **PTHREAD_MUTEX_STALLED**

36631 No special actions are taken if the owner of the mutex is terminated while holding the
 36632 mutex lock. This can lead to deadlocks if no other thread can unlock the mutex.
 36633 This is the default value.

36634 **PTHREAD_MUTEX_ROBUST**

36635 If the process containing the owning thread of a robust mutex terminates while holding the
 36636 mutex lock, the next thread that acquires the mutex shall be notified about the termination
 36637 by the return value [EOWNERDEAD] from the locking function. If the owning thread of a
 36638 robust mutex terminates while holding the mutex lock, the next thread that acquires the
 36639 mutex may be notified about the termination by the return value [EOWNERDEAD]. The
 36640 notified thread can then attempt to mark the state protected by the mutex as consistent
 36641 again by a call to *pthread_mutex_consistent()*. After a subsequent successful call to
 36642 *pthread_mutex_unlock()*, the mutex lock shall be released and can be used normally by other
 36643 threads. If the mutex is unlocked without a call to *pthread_mutex_consistent()*, it shall be in a
 36644 permanently unusable state and all attempts to lock the mutex shall fail with the error
 36645 [ENOTRECOVERABLE]. The only permissible operation on such a mutex is
 36646 *pthread_mutex_destroy()*.

36647 **RETURN VALUE**

36648 Upon successful completion, the *pthread_mutexattr_getrobust()* function shall return zero and
 36649 store the value of the *robust* attribute of *attr* into the object referenced by the *robust* parameter.
 36650 Otherwise, an error value shall be returned to indicate the error. If successful, the
 36651 *pthread_mutexattr_setrobust()* function shall return zero; otherwise, an error number shall be
 36652 returned to indicate the error.

36653 **ERRORS**

36654 The *pthread_mutexattr_setrobust()* function shall fail if:

36655 [EINVAL] The value of *robust* is invalid.

36656 The *pthread_mutexattr_getrobust()* and *pthread_mutexattr_setrobust()* functions may fail if:

36657 [EINVAL] The value specified by *attr* is invalid.

36658 These functions shall not return an error code of [EINTR].

EXAMPLES

36659
36660 None.

APPLICATION USAGE

36661
36662 The actions required to make the state protected by the mutex consistent again are solely
36663 dependent on the application. If it is not possible to make the state of a mutex consistent, robust
36664 mutexes can be used to notify this situation by calling *pthread_mutex_unlock()* without a prior
36665 call to *pthread_mutex_consistent()*.

36666 If the state is declared inconsistent by calling *pthread_mutex_unlock()* without a prior call to
36667 *pthread_mutex_consistent()*, a possible approach could be to destroy the mutex and then
36668 reinitialize it. However, it should be noted that this is possible only in certain situations where
36669 the state protected by the mutex has to be reinitialized and coordination achieved with other
36670 threads blocked on the mutex, because otherwise a call to a locking function with a reference to a
36671 mutex object invalidated by a call to *pthread_mutex_destroy()* results in undefined behavior.

RATIONALE

36672
36673 None.

FUTURE DIRECTIONS

36674
36675 None.

SEE ALSO

36676
36677 *pthread_mutex_consistent()*, *pthread_mutex_destroy()*, *pthread_mutex_lock()*, the Base Definitions
36678 volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

36679
36680 First released in Issue 7.

36681 The *pthread_mutexattr_getrobust()* and *pthread_mutexattr_setrobust()* functions are moved from
36682 the Threads option to the Base.

NAME

pthread_mutexattr_gettype, pthread_mutexattr_settype — get and set the mutex type attribute

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
    int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

DESCRIPTION

The *pthread_mutexattr_gettype()* and *pthread_mutexattr_settype()* functions, respectively, shall get and set the mutex *type* attribute. This attribute is set in the *type* parameter to these functions. The default value of the *type* attribute is PTHREAD_MUTEX_DEFAULT.

The type of mutex is contained in the *type* attribute of the mutex attributes. Valid mutex types include:

PTHREAD_MUTEX_NORMAL

This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it shall deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behavior. Attempting to unlock an unlocked mutex results in undefined behavior.

PTHREAD_MUTEX_ERRORCHECK

This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking it shall return with an error. A thread attempting to unlock a mutex which another thread has locked shall return with an error. A thread attempting to unlock an unlocked mutex shall return with an error.

PTHREAD_MUTEX_RECURSIVE

A thr

ERRORS

36726

The *pthread_mutexattr_settype()* function shall fail if:

36727

[EINVAL] The value *type* is invalid.

36728

36729

The *pthread_mutexattr_gettype()* and *pthread_mutexattr_settype()* functions may fail if:

36730

[EINVAL] The value specified by *attr* is invalid.

36731

These functions shall not return an error code of [EINTR].

EXAMPLES

36732

None.

36733

APPLICATION USAGE

36734

It is advised that an application should not use a PTHREAD_MUTEX_RECURSIVE mutex with condition variables because the implicit unlock performed for a *pthread_cond_timedwait()* or *pthread_cond_wait()* may not actually release the mutex (if it had been locked multiple times). If this happens, no other thread can satisfy the condition of the predicate.

36735

36736

36737

36738

RATIONALE

36739

None.

36740

FUTURE DIRECTIONS

36741

None.

36742

SEE ALSO

36743

pthread_cond_timedwait(), the Base Definitions volume of IEEE Std 1003.1-200x, <pthread.h>

36744

CHANGE HISTORY

36745

First released in Issue 5.

36746

Issue 6

36747

The Open Group Corrigendum U033/3 is applied. The SYNOPSIS for *pthread_mutexattr_gettype()* is updated so that the first argument is of type **const pthread_mutexattr_t***.

36748

36749

36750

The **restrict** keyword is added to the *pthread_mutexattr_gettype()* prototype for alignment with the ISO/IEC 9899:1999 standard.

36751

36752

Issue 7

36753

The *pthread_mutexattr_gettype()* and *pthread_mutexattr_settype()* functions are moved from the XSI option to the Base.

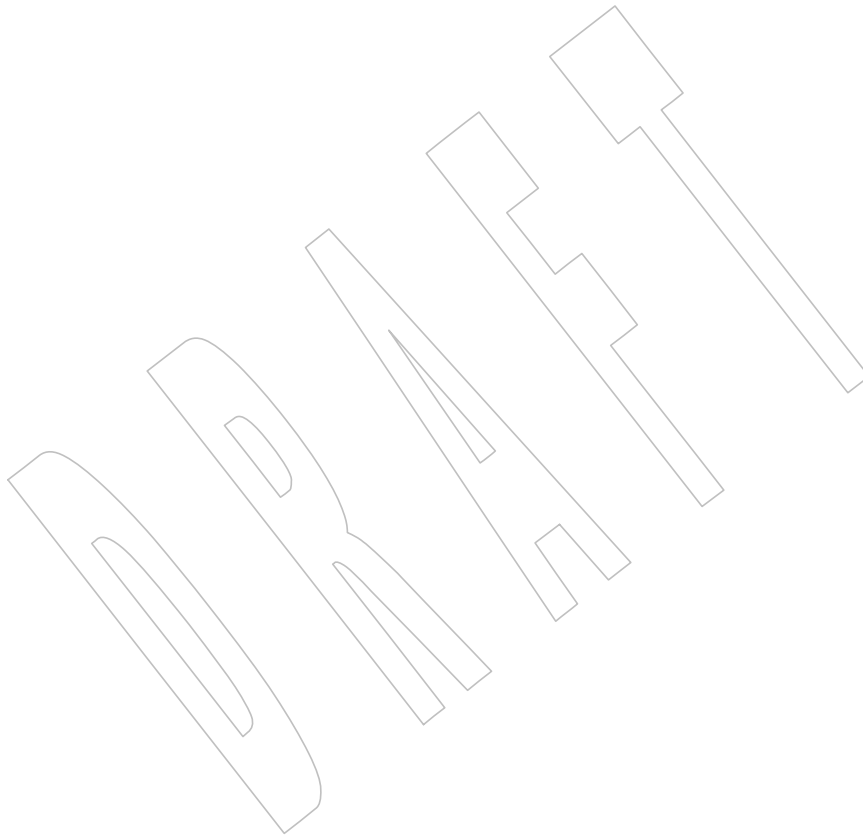
36754

36755

36756 **NAME**
36757 pthread_mutexattr_init — initialize the mutex attributes object

36758 **SYNOPSIS**
36759 #include <pthread.h>
36760 int pthread_mutexattr_init(pthread_mutexattr_t *attr);

36761 **DESCRIPTION**
36762 Refer to *pthread_mutexattr_destroy()*.



36763 **NAME**

36764 pthread_mutexattr_setprioceiling — set the prioceiling attribute of the mutex attributes object
36765 (**REALTIME THREADS**)

36766 **SYNOPSIS**

```
36767 RPP|TPP #include <pthread.h>  
36768 int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,  
36769 int prioceiling);
```

36770 **DESCRIPTION**

36771 Refer to [pthread_mutexattr_getprioceiling\(\)](#).

pthread_mutexattr_setprotocol()*System Interfaces*36772 **NAME**

36773 pthread_mutexattr_setprotocol — set the protocol attribute of the mutex attributes object
36774 (**REALTIME THREADS**)

36775 **SYNOPSIS**

```
36776 MC1 #include <pthread.h>  
36777 int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,  
36778 int protocol);
```

36779 **DESCRIPTION**

36780 Refer to [pthread_mutexattr_getprotocol\(\)](#).

36781 **NAME**
36782 pthread_mutexattr_setpshared — set the process-shared attribute

36783 **SYNOPSIS**

```
36784 TSH #include <pthread.h>  
36785 int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,  
36786 int pshared);
```

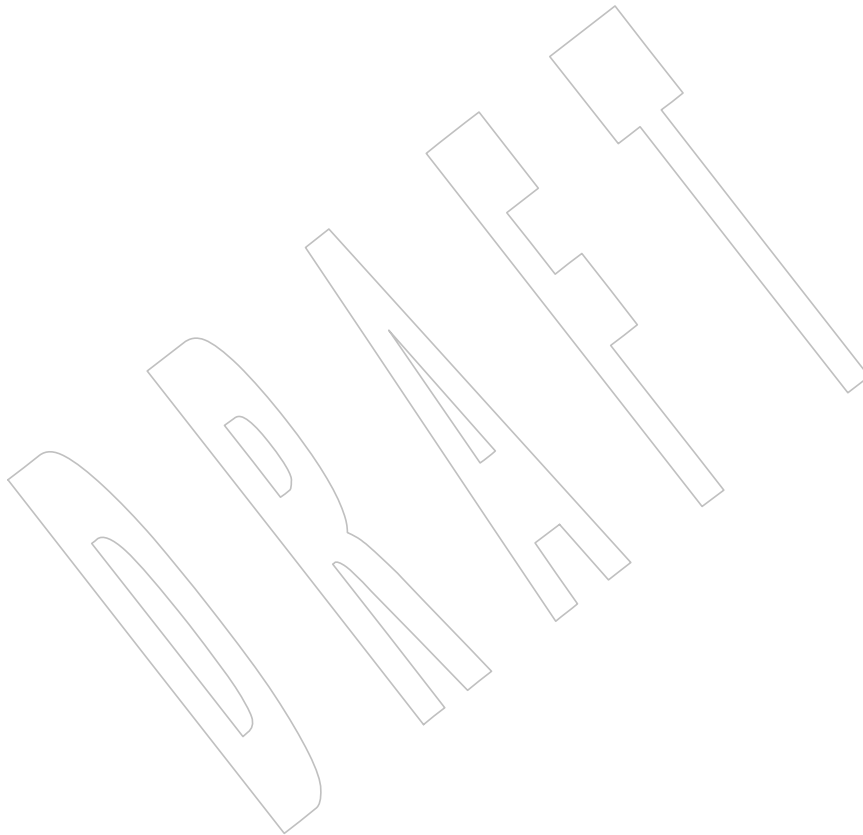
36787 **DESCRIPTION**

36788 Refer to *pthread_mutexattr_getpshared()*.

36789 **NAME**
36790 pthread_mutexattr_setrobust — get and set the mutex robust attribute

36791 **SYNOPSIS**
36792 #include <pthread.h>
36793 int pthread_mutexattr_setrobust(pthread_mutexattr_t *attr,
36794 int robust);

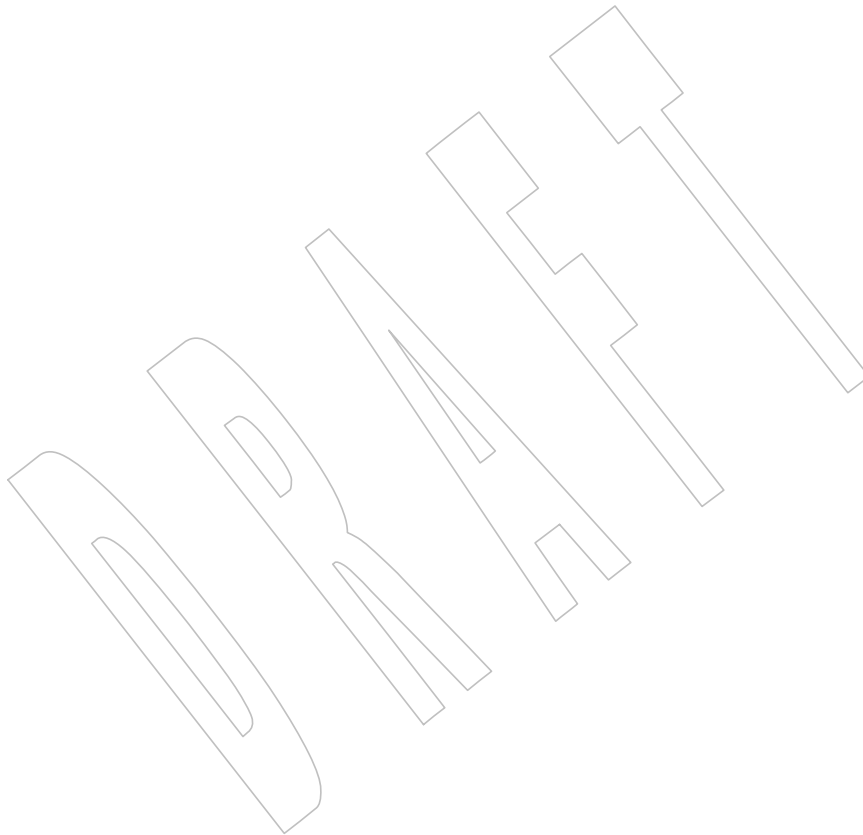
36795 **DESCRIPTION**
36796 Refer to [pthread_mutexattr_getrobust\(\)](#).



36797 **NAME**
36798 pthread_mutexattr_settype — set the mutex type attribute

36799 **SYNOPSIS**
36800 #include <pthread.h>
36801 int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);

36802 **DESCRIPTION**
36803 Refer to *pthread_mutexattr_gettype()*.



36804 **NAME**
 36805 pthread_once — dynamic package initialization

36806 **SYNOPSIS**
 36807 #include <pthread.h>
 36808 int pthread_once(pthread_once_t *once_control,
 36809 void (*init_routine)(void));
 36810 pthread_once_t once_control = PTHREAD_ONCE_INIT;

36811 **DESCRIPTION**
 36812 The first call to *pthread_once()* by any thread in a process, with a given *once_control*, shall call the
 36813 *init_routine* with no arguments. Subsequent calls of *pthread_once()* with the same *once_control*
 36814 shall not call the *init_routine*. On return from *pthread_once()*, *init_routine* shall have completed.
 36815 The *once_control* parameter shall determine whether the associated initialization routine has been
 36816 called.

36817 The *pthread_once()* function is not a cancellation point. However, if *init_routine* is a cancellation
 36818 point and is canceled, the effect on *once_control* shall be as if *pthread_once()* was never called.

36819 The constant PTHREAD_ONCE_INIT is defined in the <pthread.h> header.

36820 The behavior of *pthread_once()* is undefined if *once_control* has automatic storage duration or is
 36821 not initialized by PTHREAD_ONCE_INIT.

36822 **RETURN VALUE**
 36823 Upon successful completion, *pthread_once()* shall return zero; otherwise, an error number shall
 36824 be returned to indicate the error.

36825 **ERRORS**
 36826 The *pthread_once()* function may fail if:
 36827 [EINVAL] If either *once_control* or *init_routine* is invalid.
 36828 The *pthread_once()* function shall not return an error code of [EINTR].

36829 **EXAMPLES**
 36830 None.

36831 **APPLICATION USAGE**
 36832 None.

36833 **RATIONALE**
 36834 Some C libraries are designed for dynamic initialization. That is, the global initialization for the
 36835 library is performed when the first procedure in the library is called. In a single-threaded
 36836 program, this is normally implemented using a static variable whose value is checked on entry
 36837 to a routine, as follows:

```
36838 static int random_is_initialized = 0;
36839 extern int initialize_random();
36840
36841 int random_function()
36842 {
36843     if (random_is_initialized == 0) {
36844         initialize_random();
36845         random_is_initialized = 1;
36846     }
36847     ... /* Operations performed after initialization. */
36848 }
```

36848 To keep the same structure in a multi-threaded program, a new primitive is needed. Otherwise,
 36849 library initialization has to be accomplished by an explicit call to a library-exported initialization
 36850 function prior to any use of the library.

36851 For dynamic library initialization in a multi-threaded process, a simple initialization flag is not
 36852 sufficient; the flag needs to be protected against modification by multiple threads
 36853 simultaneously calling into the library. Protecting the flag requires the use of a mutex; however,
 36854 mutexes have to be initialized before they are used. Ensuring that the mutex is only initialized
 36855 once requires a recursive solution to this problem.

36856 The use of *pthread_once()* not only supplies an implementation-guaranteed means of dynamic
 36857 initialization, it provides an aid to the reliable construction of multi-threaded and realtime
 36858 systems. The preceding example then becomes:

```
36859 #include <pthread.h>
36860 static pthread_once_t random_is_initialized = PTHREAD_ONCE_INIT;
36861 extern int initialize_random();
36862
36863 int random_function()
36864 {
36865     (void) pthread_once(&random_is_initialized, initialize_random);
36866     ... /* Operations performed after initialization. */
36867 }
```

36867 Note that a **pthread_once_t** cannot be an array because some compilers do not accept the
 36868 construct **&<array_name>**.

36869 FUTURE DIRECTIONS

36870 None.

36871 SEE ALSO

36872 The Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

36873 CHANGE HISTORY

36874 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

36875 Issue 6

36876 The *pthread_once()* function is marked as part of the Threads option.

36877 The [EINVAL] error is added as a may fail case for if either argument is invalid.

36878 Issue 7

36879 The *pthread_once()* function is moved from the Threads option to the Base.

36880 **NAME**

36881 pthread_rwlock_destroy, pthread_rwlock_init — destroy and initialize a read-write lock object

36882 **SYNOPSIS**

36883 #include <pthread.h>

```
36884 int pthread_rwlock_destroy(pthread_rwlock_t *rwlck,
36885 int pthread_rwlock_init(pthread_rwlock_t *restrict rwlck,
36886 const pthread_rwlockattr_t *restrict attr);
```

36887 XSI pthread_rwlock_t rwlck = PTHREAD_RWLOCK_INITIALIZER;

36888 **DESCRIPTION**

36889 The *pthread_rwlock_destroy()* function shall destroy the read-write lock object referenced by
 36890 *rwlck* and release any resources used by the lock. The effect of subsequent use of the lock is
 36891 undefined until the lock is reinitialized by another call to *pthread_rwlock_init()*. An
 36892 implementation may cause *pthread_rwlock_destroy()* to set the object referenced by *rwlck* to an
 36893 invalid value. Results are undefined if *pthread_rwlock_destroy()* is called when any thread holds
 36894 *rwlck*. Attempting to destroy an uninitialized read-write lock results in undefined behavior.

36895 The *pthread_rwlock_init()* function shall allocate any resources required to use the read-write lock
 36896 referenced by *rwlck* and initializes the lock to an unlocked state with attributes referenced by
 36897 *attr*. If *attr* is NULL, the default read-write lock attributes shall be used; the effect is the same as
 36898 passing the address of a default read-write lock attributes object. Once initialized, the lock can be
 36899 used any number of times without being reinitialized. Results are undefined if
 36900 *pthread_rwlock_init()* is called specifying an already initialized read-write lock. Results are
 36901 undefined if a read-write lock is used without first being initialized.

36902 If the *pthread_rwlock_init()* function fails, *rwlck* shall not be initialized and the contents of *rwlck*
 36903 are undefined.

36904 Only the object referenced by *rwlck* may be used for performing synchronization. The result of
 36905 referring to copies of that object in calls to *pthread_rwlock_destroy()*, *pthread_rwlock_rdlock()*,
 36906 *pthread_rwlock_timedrdlock()*, *pthread_rwlock_timedwrlock()*, *pthread_rwlock_tryrdlock()*,
 36907 *pthread_rwlock_trywrlock()*, *pthread_rwlock_unlock()*, or *pthread_rwlock_wrlock()* is undefined.

36908 XSI In cases where default read-write lock attributes are appropriate, the macro
 36909 PTHREAD_RWLOCK_INITIALIZER can be used to initialize read-write locks that are statically
 36910 allocated. The effect shall be equivalent to dynamic initialization by a call to *pthread_rwlock_init()*
 36911 with the *attr* parameter specified as NULL, except that no error checks are performed.

36912 **RETURN VALUE**

36913 If successful, the *pthread_rwlock_destroy()* and *pthread_rwlock_init()* functions shall return zero;
 36914 otherwise, an error number shall be returned to indicate the error.

36915 The [EBUSY] and [EINVAL] error checks, if implemented, act as if they were performed
 36916 immediately at the beginning of processing for the function and caused an error return prior to
 36917 modifying the state of the read-write lock specified by *rwlck*.

36918 **ERRORS**

36919 The *pthread_rwlock_destroy()* function may fail if:

36920 [EBUSY] The implementation has detected an attempt to destroy the object referenced
 36921 by *rwlck* while it is locked.

36922 [EINVAL] The value specified by *rwlck* is invalid.

36923 The *pthread_rwlock_init()* function shall fail if:

- 36924 [EAGAIN] The system lacked the necessary resources (other than memory) to initialize
36925 another read-write lock.
- 36926 [ENOMEM] Insufficient memory exists to initialize the read-write lock.
- 36927 [EPERM] The caller does not have the privilege to perform the operation.
- 36928 The *pthread_rwlock_init()* function may fail if:
- 36929 [EBUSY] The implementation has detected an attempt to reinitialize the object
36930 referenced by *rwlock*, a previously initialized but not yet destroyed read-write
36931 lock.
- 36932 [EINVAL] The value specified by *attr* is invalid.
- 36933 These functions shall not return an error code of [EINTR].

EXAMPLES

36934 None.
36935

APPLICATION USAGE

36936 Applications using these and related read-write lock functions may be subject to priority
36937 inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285,
36938 Priority Inversion.
36939

RATIONALE

36940 None.
36941

FUTURE DIRECTIONS

36942 None.
36943

SEE ALSO

36944 *pthread_rwlock_rdlock()*, *pthread_rwlock_timedrdlock()*, *pthread_rwlock_timedwrlock()*,
36945 *pthread_rwlock_tryrdlock()*, *pthread_rwlock_trywrlock()*, *pthread_rwlock_unlock()*,
36946 *pthread_rwlock_wrlock()*, the Base Definitions volume of IEEE Std 1003.1-200x, <pthread.h>
36947

CHANGE HISTORY

36948 First released in Issue 5.
36949

Issue 6

36950 The following changes are made for alignment with IEEE Std 1003.1j-2000:
36951

- 36952 • The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is
36953 now part of the Threads option (previously it was part of the Read-Write Locks option in
36954 IEEE Std 1003.1j-2000 and also part of the XSI extension). The initializer macro is also
36955 deleted from the SYNOPSIS.
- 36956 • The DESCRIPTION is updated as follows:
 - 36957 — It explicitly notes allocation of resources upon initialization of a read-write lock
36958 object.
 - 36959 — A paragraph is added specifying that copies of read-write lock objects may not be
36960 used.
- 36961 • An [EINVAL] error is added to the ERRORS section for *pthread_rwlock_init()*, indicating
36962 that the *rwlock* value is invalid.
- 36963 • The SEE ALSO section is updated.

36964 The **restrict** keyword is added to the *pthread_rwlock_init()* prototype for alignment with the
36965 ISO/IEC 9899:1999 standard.

36966 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/45 is applied, adding APPLICATION
36967 USAGE relating to priority inversion.

pthread_rwlock_destroy()*System Interfaces*

36968

Issue 7

36969

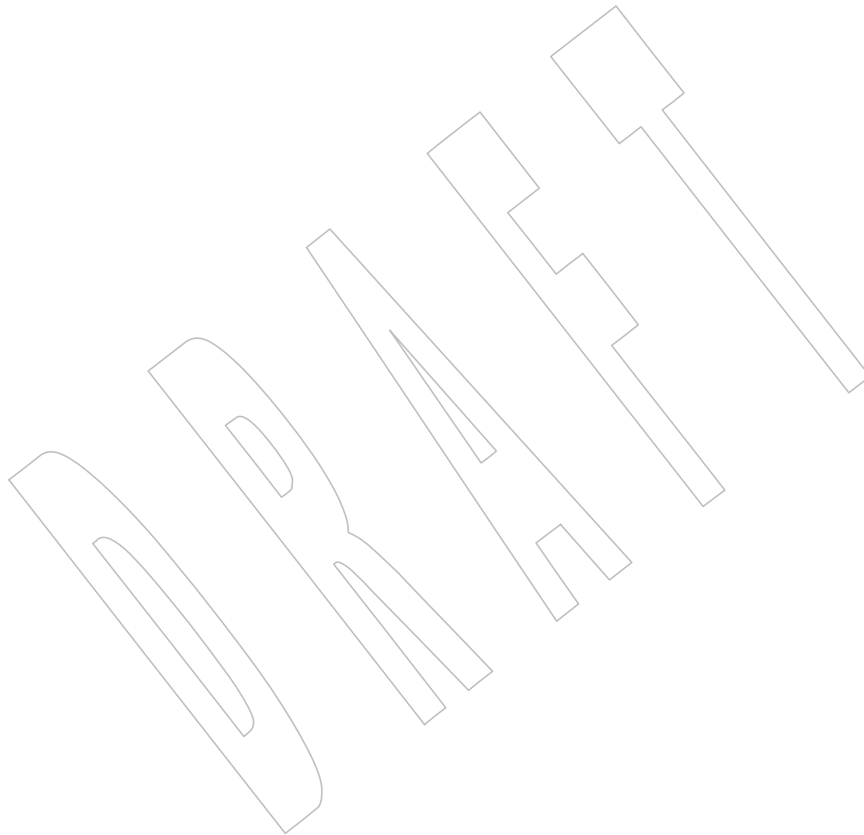
Austin Group Interpretation 1003.1-2001 #048 is applied, adding the PTHREAD_RWLOCK_INITIALIZER macro.

36970

36971

The *pthread_rwlock_destroy()* and *pthread_rwlock_init()* functions are moved from the Threads option to the Base.

36972



36973 **NAME**
 36974 `pthread_rwlock_rdlock`, `pthread_rwlock_tryrdlock` — lock a read-write lock object for reading

36975 **SYNOPSIS**

36976 `#include <pthread.h>`
 36977 `int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);`
 36978 `int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);`

36979 **DESCRIPTION**

36980 The `pthread_rwlock_rdlock()` function shall apply a read lock to the read-write lock referenced by
 36981 `rwlock`. The calling thread acquires the read lock if a writer does not hold the lock and there are
 36982 no writers blocked on the lock.

36983 TPS If the Thread Execution Scheduling option is supported, and the threads involved in the lock are
 36984 executing with the scheduling policies `SCHED_FIFO` or `SCHED_RR`, the calling thread shall not
 36985 acquire the lock if a writer holds the lock or if writers of higher or equal priority are blocked on
 36986 the lock; otherwise, the calling thread shall acquire the lock.

36987 TPS TSP If the Thread Execution Scheduling option is supported, and the threads involved in the lock are
 36988 executing with the `SCHED_SPORADIC` scheduling policy, the calling thread shall not acquire
 36989 the lock if a writer holds the lock or if writers of higher or equal priority are blocked on the lock;
 36990 otherwise, the calling thread shall acquire the lock.

36991 If the Thread Execution Scheduling option is not supported, it is implementation-defined
 36992 whether the calling thread acquires the lock when a writer does not hold the lock and there are
 36993 writers blocked on the lock. If a writer holds the lock, the calling thread shall not acquire the
 36994 read lock. If the read lock is not acquired, the calling thread shall block until it can acquire the
 36995 lock. The calling thread may deadlock if at the time the call is made it holds a write lock.

36996 A thread may hold multiple concurrent read locks on `rwlock` (that is, successfully call the
 36997 `pthread_rwlock_rdlock()` function n times). If so, the application shall ensure that the thread
 36998 performs matching unlocks (that is, it calls the `pthread_rwlock_unlock()` function n times).

36999 The maximum number of simultaneous read locks that an implementation guarantees can be
 37000 applied to a read-write lock shall be implementation-defined. The `pthread_rwlock_rdlock()`
 37001 function may fail if this maximum would be exceeded.

37002 The `pthread_rwlock_tryrdlock()` function shall apply a read lock as in the `pthread_rwlock_rdlock()`
 37003 function, with the exception that the function shall fail if the equivalent `pthread_rwlock_rdlock()`
 37004 call would have blocked the calling thread. In no case shall the `pthread_rwlock_tryrdlock()`
 37005 function ever block; it always either acquires the lock or fails and returns immediately.

37006 Results are undefined if any of these functions are called with an uninitialized read-write lock.

37007 If a signal is delivered to a thread waiting for a read-write lock for reading, upon return from the
 37008 signal handler the thread resumes waiting for the read-write lock for reading as if it was not
 37009 interrupted.

37010 **RETURN VALUE**

37011 If successful, the `pthread_rwlock_rdlock()` function shall return zero; otherwise, an error number
 37012 shall be returned to indicate the error.

37013 The `pthread_rwlock_tryrdlock()` function shall return zero if the lock for reading on the read-write
 37014 lock object referenced by `rwlock` is acquired. Otherwise, an error number shall be returned to
 37015 indicate the error.

ERRORS

37016

37017

The *pthread_rwlock_tryrdlock()* function shall fail if:

37018

[EBUSY] The read-write lock could not be acquired for reading because a writer holds the lock or a writer with the appropriate priority was blocked on it.

37019

37020

The *pthread_rwlock_rdlock()* and *pthread_rwlock_tryrdlock()* functions may fail if:

37021

[EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock object.

37022

37023

[EAGAIN] The read lock could not be acquired because the maximum number of read locks for *rwlock* has been exceeded.

37024

37025

The *pthread_rwlock_rdlock()* function may fail if:

37026

[EDEADLK] A deadlock condition was detected or the current thread already owns the read-write lock for writing.

37027

37028

These functions shall not return an error code of [EINTR].

EXAMPLES

37029

None.

37030

APPLICATION USAGE

37031

Applications using these functions may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

37032

37033

RATIONALE

37034

None.

37035

FUTURE DIRECTIONS

37036

None.

37037

SEE ALSO

37038

pthread_rwlock_destroy(), *pthread_rwlock_timedrdlock()*, *pthread_rwlock_timedwrlock()*, *pthread_rwlock_trywrlock()*, *pthread_rwlock_unlock()*, *pthread_rwlock_wrlock()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, <pthread.h>

37039

37040

37041

CHANGE HISTORY

37042

First released in Issue 5.

37043

Issue 6

37044

The following changes are made for alignment with IEEE Std 1003.1j-2000:

37045

37046

- The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is now part of the Threads option (previously it was part of the Read-Write Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension).

37047

37048

- The DESCRIPTION is updated as follows:

37049

- Conditions under which writers have precedence over readers are specified.

37050

- Failure of *pthread_rwlock_tryrdlock()* is clarified.

37051

- A paragraph on the maximum number of read locks is added.

37052

- In the ERRORS sections, [EBUSY] is modified to take into account write priority, and [EDEADLK] is deleted as a *pthread_rwlock_tryrdlock()* error.

37053

37054

- The SEE ALSO section is updated.

37055

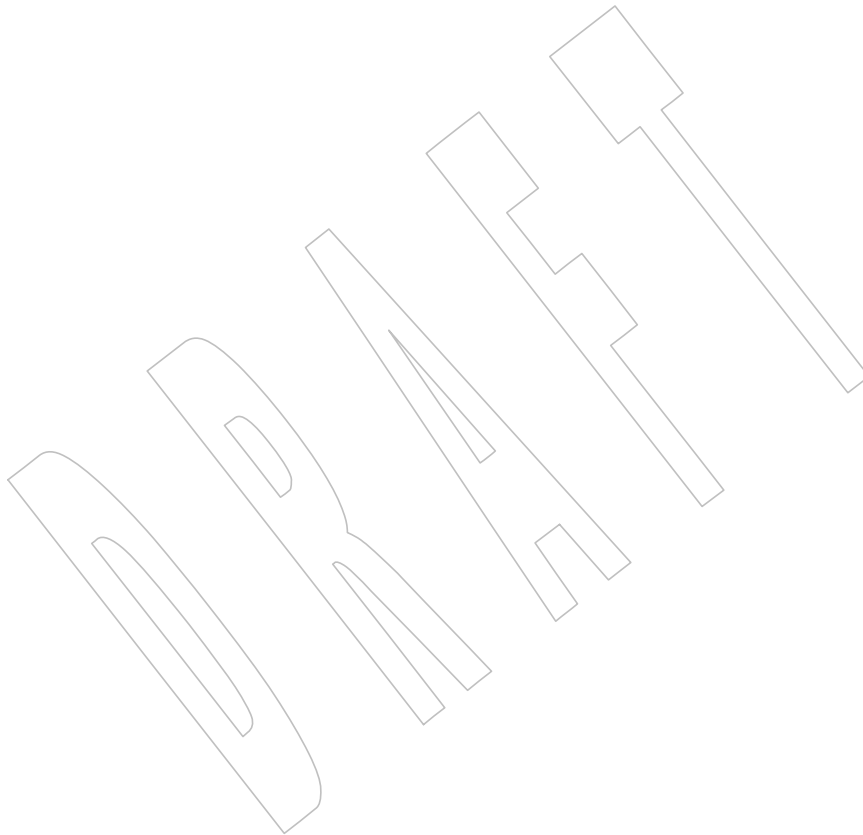
IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/101 is applied, updating the ERRORS section so that the [EDEADLK] error includes detection of a deadlock condition.

37056

37057

37058
37059
37060**Issue 7**

The *pthread_rwlock_rdlock()* and *pthread_rwlock_tryrdlock()* functions are moved from the Threads option to the Base.



37061 **NAME**
 37062 pthread_rwlock_timedrdlock — lock a read-write lock for reading

37063 **SYNOPSIS**

```
37064 #include <pthread.h>
37065 #include <time.h>
37066
37066 int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rlock,
37067                               const struct timespec *restrict abs_timeout);
```

37068 **DESCRIPTION**

37069 The *pthread_rwlock_timedrdlock()* function shall apply a read lock to the read-write lock
 37070 referenced by *rlock* as in the *pthread_rwlock_rdlock()* function. However, if the lock cannot be
 37071 acquired without waiting for other threads to unlock the lock, this wait shall be terminated
 37072 when the specified timeout expires. The timeout shall expire when the absolute time specified
 37073 by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the
 37074 value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout*
 37075 has already been passed at the time of the call.

37076 The timeout shall be based on the CLOCK_REALTIME clock. The resolution of the timeout shall
 37077 be the resolution of the CLOCK_REALTIME clock. The **timespec** data type is defined in the
 37078 **<time.h>** header. Under no circumstances shall the function fail with a timeout if the lock can be
 37079 acquired immediately. The validity of the *abs_timeout* parameter need not be checked if the lock
 37080 can be immediately acquired.

37081 If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-
 37082 write lock via a call to *pthread_rwlock_timedrdlock()*, upon return from the signal handler the
 37083 thread shall resume waiting for the lock as if it was not interrupted.

37084 The calling thread may deadlock if at the time the call is made it holds a write lock on *rlock*.
 37085 The results are undefined if this function is called with an uninitialized read-write lock.

37086 **RETURN VALUE**

37087 The *pthread_rwlock_timedrdlock()* function shall return zero if the lock for reading on the read-
 37088 write lock object referenced by *rlock* is acquired. Otherwise, an error number shall be returned
 37089 to indicate the error.

37090 **ERRORS**

37091 The *pthread_rwlock_timedrdlock()* function shall fail if:

37092 [ETIMEDOUT] The lock could not be acquired before the specified timeout expired.

37093 The *pthread_rwlock_timedrdlock()* function may fail if:

37094 [EAGAIN] The read lock could not be acquired because the maximum number of read
 37095 locks for lock would be exceeded.

37096 [EDEADLK] A deadlock condition was detected or the calling thread already holds a write
 37097 lock on *rlock*.

37098 [EINVAL] The value specified by *rlock* does not refer to an initialized read-write lock
 37099 object, or the *abs_timeout* nanosecond value is less than zero or greater than or
 37100 equal to 1 000 million.

37101 This function shall not return an error code of [EINTR].

37102
37103
37104
37105
37106
37107
37108
37109
37110
37111
37112
37113
37114
37115
37116
37117
37118
37119
37120
37121

EXAMPLES

None.

APPLICATION USAGE

Applications using this function may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_destroy(), *pthread_rwlock_rdlock()*, *pthread_rwlock_timedwrlock()*,
pthread_rwlock_tryrdlock(), *pthread_rwlock_trywrlock()*, *pthread_rwlock_unlock()*,
pthread_rwlock_wrlock(), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10,
Memory Synchronization, **<pthread.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/102 is applied, updating the ERRORS section so that the [EDEADLK] error includes detection of a deadlock condition.

Issue 7

The *pthread_rwlock_timedrdlock()* function is moved from the Timeouts option to the Base.

37122 **NAME**

37123 pthread_rwlock_timedwrlock — lock a read-write lock for writing

37124 **SYNOPSIS**

37125 #include <pthread.h>

37126 #include <time.h>

37127 int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict *rwlock*,
37128 const struct timespec *restrict *abs_timeout*);37129 **DESCRIPTION**37130 The *pthread_rwlock_timedwrlock()* function shall apply a write lock to the read-write lock
37131 referenced by *rwlock* as in the *pthread_rwlock_wrlock()* function. However, if the lock cannot be
37132 acquired without waiting for other threads to unlock the lock, this wait shall be terminated
37133 when the specified timeout expires. The timeout shall expire when the absolute time specified
37134 by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the
37135 value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout*
37136 has already been passed at the time of the call.37137 The timeout shall be based on the CLOCK_REALTIME clock. The resolution of the timeout shall
37138 be the resolution of the CLOCK_REALTIME clock. The **timespec** data type is defined in the
37139 <**time.h**> header. Under no circumstances shall the function fail with a timeout if the lock can be
37140 acquired immediately. The validity of the *abs_timeout* parameter need not be checked if the lock
37141 can be immediately acquired.37142 If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-
37143 write lock via a call to *pthread_rwlock_timedwrlock()*, upon return from the signal handler the
37144 thread shall resume waiting for the lock as if it was not interrupted.37145 The calling thread may deadlock if at the time the call is made it holds the read-write lock. The
37146 results are undefined if this function is called with an uninitialized read-write lock.37147 **RETURN VALUE**37148 The *pthread_rwlock_timedwrlock()* function shall return zero if the lock for writing on the read-
37149 write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned
37150 to indicate the error.37151 **ERRORS**37152 The *pthread_rwlock_timedwrlock()* function shall fail if:

37153 [ETIMEDOUT] The lock could not be acquired before the specified timeout expired.

37154 The *pthread_rwlock_timedwrlock()* function may fail if:37155 [EDEADLK] A deadlock condition was detected or the calling thread already holds the
37156 *rwlock*.37157 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock
37158 object, or the *abs_timeout* nanosecond value is less than zero or greater than or
37159 equal to 1 000 million.

37160 This function shall not return an error code of [EINTR].

37161
37162
37163
37164
37165
37166
37167
37168
37169
37170
37171
37172
37173
37174
37175
37176
37177
37178
37179
37180

EXAMPLES

None.

APPLICATION USAGE

Applications using this function may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_destroy(), *pthread_rwlock_rdlock()*, *pthread_rwlock_timedrdlock()*,
pthread_rwlock_tryrdlock(), *pthread_rwlock_trywrlock()*, *pthread_rwlock_unlock()*,
pthread_rwlock_wrlock(), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10,
Memory Synchronization, **<pthread.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/103 is applied, updating the ERRORS section so that the [EDEADLK] error includes detection of a deadlock condition.

Issue 7

The *pthread_rwlock_timedwrlock()* function is moved from the Timeouts option to the Base.

37181 **NAME**
37182 pthread_rwlock_tryrdlock — lock a read-write lock object for reading

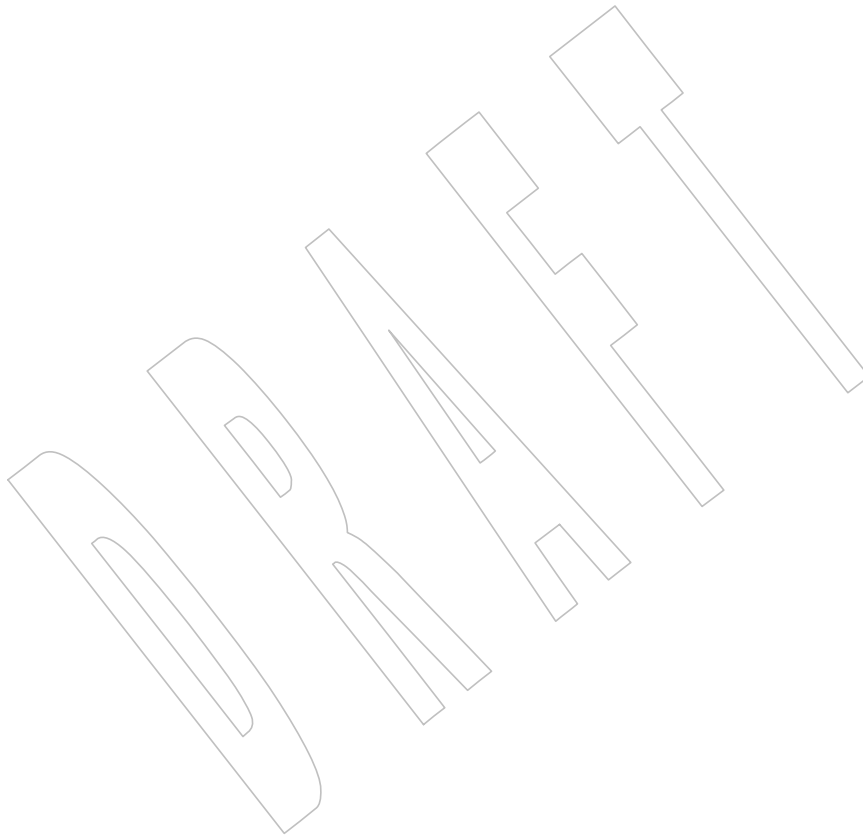
37183 **SYNOPSIS**

37184 #include <pthread.h>

37185 int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

37186 **DESCRIPTION**

37187 Refer to *pthread_rwlock_rdlock()*.



NAME

pthread_rwlock_trywrlock, pthread_rwlock_wrlock — lock a read-write lock object for writing

SYNOPSIS

```
#include <pthread.h>

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

DESCRIPTION

The *pthread_rwlock_trywrlock()* function shall apply a write lock like the *pthread_rwlock_wrlock()* function, with the exception that the function shall fail if any thread currently holds *rwlock* (for reading or writing).

The *pthread_rwlock_wrlock()* function shall apply a write lock to the read-write lock referenced by *rwlock*. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock *rwlock*. Otherwise, the thread shall block until it can acquire the lock. The calling thread may deadlock if at the time the call is made it holds the read-write lock (whether a read or write lock).

Implementations may favor writers over readers to avoid writer starvation.

Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock for writing, upon return from the signal handler the thread resumes waiting for the read-write lock for writing as if it was not interrupted.

RETURN VALUE

The *pthread_rwlock_trywrlock()* function shall return zero if the lock for writing on the read-write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned to indicate the error.

If successful, the *pthread_rwlock_wrlock()* function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_rwlock_trywrlock()* function shall fail if:

pthread_rwlock_trywrlock()*System Interfaces*37225
37226

37227
37228
37229

37230
37231

37232
37233

37234
37235
37236
37237

37238
37239

37240
37241

37242
37243
37244

37245
37246

37247
37248

37249
37250
37251**EXAMPLES**

None.

APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_destroy(), *pthread_rwlock_rdlock()*, *pthread_rwlock_timedrdlock()*, *pthread_rwlock_timedwrlock()*, *pthread_rwlock_tryrdlock()*, *pthread_rwlock_unlock()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, <pthread.h>

CHANGE HISTORY

First released in Issue 5.

Issue 6

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is now part of the Threads option (previously it was part of the Read-Write Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension).
- The [EDEADLK] error is deleted as a *pthread_rwlock_trywrlock()* error.
- The SEE ALSO section is updated.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/104 is applied, updating the ERRORS section so that the [EDEADLK] error includes detection of a deadlock condition.

Issue 7

The *pthread_rwlock_trywrlock()* and *pthread_rwlock_wrlock()* functions are moved from the Threads option to the Base.

37252 **NAME**

37253 pthread_rwlock_unlock — unlock a read-write lock object

37254 **SYNOPSIS**

37255 #include <pthread.h>

37256 int pthread_rwlock_unlock(pthread_rwlock_t **rwlock*);37257 **DESCRIPTION**37258 The *pthread_rwlock_unlock()* function shall release a lock held on the read-write lock object
37259 referenced by *rwlock*. Results are undefined if the read-write lock *rwlock* is not held by the
37260 calling thread.37261 If this function is called to release a read lock from the read-write lock object and there are other
37262 read locks currently held on this read-write lock object, the read-write lock object remains in the
37263 read locked state. If this function releases the last read lock for this read-write lock object, the
37264 read-write lock object shall be put in the unlocked state with no owners.37265 If this function is called to release a write lock for this read-write lock object, the read-write lock
37266 object shall be put in the unlocked state.37267 If there are threads blocked on the lock when it becomes available, the scheduling policy shall
37268 determine which thread(s) shall acquire the lock. If the Thread Execution Scheduling option is
37269 supported, when threads executing with the scheduling policies SCHED_FIFO, SCHED_RR, or
37270 SCHED_SPORADIC are waiting on the lock, they shall acquire the lock in priority order when
37271 the lock becomes available. For equal priority threads, write locks shall take precedence over
37272 read locks. If the Thread Execution Scheduling option is not supported, it is implementation-
37273 defined whether write locks take precedence over read locks.

37274 Results are undefined if any of these functions are called with an uninitialized read-write lock.

37275 **RETURN VALUE**37276 If successful, the *pthread_rwlock_unlock()* function shall return zero; otherwise, an error number
37277 shall be returned to indicate the error.37278 **ERRORS**37279 The *pthread_rwlock_unlock()* function may fail if:37280 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock
37281 object.

37282 [EPERM] The current thread does not hold a lock on the read-write lock.

37283 The *pthread_rwlock_unlock()* function shall not return an error code of [EINTR].37284 **EXAMPLES**

37285 None.

37286 **APPLICATION USAGE**

37287 None.

37288 **RATIONALE**

37289 None.

37290 **FUTURE DIRECTIONS**

37291 None.

37292
37293
37294
37295
37296

37297
37298

37299
37300

37301
37302
37303

37304
37305
37306
37307

37308
37309**SEE ALSO**

pthread_rwlock_destroy(), *pthread_rwlock_rdlock()*, *pthread_rwlock_timedrdlock()*,
pthread_rwlock_timedwrlock(), *pthread_rwlock_tryrdlock()*, *pthread_rwlock_trywrlock()*,
pthread_rwlock_wrlock(), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10,
Memory Synchronization, <**pthread.h**>

CHANGE HISTORY

First released in Issue 5.

Issue 6

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is now part of the Threads option (previously it was part of the Read-Write Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension).
- The DESCRIPTION is updated as follows:
 - The conditions under which writers have precedence over readers are specified.
 - The concept of read-write lock owner is deleted.
- The SEE ALSO section is updated.

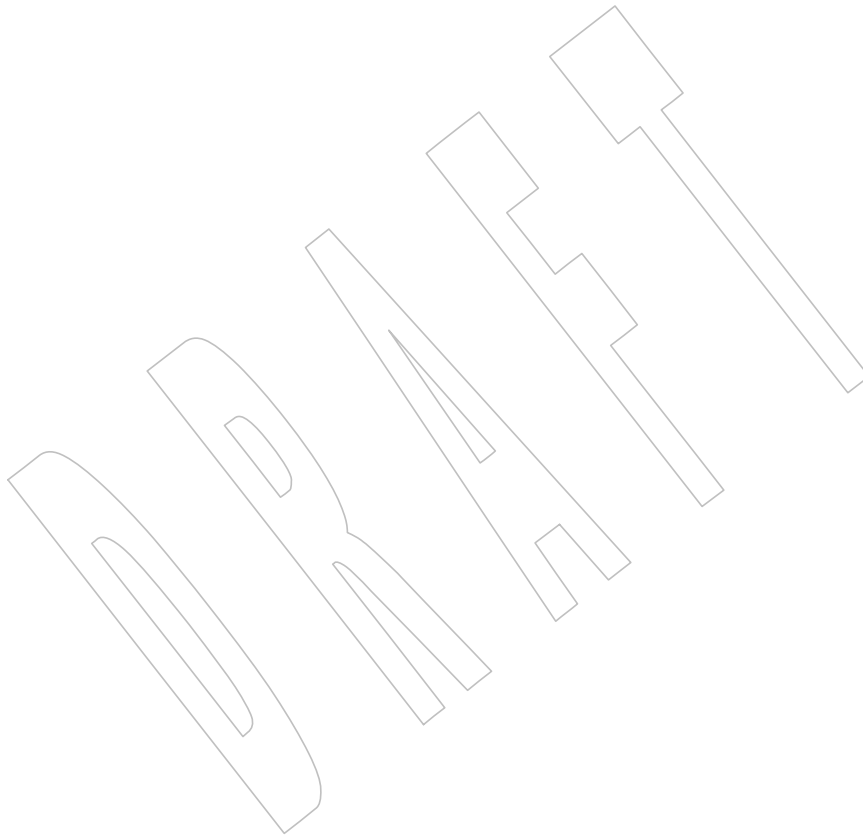
Issue 7

The *pthread_rwlock_unlock()* function is moved from the Threads option to the Base.

37310 **NAME**
37311 pthread_rwlock_wrlock — lock a read-write lock object for writing

37312 **SYNOPSIS**
37313 #include <pthread.h>
37314 int pthread_rwlock_wrlock(pthread_rwlock_t *rlock);

37315 **DESCRIPTION**
37316 Refer to [pthread_rwlock_trywrlock\(\)](#).



37317 **NAME**

37318 pthread_rwlockattr_destroy, pthread_rwlockattr_init — destroy and initialize the read-write
 37319 lock attributes object

37320 **SYNOPSIS**

```
37321 #include <pthread.h>
37322
37322 int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
37323 int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

37324 **DESCRIPTION**

37325 The *pthread_rwlockattr_destroy()* function shall destroy a read-write lock attributes object. A
 37326 destroyed *attr* attributes object can be reinitialized using *pthread_rwlockattr_init()*; the results of
 37327 otherwise referencing the object after it has been destroyed are undefined. An implementation
 37328 may cause *pthread_rwlockattr_destroy()* to set the object referenced by *attr* to an invalid value.

37329 The *pthread_rwlockattr_init()* function shall initialize a read-write lock attributes object *attr* with
 37330 the default value for all of the attributes defined by the implementation.

37331 Results are undefined if *pthread_rwlockattr_init()* is called specifying an already initialized *attr*
 37332 attributes object.

37333 After a read-write lock attributes object has been used to initialize one or more read-write locks,
 37334 any function affecting the attributes object (including destruction) shall not affect any previously
 37335 initialized read-write locks.

37336 **RETURN VALUE**

37337 If successful, the *pthread_rwlockattr_destroy()* and *pthread_rwlockattr_init()* functions shall return
 37338 zero; otherwise, an error number shall be returned to indicate the error.

37339 **ERRORS**

37340 The *pthread_rwlockattr_destroy()* function may fail if:

37341 [EINVAL] The value specified by *attr* is invalid.

37342 The *pthread_rwlockattr_init()* function shall fail if:

37343 [ENOMEM] Insufficient memory exists to initialize the read-write lock attributes object.

37344 These functions shall not return an error code of [EINTR].

37345 **EXAMPLES**

37346 None.

37347 **APPLICATION USAGE**

37348 None.

37349 **RATIONALE**

37350 None.

37351 **FUTURE DIRECTIONS**

37352 None.

37353 **SEE ALSO**

37354 *pthread_rwlock_destroy()*, *pthread_rwlockattr_getpshared()*, *pthread_rwlockattr_setpshared()*, the Base
 37355 Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

37356

CHANGE HISTORY

37357

First released in Issue 5.

37358

Issue 6

37359

The following changes are made for alignment with IEEE Std 1003.1j-2000:

37360

- The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is now part of the Threads option (previously it was part of the Read-Write Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension).

37361

37362

37363

- The SEE ALSO section is updated.

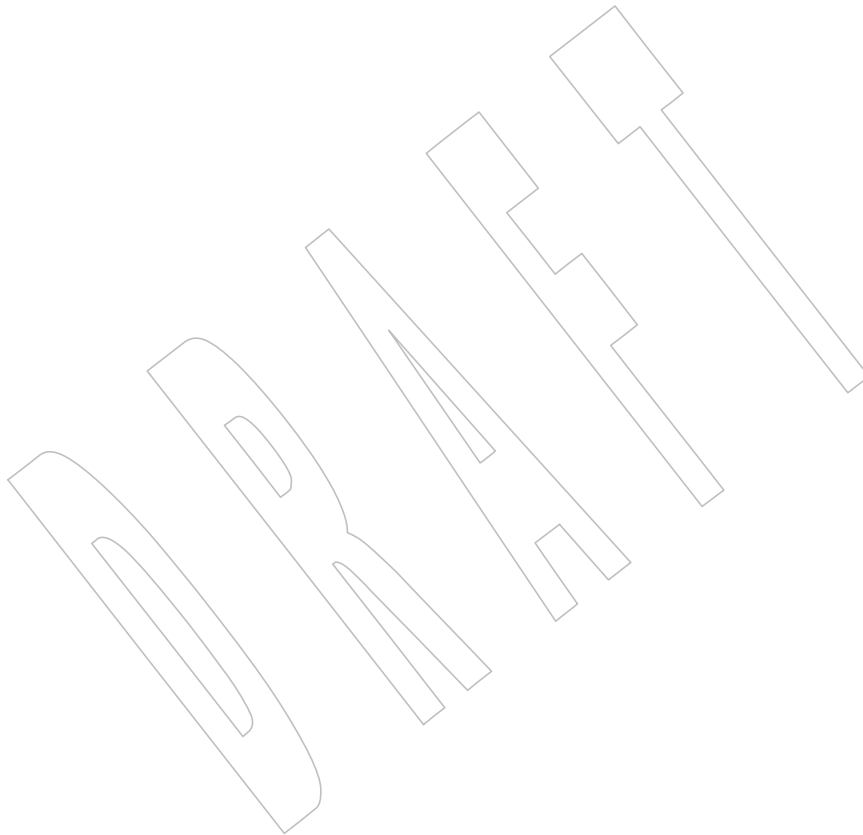
37364

Issue 7

37365

The *pthread_rwlockattr_destroy()* and *pthread_rwlockattr_init()* functions are moved from the Threads option to the Base.

37366



37367 **NAME**

37368 pthread_rwlockattr_getpshared, pthread_rwlockattr_setpshared — get and set the process-
 37369 shared attribute of the read-write lock attributes object

37370 **SYNOPSIS**

```
37371 TSH #include <pthread.h>
37372
37372 int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *
37373     restrict attr, int *restrict pshared);
37374 int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
37375     int pshared);
```

37376 **DESCRIPTION**

37377 The *pthread_rwlockattr_getpshared()* function shall obtain the value of the *process-shared* attribute
 37378 from the initialized attributes object referenced by *attr*. The *pthread_rwlockattr_setpshared()*
 37379 function shall set the *process-shared* attribute in an initialized attributes object referenced by *attr*.

37380 The *process-shared* attribute shall be set to PTHREAD_PROCESS_SHARED to permit a read-write
 37381 lock to be operated upon by any thread that has access to the memory where the read-write lock
 37382 is allocated, even if the read-write lock is allocated in memory that is shared by multiple
 37383 processes. If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the read-write lock
 37384 shall only be operated upon by threads created within the same process as the thread that
 37385 initialized the read-write lock; if threads of differing processes attempt to operate on such a
 37386 read-write lock, the behavior is undefined. The default value of the *process-shared* attribute shall
 37387 be PTHREAD_PROCESS_PRIVATE.

37388 Additional attributes, their default values, and the names of the associated functions to get and
 37389 set those attribute values are implementation-defined.

37390 **RETURN VALUE**

37391 Upon successful completion, the *pthread_rwlockattr_getpshared()* function shall return zero and
 37392 store the value of the *process-shared* attribute of *attr* into the object referenced by the *pshared*
 37393 parameter. Otherwise, an error number shall be returned to indicate the error.

37394 If successful, the *pthread_rwlockattr_setpshared()* function shall return zero; otherwise, an error
 37395 number shall be returned to indicate the error.

37396 **ERRORS**

37397 The *pthread_rwlockattr_getpshared()* and *pthread_rwlockattr_setpshared()* functions may fail if:

37398 [EINVAL] The value specified by *attr* is invalid.

37399 The *pthread_rwlockattr_setpshared()* function may fail if:

37400 [EINVAL] The new value specified for the attribute is outside the range of legal values
 37401 for that attribute.

37402 These functions shall not return an error code of [EINTR].

37403
37404
37405
37406
37407
37408
37409
37410
37411
37412
37413
37414
37415
37416
37417
37418
37419
37420
37421
37422
37423
37424
37425
37426
37427

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_destroy(), *pthread_rwlockattr_destroy()*, *pthread_rwlockattr_init()*, the Base Definitions volume of IEEE Std 1003.1-200x, <pthread.h>

CHANGE HISTORY

First released in Issue 5.

Issue 6

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- The margin code in the SYNOPSIS is changed to THR TSH to indicate that the functionality is now part of the Threads option (previously it was part of the Read-Write Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension).
- The DESCRIPTION notes that additional attributes are implementation-defined.
- The SEE ALSO section is updated.

The **restrict** keyword is added to the *pthread_rwlockattr_getpshared()* prototype for alignment with the ISO/IEC 9899:1999 standard.

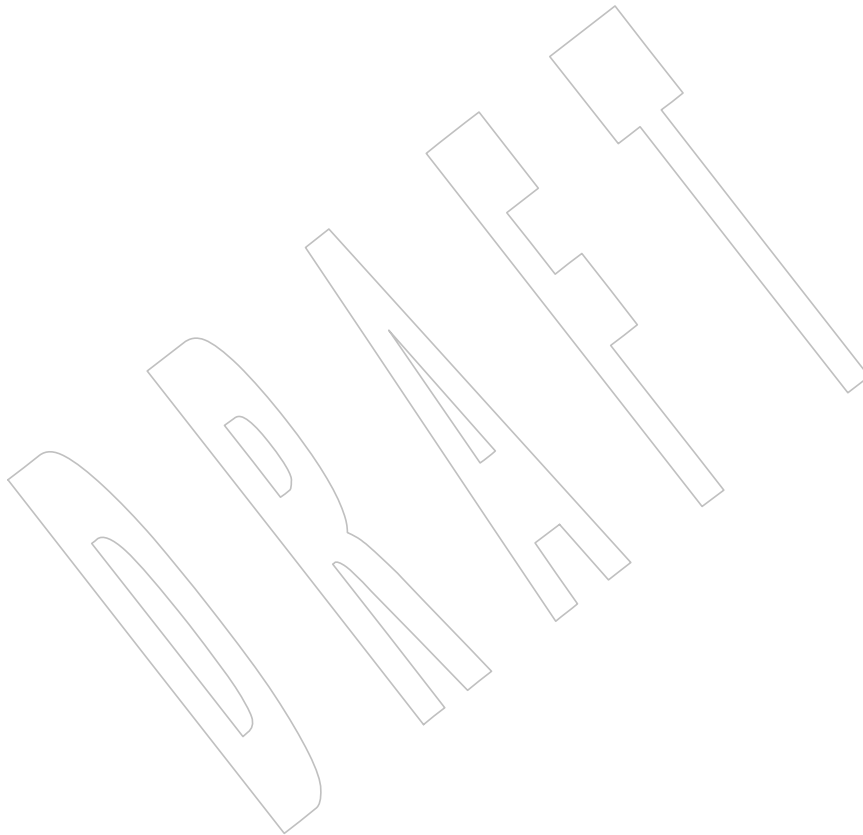
Issue 7

The *pthread_rwlockattr_getpshared()* and *pthread_rwlockattr_setpshared()* functions are moved from the Threads option.

37428 **NAME**
37429 pthread_rwlockattr_init — initialize the read-write lock attributes object

37430 **SYNOPSIS**
37431 #include <pthread.h>
37432 int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);

37433 **DESCRIPTION**
37434 Refer to *pthread_rwlockattr_destroy()*.



37435 **NAME**

37436 pthread_rwlockattr_setpshared — set the process-shared attribute of the read-write lock
37437 attributes object

37438 **SYNOPSIS**

```
37439 TSH #include <pthread.h>  
37440 int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,  
37441 int pshared);
```

37442 **DESCRIPTION**

37443 Refer to [pthread_rwlockattr_getpshared\(\)](#).

37444 **NAME**

37445 pthread_self — get the calling thread ID

37446 **SYNOPSIS**

37447 #include <pthread.h>

37448 pthread_t pthread_self(void);

37449 **DESCRIPTION**

37450 The *pthread_self()* function shall return the thread ID of the calling thread.

37451 **RETURN VALUE**

37452 The *pthread_self()* function shall always be successful and no return value is reserved to indicate

37453 an error.

37454 **ERRORS**

37455 No errors are defined.

37456 **EXAMPLES**

37457 None.

37458 **APPLICATION USAGE**

37459 None.

37460 **RATIONALE**

37461 The *pthread_self()* function provides a capability similar to the *getpid()* function for processes

37462 and the rationale is the same: the creation call does not provide the thread ID to the created

37463 thread.

37464 **FUTURE DIRECTIONS**

37465 None.

37466 **SEE ALSO**

37467 *pthread_create()*, *pthread_equal()*, the Base Definitions volume of IEEE Std 1003.1-200x,

37468 <pthread.h>

37469 **CHANGE HISTORY**

37470 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

37471 **Issue 6**

37472 The *pthread_self()* function is marked as part of the Threads option.

37473 **Issue 7**

37474 Austin Group Interpretation 1003.1-2001 #063 is applied, updating the RETURN VALUE section.

37475 The *pthread_self()* function is moved from the Threads option to the Base.

37476 **NAME**

37477 pthread_setcancelstate, pthread_setcanceltype, pthread_testcancel — set cancelability state

37478 **SYNOPSIS**

37479 #include <pthread.h>

37480 int pthread_setcancelstate(int *state*, int **oldstate*);37481 int pthread_setcanceltype(int *type*, int **oldtype*);

37482 void pthread_testcancel(void);

37483 **DESCRIPTION**

37484 The *pthread_setcancelstate()* function shall atomically both set the calling thread's cancelability state to the indicated *state* and return the previous cancelability state at the location referenced by *oldstate*. Legal values for *state* are PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DISABLE.

37488 The *pthread_setcanceltype()* function shall atomically both set the calling thread's cancelability type to the indicated *type* and return the previous cancelability type at the location referenced by *oldtype*. Legal values for *type* are PTHREAD_CANCEL_DEFERRED and PTHREAD_CANCEL_ASYNCCHRONOUS.

37492 The cancelability state and type of any newly created threads, including the thread in which *main()* was first invoked, shall be PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DEFERRED respectively.

37495 The *pthread_testcancel()* function shall create a cancellation point in the calling thread. The *pthread_testcancel()* function shall have no effect if cancelability is disabled.

37497 **RETURN VALUE**

37498 If successful, the *pthread_setcancelstate()* and *pthread_setcanceltype()* functions shall return zero; otherwise, an error number shall be returned to indicate the error.

37500 **ERRORS**37501 The *pthread_setcancelstate()* function may fail if:

37502 [EINVAL] The specified state is not PTHREAD_CANCEL_ENABLE or
37503 PTHREAD_CANCEL_DISABLE.

37504 The *pthread_setcanceltype()* function may fail if:

37505 [EINVAL] The specified type is not PTHREAD_CANCEL_DEFERRED or
37506 PTHREAD_CANCEL_ASYNCCHRONOUS.

37507 These functions shall not return an error code of [EINTR].

37508 **EXAMPLES**

37509 None.

37510 **APPLICATION USAGE**

37511 None.

37512 **RATIONALE**

37513 The *pthread_setcancelstate()* and *pthread_setcanceltype()* functions control the points at which a
37514 thread may be asynchronously canceled. For cancellation control to be usable in modular
37515 fashion, some rules need to be followed.

37516 An object can be considered to be a generalization of a procedure. It is a set of procedures and
37517 global variables written as a unit and called by clients not known by the object. Objects may
37518 depend on other objects.

37519 First, cancelability should only be disabled on entry to an object, never explicitly enabled. On
 37520 exit from an object, the cancelability state should always be restored to its value on entry to the
 37521 object.

37522 This follows from a modularity argument: if the client of an object (or the client of an object that
 37523 uses that object) has disabled cancelability, it is because the client does not want to be concerned
 37524 about cleaning up if the thread is canceled while executing some sequence of actions. If an object
 37525 is called in such a state and it enables cancelability and a cancellation request is pending for that
 37526 thread, then the thread is canceled, contrary to the wish of the client that disabled.

37527 Second, the cancelability type may be explicitly set to either *deferred* or *asynchronous* upon entry
 37528 to an object. But as with the cancelability state, on exit from an object the cancelability type
 37529 should always be restored to its value on entry to the object.

37530 Finally, only functions that are cancel-safe may be called from a thread that is asynchronously
 37531 cancelable.

37532 FUTURE DIRECTIONS

37533 None.

37534 SEE ALSO

37535 *pthread_cancel()*, the Base Definitions volume of IEEE Std 1003.1-200x, <pthread.h>

37536 CHANGE HISTORY

37537 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

37538 Issue 6

37539 The *pthread_setcancelstate()*, *pthread_setcanceltype()*, and *pthread_testcancel()* functions are marked
 37540 as part of the Threads option.

37541 Issue 7

37542 The *pthread_setcancelstate()*, *pthread_setcanceltype()*, and *pthread_testcancel()* functions are moved
 37543 from the Threads option to the Base.

37544 **NAME**
37545 pthread_setconcurrency — set the level of concurrency

37546 **SYNOPSIS**

37547 OB XSI #include <pthread.h>
37548 int pthread_setconcurrency(int new_level);

37549 **DESCRIPTION**

37550 Refer to [pthread_getconcurrency\(\)](#).

pthread_setschedparam()*System Interfaces*

37551

NAME

37552

pthread_setschedparam — dynamic thread scheduling parameters access (**REALTIME THREADS**)

37553

37554

SYNOPSIS

37555

```
TPS #include <pthread.h>
```

37556

```
int pthread_setschedparam(pthread_t thread, int policy,  
    const struct sched_param *param);
```

37557

37558

DESCRIPTION

37559

Refer to *pthread_getschedparam()*.

37560 **NAME**

37561 pthread_setschedprio — dynamic thread scheduling parameters access (**REALTIME**
 37562 **THREADS**)

37563 **SYNOPSIS**

```
37564 TPS #include <pthread.h>
37565 int pthread_setschedprio(pthread_t thread, int prio);
```

37566 **DESCRIPTION**

37567 The *pthread_setschedprio()* function shall set the scheduling priority for the thread whose thread
 37568 ID is given by *thread* to the value given by *prio*. See [Scheduling Policies](#) for a description on how
 37569 this function call affects the ordering of the thread in the thread list for its new priority.

37570 If the *pthread_setschedprio()* function fails, the scheduling priority of the target thread shall not be
 37571 changed.

37572 **RETURN VALUE**

37573 If successful, the *pthread_setschedprio()* function shall return zero; otherwise, an error number
 37574 shall be returned to indicate the error.

37575 **ERRORS**

37576 The *pthread_setschedprio()* function may fail if:

- | | | |
|-------|-----------|--|
| 37577 | [EINVAL] | The value of <i>prio</i> is invalid for the scheduling policy of the specified thread. |
| 37578 | [ENOTSUP] | An attempt was made to set the priority to an unsupported value. |
| 37579 | [EPERM] | The caller does not have the appropriate permission to set the scheduling
37580 priority of the specified thread. |
| 37581 | [ESRCH] | The value specified by <i>thread</i> does not refer to an existing thread. |
| 37582 | | The <i>pthread_setschedprio()</i> function shall not return an error code of [EINTR]. |

37583 **EXAMPLES**

37584 None.

37585 **APPLICATION USAGE**

37586 None.

37587 **RATIONALE**

37588 The *pthread_setschedprio()* function provides a way for an application to temporarily raise its
 37589 priority and then lower it again, without having the undesired side effect of yielding to other
 37590 threads of the same priority. This is necessary if the application is to implement its own
 37591 strategies for bounding priority inversion, such as priority inheritance or priority ceilings. This
 37592 capability is especially important if the implementation does not support the Thread Priority
 37593 Protection or Thread Priority Inheritance options, but even if those options are supported it is
 37594 needed if the application is to bound priority inheritance for other resources, such as
 37595 semaphores.

37596 The standard developers considered that while it might be preferable conceptually to solve this
 37597 problem by modifying the specification of *pthread_setschedparam()*, it was too late to make such a
 37598 change, as there may be implementations that would need to be changed. Therefore, this new
 37599 function was introduced.

pthread_setschedprio()*System Interfaces*

37600

FUTURE DIRECTIONS

37601

None.

37602

SEE ALSO

37603

[Scheduling Policies](#) (on page 44), [pthread_getschedparam\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, <pthread.h>

37604

37605

CHANGE HISTORY

37606

First released in Issue 6. Included as a response to IEEE PASC Interpretation 1003.1 #96.

37607

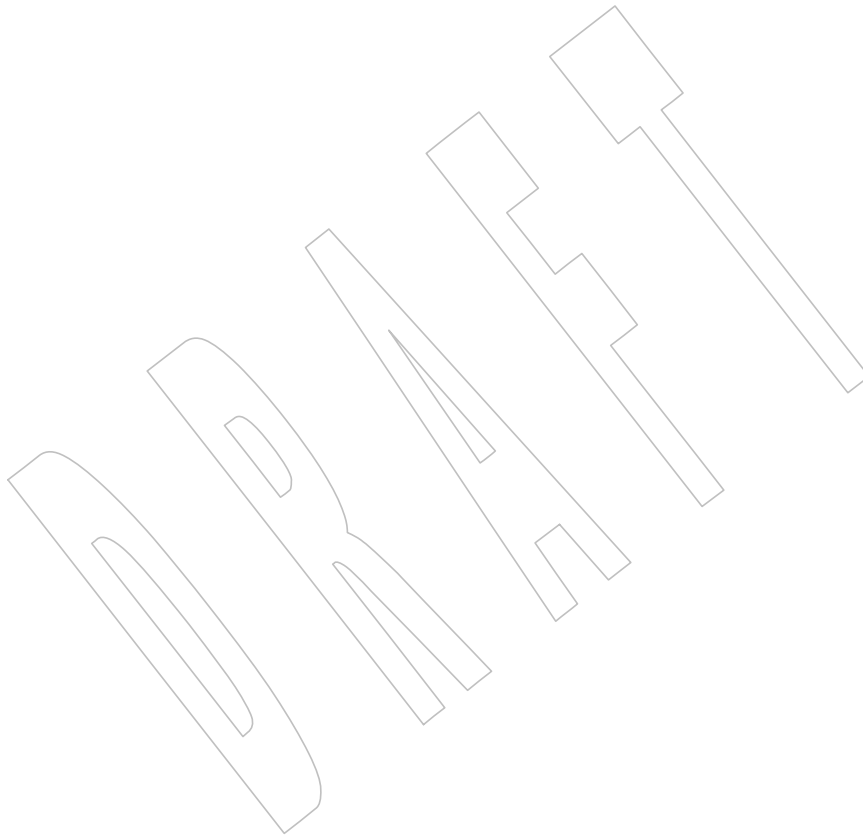
Issue 7

37608

Austin Group Interpretation 1003.1-2001 #069 is applied, updating the [EPERM] error.

37609

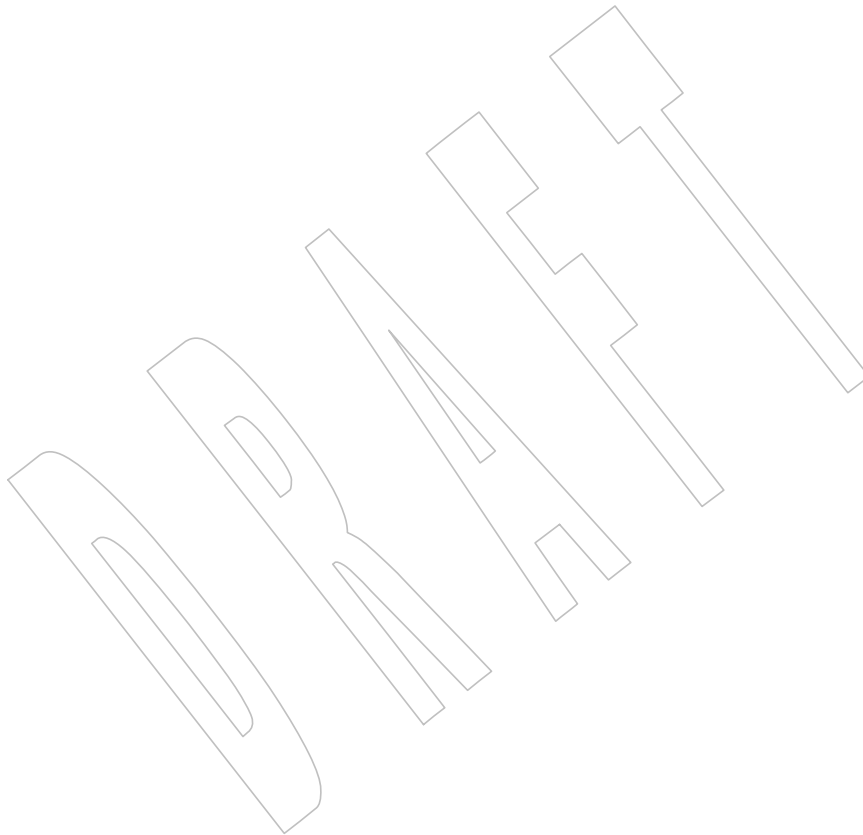
The `pthread_setschedprio()` function is moved from the Threads option.



37610 **NAME**
37611 pthread_setspecific — thread-specific data management

37612 **SYNOPSIS**
37613 #include <pthread.h>
37614 int pthread_setspecific(pthread_key_t key, const void *value);

37615 **DESCRIPTION**
37616 Refer to *pthread_getspecific()*.



37617 **NAME**

37618 pthread_sigmask, sigprocmask — examine and change blocked signals

37619 **SYNOPSIS**

```

37620 CX      #include <signal.h>
37621
37621      int pthread_sigmask(int how, const sigset_t *restrict set,
37622                          sigset_t *restrict oset);
37623      int sigprocmask(int how, const sigset_t *restrict set,
37624                      sigset_t *restrict oset);

```

37625 **DESCRIPTION**

37626 The *pthread_sigmask()* function shall examine or change (or both) the calling thread's signal
 37627 mask, regardless of the number of threads in the process. The function shall be equivalent to
 37628 *sigprocmask()*, without the restriction that the call be made in a single-threaded process.

37629 In a single-threaded process, the *sigprocmask()* function shall examine or change (or both) the
 37630 signal mask of the calling thread.

37631 If the argument *set* is not a null pointer, it points to a set of signals to be used to change the
 37632 currently blocked set.

37633 The argument *how* indicates the way in which the set is changed, and the application shall
 37634 ensure it consists of one of the following values:

37635 SIG_BLOCK The resulting set shall be the union of the current set and the signal set
 37636 pointed to by *set*.

37637 SIG_SETMASK The resulting set shall be the signal set pointed to by *set*.

37638 SIG_UNBLOCK The resulting set shall be the intersection of the current set and the
 37639 complement of the signal set pointed to by *set*.

37640 If the argument *oset* is not a null pointer, the previous mask shall be stored in the location
 37641 pointed to by *oset*. If *set* is a null pointer, the value of the argument *how* is not significant and the
 37642 thread's signal mask shall be unchanged; thus the call can be used to enquire about currently
 37643 blocked signals.

37644 If there are any pending unblocked signals after the call to *sigprocmask()*, at least one of those
 37645 signals shall be delivered before the call to *sigprocmask()* returns.

37646 It is not possible to block those signals which cannot be ignored. This shall be enforced by the
 37647 system without causing an error to be indicated.

37648 If any of the SIGFPE, SIGILL, SIGSEGV, or SIGBUS signals are generated while they are blocked,
 37649 the result is undefined, unless the signal was generated by the *kill()* function, the *sigqueue()*
 37650 function, or the *raise()* function.

37651 If *sigprocmask()* fails, the thread's signal mask shall not be changed.

37652 The use of the *sigprocmask()* function is unspecified in a multi-threaded process.

37653 **RETURN VALUE**

37654 Upon successful completion *pthread_sigmask()* shall return 0; otherwise, it shall return the
 37655 corresponding error number.

37656 Upon successful completion, *sigprocmask()* shall return 0; otherwise, -1 shall be returned, *errno*
 37657 shall be set to indicate the error, and the signal mask of the process shall be unchanged.

ERRORS

37658
37659 The *pthread_sigmask()* and *sigprocmask()* functions shall fail if:

37660 [EINVAL] The value of the *how* argument is not equal to one of the defined values.

37661 The *pthread_sigmask()* function shall not return an error code of [EINTR].

EXAMPLES**Signalling in a Multi-Threaded Process**

37662
37663 This example shows the use of *pthread_sigmask()* in order to deal with signals in a multi-threaded process. It provides a fairly general framework that could be easily adapted/extended.

```

37666 #include <stdio.h>
37667 #include <stdlib.h>
37668 #include <pthread.h>
37669 #include <signal.h>
37670 #include <string.h>
37671 #include <errno.h>
37672 ...
37673 static sigset_t  signal_mask; /* signals to block          */
37674 int main (int argc, char *argv[])
37675 {
37676     pthread_t  sig_thr_id; /* signal handler thread ID */
37677     int        rc; /* return code          */
37678     sigemptyset (&signal_mask);
37679     sigaddset (&signal_mask, SIGINT);
37680     sigaddset (&signal_mask, SIGTERM);
37681     rc = pthread_sigmask (SIG_BLOCK, &signal_mask, NULL);
37682     if (rc != 0) {
37683         /* handle error */
37684         ...
37685     }
37686     /* any newly created threads inherit the signal mask */
37687     rc = pthread_create (&sig_thr_id, NULL, signal_thread, NULL);
37688     if (rc != 0) {
37689         /* handle error */
37690         ...
37691     }
37692     /* APPLICATION CODE */
37693     ...
37694 }
37695 void *signal_thread (void *arg)
37696 {
37697     int        sig_caught; /* signal caught          */
37698     int        rc; /* returned code          */
37699     rc = sigwait (&signal_mask, &sig_caught);
37700     if (rc != 0) {
37701         /* handle error */
37702     }
37703     switch (sig_caught)
37704     {

```

```

37705         case SIGINT:      /* process SIGINT */
37706             ...
37707             break;
37708         case SIGTERM:     /* process SIGTERM */
37709             ...
37710             break;
37711         default:         /* should normally not happen */
37712             fprintf (stderr, "\nUnexpected signal %d\n", sig_caught);
37713             break;
37714     }
37715 }

```

APPLICATION USAGE

None.

RATIONALE

When a thread's signal mask is changed in a signal-catching function that is installed by *sigaction()*, the restoration of the signal mask on return from the signal-catching function overrides that change (see *sigaction()*). If the signal-catching function was installed with *signal()*, it is unspecified whether this occurs.

See *kill()* for a discussion of the requirement on delivery of signals.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *sigaction()*, *sigaddset()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*, *sigpending()*, *sigqueue()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

The *pthread_sigmask()* function is added for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_sigmask()* function is marked as part of the Threads option.

The SYNOPSIS for *sigprocmask()* is marked as a CX extension to note that the presence of this function in the **<signal.h>** header is an extension to the ISO C standard.

The following changes are made for alignment with the ISO POSIX-1:1996 standard:

- The DESCRIPTION is updated to explicitly state the functions which may generate the signal.

The normative text is updated to avoid use of the term “must” for application requirements.

The **restrict** keyword is added to the *pthread_sigmask()* and *sigprocmask()* prototypes for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/105 is applied, updating “process’ signal mask” to “thread’s signal mask” in the DESCRIPTION and RATIONALE sections.

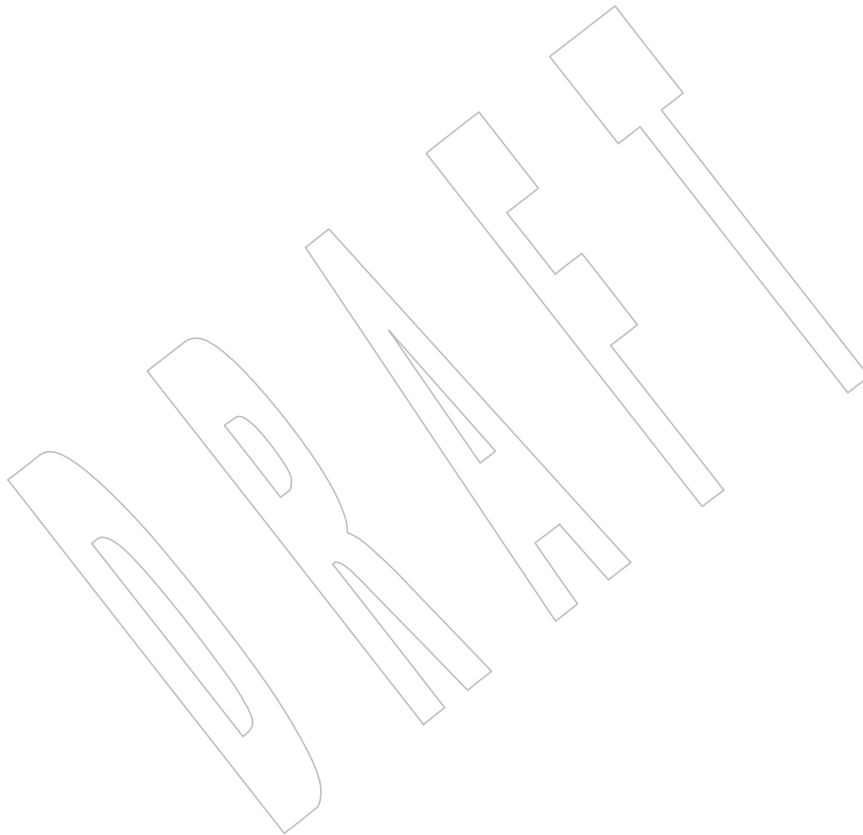
IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/106 is applied, adding the example to the EXAMPLES section.

37748

Issue 7

37749

The *pthread_sigmask()* function is moved from the Threads option to the Base.



NAME

pthread_spin_destroy, pthread_spin_init — destroy or initialize a spin lock object

SYNOPSIS

```
#include <pthread.h>

int pthread_spin_destroy(pthread_spinlock_t *lock);
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```

DESCRIPTION

The *pthread_spin_destroy()* function shall destroy the spin lock referenced by *lock* and release any resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is reinitialized by another call to *pthread_spin_init()*. The results are undefined if *pthread_spin_destroy()* is called when a thread holds the lock, or if this function is called with an uninitialized thread spin lock.

The *pthread_spin_init()* function shall allocate any resources required to use the spin lock referenced by *lock* and initialize the lock to an unlocked state.

TSH

If the Thread Process-Shared Synchronization option is supported and the value of *pshared* is `PTHREAD_PROCESS_SHARED`, the implementation shall permit the spin lock to be operated upon by any thread that has access to the memory where the spin lock is allocated, even if it is allocated in memory that is shared by multiple processes.

If the Thread Process-Shared Synchronization option is supported and the value of *pshared* is `PTHREAD_PROCESS_PRIVATE`, or if the option is not supported, the spin lock shall only be

37791 [ENOMEM] Insufficient memory exists to initialize the lock.

37792 These functions shall not return an error code of [EINTR].

37793 EXAMPLES

37794 None.

37795 APPLICATION USAGE

37796 None.

37797 RATIONALE

37798 None.

37799 FUTURE DIRECTIONS

37800 None.

37801 SEE ALSO

37802 *pthread_spin_lock()*, *pthread_spin_unlock()*, the Base Definitions volume of IEEE Std 1003.1-200x,
37803 **<pthread.h>**

37804 CHANGE HISTORY

37805 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

37806 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

37807 Issue 7

37808 The *pthread_spin_destroy()* and *pthread_spin_init()* functions are moved from the Spin Locks
37809 option to the Base.

DRAFT

37810 **NAME**

37811 pthread_spin_lock, pthread_spin_trylock — lock a spin lock object

37812 **SYNOPSIS**

37813 #include <pthread.h>

37814 int pthread_spin_lock(pthread_spinlock_t *lock);

37815 int pthread_spin_trylock(pthread_spinlock_t *lock);

37816 **DESCRIPTION**

37817 The *pthread_spin_lock()* function shall lock the spin lock referenced by *lock*. The calling thread
 37818 shall acquire the lock if it is not held by another thread. Otherwise, the thread shall spin (that is,
 37819 shall not return from the *pthread_spin_lock()* call) until the lock becomes available. The results are
 37820 undefined if the calling thread holds the lock at the time the call is made. The
 37821 *pthread_spin_trylock()* function shall lock the spin lock referenced by *lock* if it is not held by any
 37822 thread. Otherwise, the function shall fail.

37823 The results are undefined if any of these functions is called with an uninitialized spin lock.

37824 **RETURN VALUE**

37825 Upon successful completion, these functions shall return zero; otherwise, an error number shall
 37826 be returned to indicate the error.

37827 **ERRORS**

37828 These functions may fail if:

37829 [EINVAL] The value specified by *lock* does not refer to an initialized spin lock object.37830 The *pthread_spin_lock()* function may fail if:

37831 [EDEADLK] A deadlock condition was detected or the calling thread already holds the
 37832 lock.

37833 The *pthread_spin_trylock()* function shall fail if:

37834 [EBUSY] A thread currently holds the lock.

37835 These functions shall not return an error code of [EINTR].

37836 **EXAMPLES**

37837 None.

37838 **APPLICATION USAGE**

37839 Applications using this function may be subject to priority inversion, as discussed in the Base
 37840 Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

37841 **RATIONALE**

37842 None.

37843 **FUTURE DIRECTIONS**

37844 None.

37845 **SEE ALSO**

37846 *pthread_spin_destroy()*, *pthread_spin_unlock()*, the Base Definitions volume of
 37847 IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, <pthread.h>

37848 **CHANGE HISTORY**

37849 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

37850 In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.

37851 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/107 is applied, updating the ERRORS

37852

section so that the [EDEADLK] error includes detection of a deadlock condition.

37853

Issue 7

37854

The *pthread_spin_lock()* and *pthread_spin_trylock()* functions are moved from the Spin Locks option to the Base.

37855



37856 **NAME**

37857 pthread_spin_unlock — unlock a spin lock object

37858 **SYNOPSIS**

37859 #include <pthread.h>

37860 int pthread_spin_unlock(pthread_spinlock_t *lock);

37861 **DESCRIPTION**

37862 The *pthread_spin_unlock()* function shall release the spin lock referenced by *lock* which was
 37863 locked via the *pthread_spin_lock()* or *pthread_spin_trylock()* functions. The results are undefined if
 37864 the lock is not held by the calling thread. If there are threads spinning on the lock when
 37865 *pthread_spin_unlock()* is called, the lock becomes available and an unspecified spinning thread
 37866 shall acquire the lock.

37867 The results are undefined if this function is called with an uninitialized thread spin lock.

37868 **RETURN VALUE**

37869 Upon successful completion, the *pthread_spin_unlock()* function shall return zero; otherwise, an
 37870 error number shall be returned to indicate the error.

37871 **ERRORS**37872 The *pthread_spin_unlock()* function may fail if:

37873 [EINVAL] An invalid argument was specified.

37874 [EPERM] The calling thread does not hold the lock.

37875 This function shall not return an error code of [EINTR].

37876 **EXAMPLES**

37877 None.

37878 **APPLICATION USAGE**

37879 None.

37880 **RATIONALE**

37881 None.

37882 **FUTURE DIRECTIONS**

37883 None.

37884 **SEE ALSO**

37885 *pthread_spin_destroy()*, *pthread_spin_lock()*, the Base Definitions volume of IEEE Std 1003.1-200x,
 37886 Section 4.10, Memory Synchronization, <pthread.h>

37887 **CHANGE HISTORY**

37888 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

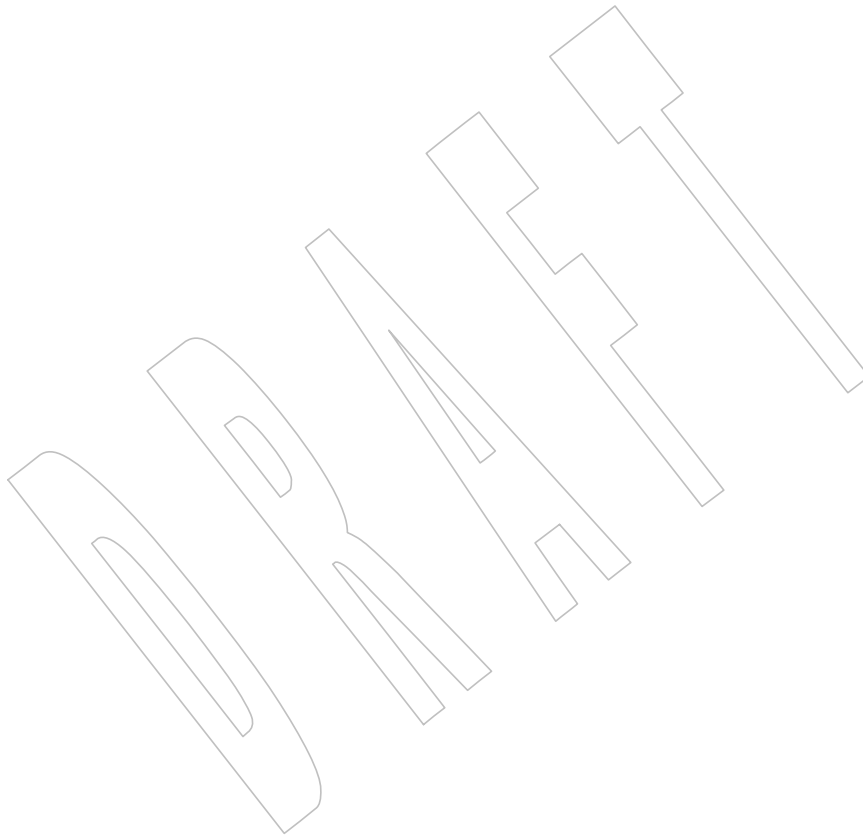
37889 In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.

37890 **Issue 7**37891 The *pthread_spin_unlock()* function is moved from the Spin Locks option to the Base.

37892 **NAME**
37893 pthread_testcancel — set cancelability state

37894 **SYNOPSIS**
37895 #include <pthread.h>
37896 void pthread_testcancel(void);

37897 **DESCRIPTION**
37898 Refer to *pthread_setcancelstate()*.



37899 **NAME**
 37900 ptsname — get name of the slave pseudo-terminal device

37901 **SYNOPSIS**
 37902 XSI `#include <stdlib.h>`
 37903 `char *ptsname(int fildes);`

37904 **DESCRIPTION**
 37905 The *ptsname()* function shall return the name of the slave pseudo-terminal device associated
 37906 with a master pseudo-terminal device. The *fildes* argument is a file descriptor that refers to the
 37907 master device. The *ptsname()* function shall return a pointer to a string containing the pathname
 37908 of the corresponding slave device.

37909 The *ptsname()* function need not be thread-safe. A function that is not required to be thread-safe
 37910 is not required to be reentrant.

37911 **RETURN VALUE**
 37912 Upon successful completion, *ptsname()* shall return a pointer to a string which is the name of the
 37913 pseudo-terminal slave device. Upon failure, *ptsname()* shall return a null pointer. This could
 37914 occur if *fildes* is an invalid file descriptor or if the slave device name does not exist in the file
 37915 system.

37916 **ERRORS**
 37917 No errors are defined.

37918 **EXAMPLES**
 37919 None.

37920 **APPLICATION USAGE**
 37921 The value returned may point to a static data area that is overwritten by each call to *ptsname()*.

37922 **RATIONALE**
 37923 None.

37924 **FUTURE DIRECTIONS**
 37925 None.

37926 **SEE ALSO**
 37927 *grantpt()*, *open()*, *ttyname()*, *unlockpt()*, the Base Definitions volume of IEEE Std 1003.1-200x,
 37928 `<stdlib.h>`

37929 **CHANGE HISTORY**
 37930 First released in Issue 4, Version 2.

37931 **Issue 5**
 37932 Moved from X/OPEN UNIX extension to BASE.

37933 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

37934 **NAME**
 37935 putc — put a byte on a stream

37936 **SYNOPSIS**
 37937 #include <stdio.h>
 37938 int putc(int *c*, FILE **stream*);

37939 **DESCRIPTION**
 37940 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 37941 conflict between the requirements described here and the ISO C standard is unintentional. This
 37942 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

37943 The *putc()* function shall be equivalent to *fputc()*, except that if it is implemented as a macro it
 37944 may evaluate *stream* more than once, so the argument should never be an expression with side
 37945 effects.

37946 **RETURN VALUE**
 37947 Refer to *fputc()*.

37948 **ERRORS**
 37949 Refer to *fputc()*.

37950 **EXAMPLES**
 37951 None.

37952 **APPLICATION USAGE**
 37953 Since it may be implemented as a macro, *putc()* may treat a *stream* argument with side effects
 37954 incorrectly. In particular, *putc(c,*f++)* does not necessarily work correctly. Therefore, use of this
 37955 function is not recommended in such situations; *fputc()* should be used instead.

37956 **RATIONALE**
 37957 None.

37958 **FUTURE DIRECTIONS**
 37959 None.

37960 **SEE ALSO**
 37961 *fputc()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>

37962 **CHANGE HISTORY**
 37963 First released in Issue 1. Derived from Issue 1 of the SVID.

putc_unlocked()

37964 **NAME**
37965 putc_unlocked — stdio with explicit client locking

37966 **SYNOPSIS**

37967 CX #include <stdio.h>
37968 int putc_unlocked(int *c*, FILE **stream*);

37969 **DESCRIPTION**

37970 Refer to [getc_unlocked\(\)](#).

37971 **NAME**
37972 putchar — put a byte on a stdout stream

37973 **SYNOPSIS**
37974 #include <stdio.h>
37975 int putchar(int c);

37976 **DESCRIPTION**
37977 CX The functionality described on this reference page is aligned with the ISO C standard. Any
37978 conflict between the requirements described here and the ISO C standard is unintentional. This
37979 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

37980 The function call *putchar(c)* shall be equivalent to *putc(c,stdout)*.

37981 **RETURN VALUE**
37982 Refer to *fputc()*.

37983 **ERRORS**
37984 Refer to *fputc()*.

37985 **EXAMPLES**
37986 None.

37987 **APPLICATION USAGE**
37988 None.

37989 **RATIONALE**
37990 None.

37991 **FUTURE DIRECTIONS**
37992 None.

37993 **SEE ALSO**
37994 *putc()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>

37995 **CHANGE HISTORY**
37996 First released in Issue 1. Derived from Issue 1 of the SVID.

putchar_unlocked()*System Interfaces*

37997 **NAME**
37998 putchar_unlocked — stdio with explicit client locking

37999 **SYNOPSIS**

38000 CX #include <stdio.h>
38001 int putchar_unlocked(int c);

38002 **DESCRIPTION**

38003 Refer to *getc_unlocked()*.

38004 **NAME**
 38005 putenv — change or add a value to an environment

38006 **SYNOPSIS**

38007 XSI `#include <stdlib.h>`
 38008 `int putenv(char *string);`

38009 **DESCRIPTION**

38010 The *putenv()* function shall use the *string* argument to set environment variable values. The
 38011 *string* argument should point to a string of the form "*name=value*". The *putenv()* function shall
 38012 make the value of the environment variable *name* equal to *value* by altering an existing variable
 38013 or creating a new one. In either case, the string pointed to by *string* shall become part of the
 38014 environment, so altering the string shall change the environment. The space used by *string* is no
 38015 longer used once a new string which defines *name* is passed to *putenv()*.

38016 The *putenv()* function need not be thread-safe. A function that is not required to be thread-safe is
 38017 not required to be reentrant.

38018 **RETURN VALUE**

38019 Upon successful completion, *putenv()* shall return 0; otherwise, it shall return a non-zero value
 38020 and set *errno* to indicate the error.

38021 **ERRORS**

38022 The *putenv()* function may fail if:
 38023 [ENOMEM] Insufficient memory was available.

38024 **EXAMPLES**

38025 **Changing the Value of an Environment Variable**

38026 The following example changes the value of the *HOME* environment variable to the value
 38027 */usr/home*.

```
38028 #include <stdlib.h>
38029 ...
38030 static char *var = "HOME=/usr/home";
38031 int ret;
38032
38033 ret = putenv(var);
```

38033 **APPLICATION USAGE**

38034 The *putenv()* function manipulates the environment pointed to by *environ*, and can be used in
 38035 conjunction with *getenv()*.

38036 See *exec*, for restrictions on changing the environment in multi-threaded applications.

38037 This routine may use *malloc()* to enlarge the environment.

38038 A potential error is to call *putenv()* with an automatic variable as the argument, then return from
 38039 the calling function while *string* is still part of the environment.

38040 The *setenv()* function is preferred over this function.

38041
38042
38043
38044
38045
38046
38047
38048
38049
38050
38051
38052
38053
38054
38055
38056
38057

RATIONALE

The standard developers noted that *putenv()* is the only function available to add to the environment without permitting memory leaks.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *getenv()*, *malloc()*, *setenv()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdlib.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

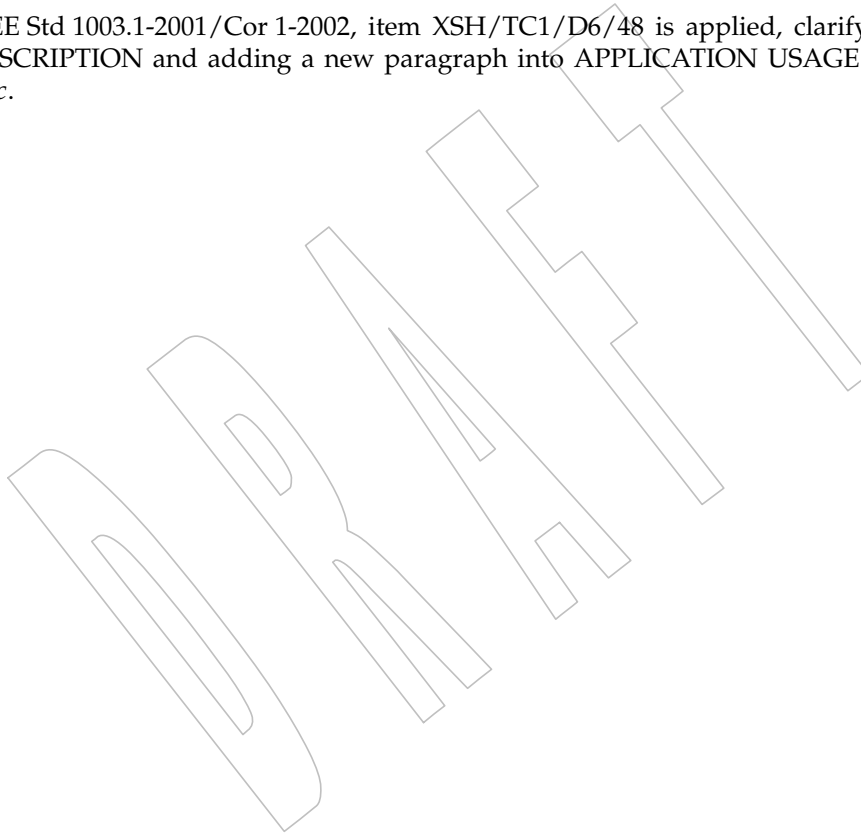
Issue 5

The type of the argument to this function is changed from **const char *** to **char ***. This was indicated as a FUTURE DIRECTION in previous issues.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/48 is applied, clarifying wording in the DESCRIPTION and adding a new paragraph into APPLICATION USAGE referring readers to *exec*.



38058 **NAME**
 38059 putmsg, putpmsg — send a message on a STREAM (**STREAMS**)

38060 **SYNOPSIS**

```
38061 OB XSR #include <stropts.h>
38062
38062 int putmsg(int fildes, const struct strbuf *ctlptr,
38063           const struct strbuf *dataptr, int flags);
38064 int putpmsg(int fildes, const struct strbuf *ctlptr,
38065            const struct strbuf *dataptr, int band, int flags);
```

38066 **DESCRIPTION**

38067 The *putmsg()* function shall create a message from a process buffer(s) and send the message to a
 38068 STREAMS file. The message may contain either a data part, a control part, or both. The data and
 38069 control parts are distinguished by placement in separate buffers, as described below. The
 38070 semantics of each part are defined by the STREAMS module that receives the message.

38071 The *putpmsg()* function is equivalent to *putmsg()*, except that the process can send messages in
 38072 different priority bands. Except where noted, all requirements on *putmsg()* also pertain to
 38073 *putpmsg()*.

38074 The *fildes* argument specifies a file descriptor referencing an open STREAM. The *ctlptr* and
 38075 *dataptr* arguments each point to a **strbuf** structure.

38076 The *ctlptr* argument points to the structure describing the control part, if any, to be included in
 38077 the message. The *buf* member in the **strbuf** structure points to the buffer where the control
 38078 information resides, and the *len* member indicates the number of bytes to be sent. The *maxlen*
 38079 member is not used by *putmsg()*. In a similar manner, the argument *dataptr* specifies the data, if
 38080 any, to be included in the message. The *flags* argument indicates what type of message should be
 38081 sent and is described further below.

38082 To send the data part of a message, the application shall ensure that *dataptr* is not a null pointer
 38083 and the *len* member of *dataptr* is 0 or greater. To send the control part of a message, the
 38084 application shall ensure that the corresponding values are set for *ctlptr*. No data (control) part
 38085 shall be sent if either *dataptr(ctlptr)* is a null pointer or the *len* member of *dataptr(ctlptr)* is set to
 38086 -1.

38087 For *putmsg()*, if a control part is specified and *flags* is set to RS_HIPRI, a high priority message
 38088 shall be sent. If no control part is specified, and *flags* is set to RS_HIPRI, *putmsg()* shall fail and
 38089 set *errno* to [EINVAL]. If *flags* is set to 0, a normal message (priority band equal to 0) shall be
 38090 sent. If a control part and data part are not specified and *flags* is set to 0, no message shall be
 38091 sent and 0 shall be returned.

38092 For *putpmsg()*, the flags are different. The *flags* argument is a bitmask with the following
 38093 mutually-exclusive flags defined: MSG_HIPRI and MSG_BAND. If *flags* is set to 0, *putpmsg()*
 38094 shall fail and set *errno* to [EINVAL]. If a control part is specified and *flags* is set to MSG_HIPRI
 38095 and *band* is set to 0, a high-priority message shall be sent. If *flags* is set to MSG_HIPRI and either
 38096 no control part is specified or *band* is set to a non-zero value, *putpmsg()* shall fail and set *errno* to
 38097 [EINVAL]. If *flags* is set to MSG_BAND, then a message shall be sent in the priority band
 38098 specified by *band*. If a control part and data part are not specified and *flags* is set to MSG_BAND,
 38099 no message shall be sent and 0 shall be returned.

38100 The *putmsg()* function shall block if the STREAM write queue is full due to internal flow control
 38101 conditions, with the following exceptions:

putmsg()

- 38102 • For high-priority messages, *putmsg()* shall not block on this condition and continues
38103 processing the message.
- 38104 • For other messages, *putmsg()* shall not block but shall fail when the write queue is full and
38105 O_NONBLOCK is set.

38106 The *putmsg()* function shall also block, unless prevented by lack of internal resources, while
38107 waiting for the availability of message blocks in the STREAM, regardless of priority or whether
38108 O_NONBLOCK has been specified. No partial message shall be sent.

RETURN VALUE

38109 Upon successful completion, *putmsg()* and *putpmsg()* shall return 0; otherwise, they shall return
38110 -1 and set *errno* to indicate the error.
38111

ERRORS

38112 The *putmsg()* and *putpmsg()* functions shall fail if:

- 38114 [EAGAIN] A non-priority message was specified, the O_NONBLOCK flag is set, and the
38115 STREAM write queue is full due to internal flow control conditions; or buffers
38116 could not be allocated for the message that was to be created.
- 38117 [EBADF] *fildev* is not a valid file descriptor open for writing.
- 38118 [EINTR] A signal was caught during *putmsg()*.
- 38119 [EINVAL] An undefined value is specified in *flags*, or *flags* is set to RS_HIPRI or
38120 MSG_HIPRI and no control part is supplied, or the STREAM or multiplexer
38121 referenced by *fildev* is linked (directly or indirectly) downstream from a
38122 multiplexer, or *flags* is set to MSG_HIPRI and *band* is non-zero (for *putpmsg()*
38123 only).
- 38124 [ENOSR] Buffers could not be allocated for the message that was to be created due to
38125 insufficient STREAMS memory resources.
- 38126 [ENOSTR] A STREAM is not associated with *fildev*.
- 38127 [ENXIO] A hangup condition was generated downstream for the specified STREAM.
- 38128 [EPIPE] or [EIO] The *fildev* argument refers to a STREAMS-based pipe and the other end of the
38129 pipe is closed. A SIGPIPE signal is generated for the calling thread.
- 38130 [ERANGE] The size of the data part of the message does not fall within the range
38131 specified by the maximum and minimum packet sizes of the topmost
38132 STREAM module. This value is also returned if the control part of the message
38133 is larger than the maximum configured size of the control part of a message,
38134 or if the data part of a message is larger than the maximum configured size of
38135 the data part of a message.

38136 In addition, *putmsg()* and *putpmsg()* shall fail if the STREAM head had processed an
38137 asynchronous error before the call. In this case, the value of *errno* does not reflect the result of
38138 *putmsg()* or *putpmsg()*, but reflects the prior error.

38139 **EXAMPLES**38140 **Sending a High-Priority Message**

38141 The value of *fd* is assumed to refer to an open STREAMS file. This call to *putmsg()* does the
 38142 following:

- 38143 1. Creates a high-priority message with a control part and a data part, using the buffers
 38144 pointed to by *ctrlbuf* and *databuf*, respectively.
- 38145 2. Sends the message to the STREAMS file identified by *fd*.

```

38146 #include <stropts.h>
38147 #include <string.h>
38148 ...
38149 int fd;
38150 char *ctrlbuf = "This is the control part";
38151 char *databuf = "This is the data part";
38152 struct strbuf ctrl;
38153 struct strbuf data;
38154 int ret;

38155 ctrl.buf = ctrlbuf;
38156 ctrl.len = strlen(ctrlbuf);

38157 data.buf = databuf;
38158 data.len = strlen(databuf);

38159 ret = putmsg(fd, &ctrl, &data, MSG_HIPRI);

```

38160 **Using putpmsg()**

38161 This example has the same effect as the previous example. In this example, however, the
 38162 *putpmsg()* function creates and sends the message to the STREAMS file.

```

38163 #include <stropts.h>
38164 #include <string.h>
38165 ...
38166 int fd;
38167 char *ctrlbuf = "This is the control part";
38168 char *databuf = "This is the data part";
38169 struct strbuf ctrl;
38170 struct strbuf data;
38171 int ret;

38172 ctrl.buf = ctrlbuf;
38173 ctrl.len = strlen(ctrlbuf);

38174 data.buf = databuf;
38175 data.len = strlen(databuf);

38176 ret = putpmsg(fd, &ctrl, &data, 0, MSG_HIPRI);

```

38177 **APPLICATION USAGE**

38178 None.

38179 **RATIONALE**

38180 None.

38181

FUTURE DIRECTIONS

38182

The *putmsg()* and *putpmsg()* functions may be removed in a future version.

38183

SEE ALSO

38184

Section 2.6 (on page 38), *getmsg()*, *poll()*, *read()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**

38185

38186

CHANGE HISTORY

38187

First released in Issue 4, Version 2.

38188

Issue 5

38189

Moved from X/OPEN UNIX extension to BASE.

38190

The following text is removed from the DESCRIPTION: “The STREAM head guarantees that the control part of a message generated by *putmsg()* is at least 64 bytes in length”.

38191

38192

Issue 6

38193

This function is marked as part of the XSI STREAMS Option Group.

38194

The normative text is updated to avoid use of the term “must” for application requirements.

38195

Issue 7

38196

The *putmsg()* and *putpmsg()* functions are marked obsolescent.

DRAFT

38197 **NAME**

38198 puts — put a string on standard output

38199 **SYNOPSIS**

38200 #include <stdio.h>

38201 int puts(const char *s);

38202 **DESCRIPTION**

38203 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 38204 conflict between the requirements described here and the ISO C standard is unintentional. This
 38205 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

38206 The *puts()* function shall write the string pointed to by *s*, followed by a <newline>, to the
 38207 standard output stream *stdout*. The terminating null byte shall not be written.

38208 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
 38209 execution of *puts()* and the next successful completion of a call to *fflush()* or *fclose()* on the same
 38210 stream or a call to *exit()* or *abort()*.

38211 **RETURN VALUE**

38212 Upon successful completion, *puts()* shall return a non-negative number. Otherwise, it shall
 38213 CX return EOF, shall set an error indicator for the stream, and *errno* shall be set to indicate the error.

38214 **ERRORS**38215 Refer to *fputc()*.38216 **EXAMPLES**38217 **Printing to Standard Output**

38218 The following example gets the current time, converts it to a string using *localtime()* and
 38219 *asctime()*, and prints it to standard output using *puts()*. It then prints the number of minutes to
 38220 an event for which it is waiting.

```
38221 #include <time.h>
38222 #include <stdio.h>
38223 ...
38224 time_t now;
38225 int minutes_to_event;
38226 ...
38227 time(&now);
38228 printf("The time is ");
38229 puts(asctime(localtime(&now)));
38230 printf("There are %d minutes to the event.\n",
38231     minutes_to_event);
38232 ...
```

38233 **APPLICATION USAGE**38234 The *puts()* function appends a <newline>, while *fputs()* does not.38235 **RATIONALE**

38236 None.

puts()

38237

FUTURE DIRECTIONS

38238

None.

38239

SEE ALSO

38240

fopen(), *fputs()*, *putc()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>

38241

CHANGE HISTORY

38242

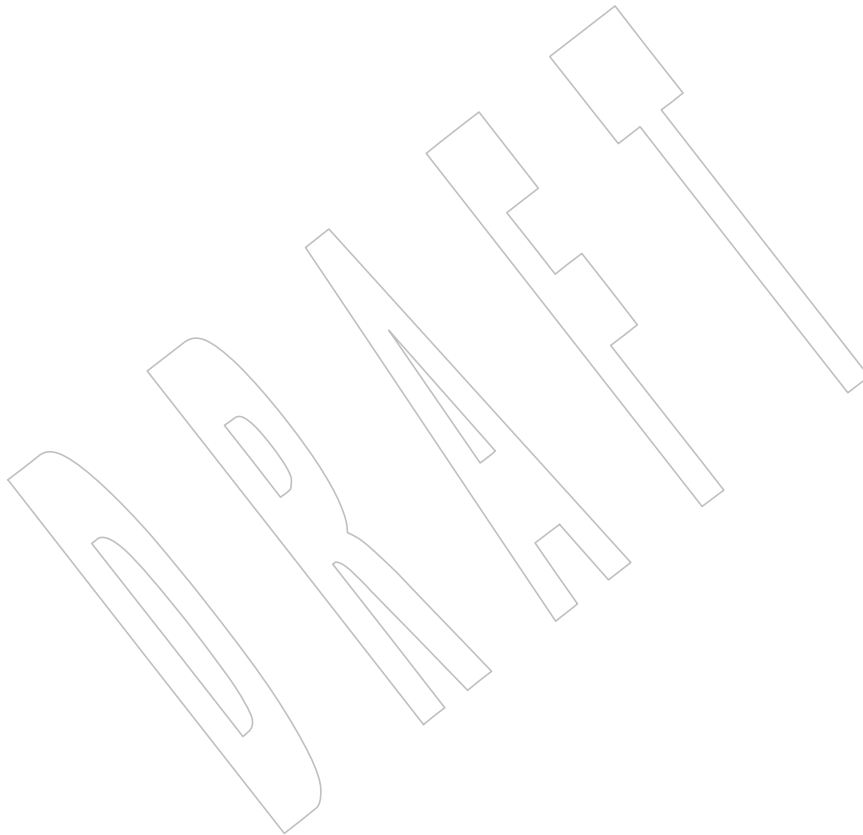
First released in Issue 1. Derived from Issue 1 of the SVID.

38243

Issue 6

38244

Extensions beyond the ISO C standard are marked.



38245 **NAME**
38246 pututxline — put an entry into the user accounting database

38247 **SYNOPSIS**

38248 XSI `#include <utmpx.h>`
38249 `struct utmpx *pututxline(const struct utmpx *utmpx);`

38250 **DESCRIPTION**

38251 Refer to *endutxent()*.

38252 **NAME**

38253 putwc — put a wide character on a stream

38254 **SYNOPSIS**

38255 #include <stdio.h>

38256 #include <wchar.h>

38257 wint_t putwc(wchar_t *wc*, FILE **stream*);38258 **DESCRIPTION**

38259 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 38260 conflict between the requirements described here and the ISO C standard is unintentional. This
 38261 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

38262 The *putwc()* function shall be equivalent to *fputwc()*, except that if it is implemented as a macro
 38263 it may evaluate *stream* more than once, so the argument should never be an expression with side
 38264 effects.

38265 **RETURN VALUE**38266 Refer to *fputwc()*.38267 **ERRORS**38268 Refer to *fputwc()*.38269 **EXAMPLES**

38270 None.

38271 **APPLICATION USAGE**

38272 Since it may be implemented as a macro, *putwc()* may treat a *stream* argument with side effects
 38273 incorrectly. In particular, *putwc(wc,*f++)* need not work correctly. Therefore, use of this function
 38274 is not recommended; *fputwc()* should be used instead.

38275 **RATIONALE**

38276 None.

38277 **FUTURE DIRECTIONS**

38278 None.

38279 **SEE ALSO**38280 *fputwc()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>, <wchar.h>38281 **CHANGE HISTORY**

38282 First released as a World-wide Portability Interface in Issue 4.

38283 **Issue 5**

38284 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument *wc*
 38285 is changed from **wint_t** to **wchar_t**.

38286 The Optional Header (OH) marking is removed from <stdio.h>.

38287 **NAME**
 38288 putwchar — put a wide character on a stdout stream

38289 **SYNOPSIS**
 38290 #include <wchar.h>

38291 wint_t putwchar(wchar_t wc);

38292 **DESCRIPTION**

38293 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 38294 conflict between the requirements described here and the ISO C standard is unintentional. This
 38295 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

38296 The function call *putwchar(wc)* shall be equivalent to *putwc(wc,stdout)*.

38297 **RETURN VALUE**

38298 Refer to *fputwc()*.

38299 **ERRORS**

38300 Refer to *fputwc()*.

38301 **EXAMPLES**

38302 None.

38303 **APPLICATION USAGE**

38304 None.

38305 **RATIONALE**

38306 None.

38307 **FUTURE DIRECTIONS**

38308 None.

38309 **SEE ALSO**

38310 *fputwc()*, *putwc()*, the Base Definitions volume of IEEE Std 1003.1-200x, <wchar.h>

38311 **CHANGE HISTORY**

38312 First released in Issue 4.

38313 **Issue 5**

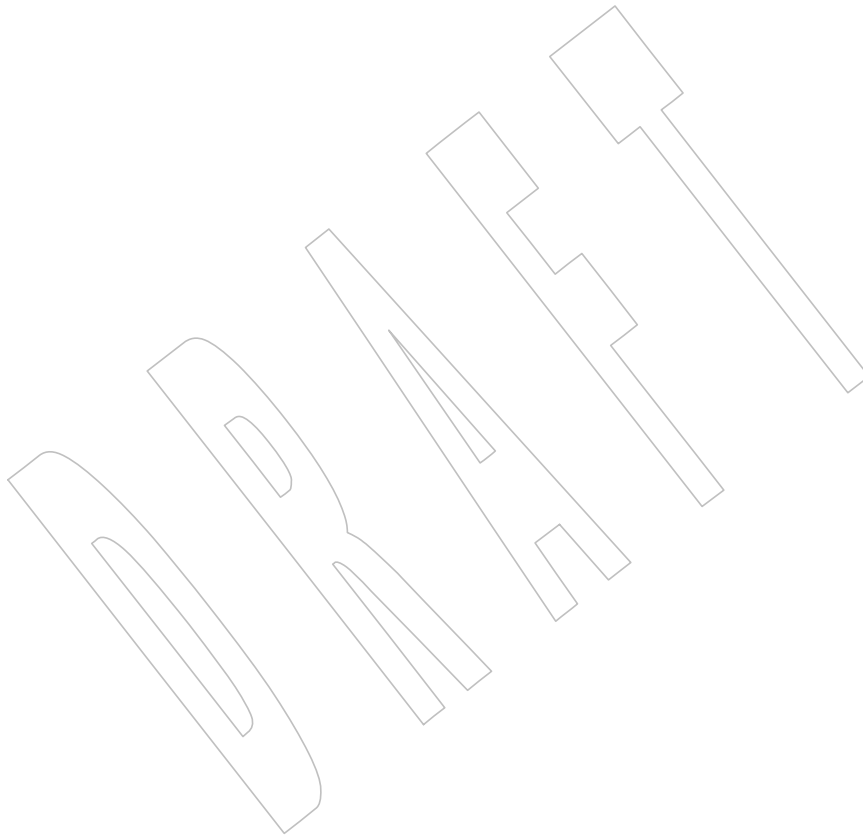
38314 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument *wc*
 38315 is changed from **wint_t** to **wchar_t**.

38316 **NAME**
38317 pwrite — write on a file

38318 **SYNOPSIS**
38319 #include <unistd.h>

38320 ssize_t pwrite(int *fd*, const void **buf*, size_t *nbyte*,
38321 off_t *offset*);

38322 **DESCRIPTION**
38323 Refer to *write()*.



38324 **NAME**

38325 qsort — sort a table of data

38326 **SYNOPSIS**

38327 #include <stdlib.h>

38328 void qsort(void *base, size_t nel, size_t width,
38329 int (*compar)(const void *, const void *));38330 **DESCRIPTION**38331 CX The functionality described on this reference page is aligned with the ISO C standard. Any
38332 conflict between the requirements described here and the ISO C standard is unintentional. This
38333 volume of IEEE Std 1003.1-200x defers to the ISO C standard.38334 The *qsort()* function shall sort an array of *nel* objects, the initial element of which is pointed to by
38335 *base*. The size of each object, in bytes, is specified by the *width* argument. If the *nel* argument has
38336 the value zero, the comparison function pointed to by *compar* shall not be called and no
38337 rearrangement shall take place.38338 The application shall ensure that the comparison function pointed to by *compar* does not alter the
38339 contents of the array. The implementation may reorder elements of the array between calls to the
38340 comparison function, but shall not alter the contents of any individual element.38341 When the same objects (consisting of *width* bytes, irrespective of their current positions in the
38342 array) are passed more than once to the comparison function, the results shall be consistent with
38343 one another. That is, they shall define a total ordering on the array.38344 The contents of the array shall be sorted in ascending order according to a comparison function.
38345 The *compar* argument is a pointer to the comparison function, which is called with two
38346 arguments that point to the elements being compared. The application shall ensure that the
38347 function returns an integer less than, equal to, or greater than 0, if the first argument is
38348 considered respectively less than, equal to, or greater than the second. If two members compare
38349 as equal, their order in the sorted array is unspecified.38350 **RETURN VALUE**38351 The *qsort()* function shall not return a value.38352 **ERRORS**

38353 No errors are defined.

38354 **EXAMPLES**

38355 None.

38356 **APPLICATION USAGE**38357 The comparison function need not compare every byte, so arbitrary data may be contained in
38358 the elements in addition to the values being compared.38359 **RATIONALE**38360 The requirement that each argument (hereafter referred to as *p*) to the comparison function is a
38361 pointer to elements of the array implies that for every call, for each argument separately, all of
38362 the following expressions are non-zero:38363 ((char *)p - (char *)base) % width == 0
38364 (char *)p >= (char *)base
38365 (char *)p < (char *)base + nel * width

38366

FUTURE DIRECTIONS

38367

None.

38368

SEE ALSO

38369

alphasort(), the Base Definitions volume of IEEE Std 1003.1-200x, <stdlib.h>

38370

CHANGE HISTORY

38371

First released in Issue 1. Derived from Issue 1 of the SVID.

38372

Issue 6

38373

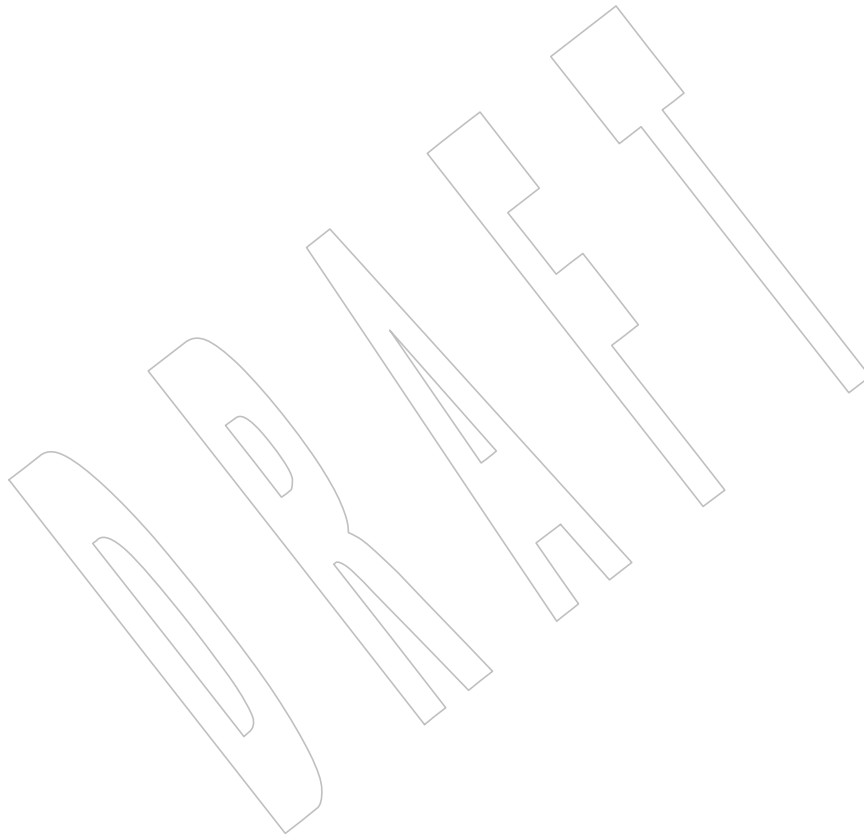
The normative text is updated to avoid use of the term “must” for application requirements.

38374

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/49 is applied, adding the last sentence to the first non-shaded paragraph in the DESCRIPTION, and the following two paragraphs. The RATIONALE is also updated. These changes are for alignment with the ISO C standard.

38375

38376



38377 **NAME**
 38378 `raise` — send a signal to the executing process

38379 **SYNOPSIS**
 38380 `#include <signal.h>`
 38381 `int raise(int sig);`

38382 **DESCRIPTION**
 38383 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 38384 conflict between the requirements described here and the ISO C standard is unintentional. This
 38385 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

38386 CX The `raise()` function shall send the signal `sig` to the executing thread or process. If a signal
 38387 handler is called, the `raise()` function shall not return until after the signal handler does.

38388 CX The effect of the `raise()` function shall be equivalent to calling:
 38389 `pthread_kill(pthread_self(), sig);`

38390 **RETURN VALUE**
 38391 CX Upon successful completion, 0 shall be returned. Otherwise, a non-zero value shall be returned
 38392 and `errno` shall be set to indicate the error.

38393 **ERRORS**
 38394 The `raise()` function shall fail if:

38395 CX [EINVAL] The value of the `sig` argument is an invalid signal number.

38396 **EXAMPLES**
 38397 None.

38398 **APPLICATION USAGE**
 38399 None.

38400 **RATIONALE**
 38401 The term “thread” is an extension to the ISO C standard.

38402 **FUTURE DIRECTIONS**
 38403 None.

38404 **SEE ALSO**
 38405 `kill()`, `sigaction()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<signal.h>`,
 38406 `<sys/types.h>`

38407 **CHANGE HISTORY**
 38408 First released in Issue 4. Derived from the ANSI C standard.

38409 **Issue 5**
 38410 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

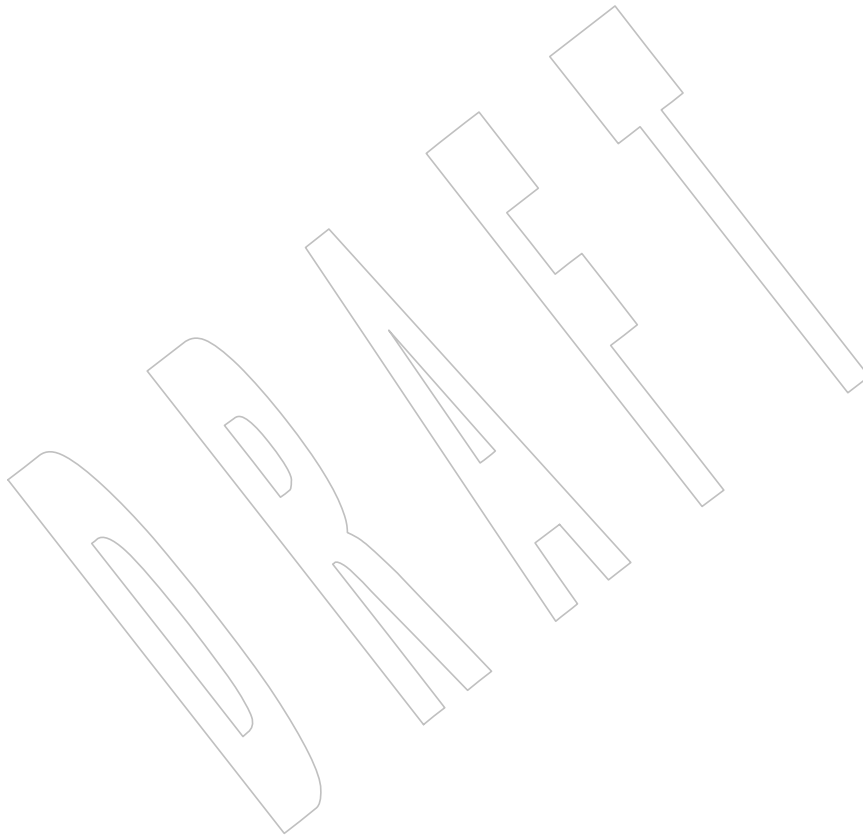
38411 **Issue 6**
 38412 Extensions beyond the ISO C standard are marked.

38413 The following new requirements on POSIX implementations derive from alignment with the
 38414 Single UNIX Specification:

- 38415 • In the RETURN VALUE section, the requirement to set `errno` on error is added.

raise()

- 38416
- The [EINVAL] error condition is added.
- 38417 **Issue 7**
- 38418 Functionality relating to the Threads option is moved to the Base.



38419 **NAME**
 38420 rand, rand_r, srand — pseudo-random number generator

38421 **SYNOPSIS**

38422 #include <stdlib.h>
 38423 int rand(void);
 38424 CX int rand_r(unsigned *seed);
 38425 void srand(unsigned seed);

38426 **DESCRIPTION**

38427 CX For *rand()* and *srand()*: The functionality described on this reference page is aligned with the
 38428 ISO C standard. Any conflict between the requirements described here and the ISO C standard is
 38429 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

38430 The *rand()* function shall compute a sequence of pseudo-random integers in the range
 38431 XSI [0,{RAND_MAX}] with a period of at least 2^{32} .

38432 CX The *rand()* function need not be thread-safe. A function that is not required to be thread-safe is
 38433 not required to be reentrant.

38434 The *rand_r()* function shall compute a sequence of pseudo-random integers in the range
 38435 [0,{RAND_MAX}]. (The value of the {RAND_MAX} macro shall be at least 32767.)

38436 If *rand_r()* is called with the same initial value for the object pointed to by *seed* and that object is
 38437 not modified between successive returns and calls to *rand_r()*, the same sequence shall be
 38438 generated.

38439 The *srand()* function uses the argument as a seed for a new sequence of pseudo-random
 38440 numbers to be returned by subsequent calls to *rand()*. If *srand()* is then called with the same
 38441 seed value, the sequence of pseudo-random numbers shall be repeated. If *rand()* is called before
 38442 any calls to *srand()* are made, the same sequence shall be generated as when *srand()* is first
 38443 called with a seed value of 1.

38444 The implementation shall behave as if no function defined in this volume of
 38445 IEEE Std 1003.1-200x calls *rand()* or *srand()*.

38446 **RETURN VALUE**

38447 The *rand()* function shall return the next pseudo-random number in the sequence.

38448 CX The *rand_r()* function shall return a pseudo-random integer.

38449 The *srand()* function shall not return a value.

38450 **ERRORS**

38451 No errors are defined.

38452 **EXAMPLES**

38453 **Generating a Pseudo-Random Number Sequence**

38454 The following example demonstrates how to generate a sequence of pseudo-random numbers.

```
38455 #include <stdio.h>
38456 #include <stdlib.h>
38457 ...
38458     long count, i;
38459     char *keystri;
38460     int elementlen, len;
38461     char c;
```



```

38462     ...
38463     /* Initial random number generator. */
38464     srand(1);
38465
38466     /* Create keys using only lowercase characters */
38467     len = 0;
38468     for (i=0; i<count; i++) {
38469         while (len < elementlen) {
38470             c = (char) (rand() % 128);
38471             if (islower(c))
38472                 keystr[len++] = c;
38473         }
38474         keystr[len] = '\0';
38475         printf("%s Element%0*ld\n", keystr, elementlen, i);
38476         len = 0;
38477     }

```

38477 **Generating the Same Sequence on Different Machines**

38478 The following code defines a pair of functions that could be incorporated into applications
38479 wishing to ensure that the same sequence of numbers is generated across different machines.

```

38480 static unsigned long next = 1;
38481 int myrand(void) /* RAND_MAX assumed to be 32767. */
38482 {
38483     next = next * 1103515245 + 12345;
38484     return((unsigned)(next/65536) % 32768);
38485 }
38486 void mysrand(unsigned seed)
38487 {
38488     next = seed;
38489 }

```

38490 **APPLICATION USAGE**

38491 The *drand48()* function provides a much more elaborate random number generator.

38492 The limitations on the amount of state that can be carried between one function call and another
38493 mean the *rand_r()* function can never be implemented in a way which satisfies all of the
38494 requirements on a pseudo-random number generator. Therefore this function should be avoided
38495 whenever non-trivial requirements (including safety) have to be fulfilled.

38496 **RATIONALE**

38497 The ISO C standard *rand()* and *srand()* functions allow per-process pseudo-random streams
38498 shared by all threads. Those two functions need not change, but there has to be mutual-
38499 exclusion that prevents interference between two threads concurrently accessing the random
38500 number generator.

38501 With regard to *rand()*, there are two different behaviors that may be wanted in a multi-threaded
38502 program:

- 38503 1. A single per-process sequence of pseudo-random numbers that is shared by all threads
38504 that call *rand()*
- 38505 2. A different sequence of pseudo-random numbers for each thread that calls *rand()*

38506 This is provided by the modified thread-safe function based on whether the seed value is global
38507 to the entire process or local to each thread.

38508 This does not address the known deficiencies of the *rand()* function implementations, which
38509 have been approached by maintaining more state. In effect, this specifies new thread-safe forms
38510 of a deficient function.

FUTURE DIRECTIONS

38511 None.
38512

SEE ALSO

38513 *drand48()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdlib.h>`
38514

CHANGE HISTORY

38515 First released in Issue 1. Derived from Issue 1 of the SVID.
38516

Issue 5

38517 The *rand_r()* function is included for alignment with the POSIX Threads Extension.
38518

38519 A note indicating that the *rand()* function need not be reentrant is added to the DESCRIPTION.

Issue 6

38520 Extensions beyond the ISO C standard are marked.
38521

38522 The *rand_r()* function is marked as part of the Thread-Safe Functions option.

Issue 7

38523 The *rand_r()* function is moved from the Thread-Safe Functions option to the Base.
38524

DRAFT

random()

38525 **NAME**
38526 random — generate pseudo-random number

SYNOPSIS

38527 XSI `#include <stdlib.h>`
38528 `long random(void);`

DESCRIPTION

38530 Refer to *initstate()*.
38531

NAME

pread, read — read from a file

SYNOPSIS

```
#include <unistd.h>

ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);
ssize_t read(int fildes, void *buf, size_t nbyte);
```

DESCRIPTION

The `read()` function shall attempt to read *nbyte* bytes from the file associated with the open file descriptor, *fildes*, into the buffer pointed to by *buf*. The behavior of multiple concurrent reads on the same pipe, FIFO, or terminal device is unspecified.

Before any action described below is taken, and if *nbyte* is zero, the `read()` function may detect and return errors as described below. In the absence of errors, or if error detection is not performed, the `read()` function shall return zero and have no other results.

On files that support seeking (for example, a regular file), the `read()` shall start at a position in the file given by the file offset associated with *fildes*. The file offset shall be incremented by the number of bytes actually read.

Files that do not support seeking—for example, terminals—always read from the current position. The value of a file offset associated with such a file is undefined.

No data transfer shall occur past the current end-of-file. If the starting position is at or after the end-of-file, 0 shall be returned. If the file refers to a device special file, the result of subsequent `read()` requests is implementation-defined.

If the value of *nbyte* is greater than {SSIZE_MAX}, the result is implementation-defined.

When attempting to read from an empty pipe or FIFO:

- If no process has the pipe open for writing, `read()` shall return 0 to indicate end-of-file.
- If some process has the pipe open for writing and O_NONBLOCK is set, `read()` shall return -1 and set *errno* to [EAGAIN].
- If some process has the pipe open for writing and O_NONBLOCK is clear, `read()` shall block the calling thread until some data is written or the pipe is closed by all processes that had the pipe open for writing.

When attempting to read a file (other than a pipe or FIFO) that supports non-blocking reads and has no data currently available:

- If O_NONBLOCK is set, `read()` shall return -1 and set *errno* to [EAGAIN].
- If O_NONBLOCK is clear, `read()` shall block the calling thread until some data becomes available.
- The use of the O_NONBLOCK flag has no effect if there is some data available.

The `read()` function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, `read()` shall return bytes with value 0. For example, `lseek()` allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data shall return bytes with value 0 until data is written into the gap.

Upon successful completion, where *nbyte* is greater than 0, `read()` shall mark for update the *st_atime* field of the file, and shall return the number of bytes read. This number shall never be

read()

38575		file is less than <i>nbyte</i> , if the <i>read()</i> request was interrupted by a signal, or if the file is a pipe or
38576		FIFO or special file and has fewer than <i>nbyte</i> bytes immediately available for reading. For
38577		example, a <i>read()</i> from a file associated with a terminal may return one typed line of data.
38578		If a <i>read()</i> is interrupted by a signal before it reads any data, it shall return -1 with <i>errno</i> set to
38579		[EINTR].
38580		If a <i>read()</i> is interrupted by a signal after it has successfully read some data, it shall return the
38581		number of bytes read.
38582		For regular files, no data transfer shall occur past the offset maximum established in the open
38583		file description associated with <i>filides</i> .
38584		If <i>filides</i> refers to a socket, <i>read()</i> shall be equivalent to <i>recv()</i> with no flags set.
38585	SIO	If the O_DSYNC and O_RSYNC bits have been set, read I/O operations on the file descriptor
38586		shall complete as defined by synchronized I/O data integrity completion. If the O_SYNC and
38587		O_RSYNC bits have been set, read I/O operations on the file descriptor shall complete as
38588		defined by synchronized I/O file integrity completion.
38589	SHM	If <i>filides</i> refers to a shared memory object, the result of the <i>read()</i> function is unspecified.
38590	TYM	If <i>filides</i> refers to a typed memory object, the result of the <i>read()</i> function is unspecified.
38591	OB XSR	A <i>read()</i> from a STREAMS file can read data in three different modes: <i>byte-stream</i> mode, <i>message-</i>
38592		<i>nondiscard</i> mode, and <i>message-discard</i> mode. The default shall be byte-stream mode. This can be
38593		changed using the I_SRDOPT <i>ioctl()</i> request, and can be tested with I_GRDOPT <i>ioctl()</i> . In byte-
38594		stream mode, <i>read()</i> shall retrieve data from the STREAM until as many bytes as were requested
38595		are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores
38596		message boundaries.
38597		In STREAMS message-nondiscard mode, <i>read()</i> shall retrieve data until as many bytes as were
38598		requested are transferred, or until a message boundary is reached. If <i>read()</i> does not retrieve all
38599		the data in a message, the remaining data shall be left on the STREAM, and can be retrieved by
38600		the next <i>read()</i> call. Message-discard mode also retrieves data until as many bytes as were
38601		requested are transferred, or a message boundary is reached. However, unread data remaining
38602		in a message after the <i>read()</i> returns shall be discarded, and shall not be available for a
38603		subsequent <i>read()</i> , <i>getmsg()</i> , or <i>getpmsg()</i> call.
38604		How <i>read()</i> handles zero-byte STREAMS messages is determined by the current read mode
38605		setting. In byte-stream mode, <i>read()</i> shall accept data until it has read <i>nbyte</i> bytes, or until there
38606		is no more data to read, or until a zero-byte message block is encountered. The <i>read()</i> function
38607		shall then return the number of bytes read, and place the zero-byte message back on the
38608		STREAM to be retrieved by the next <i>read()</i> , <i>getmsg()</i> , or <i>getpmsg()</i> . In message-nondiscard
38609		mode or message-discard mode, a zero-byte message shall return 0 and the message shall be
38610		removed from the STREAM. When a zero-byte message is read as the first message on a
38611		STREAM, the message shall be removed from the STREAM and 0 shall be returned, regardless
38612		of the read mode.
38613		A <i>read()</i> from a STREAMS file shall return the data in the message at the front of the STREAM
38614		head read queue, regardless of the priority band of the message.
38615		By default, STREAMs are in control-normal mode, in which a <i>read()</i> from a STREAMS file can
38616		only process messages that contain a data part but do not contain a control part. The <i>read()</i> shall
38617		fail if a message containing a control part is encountered at the STREAM head. This default
38618		action can be changed by placing the STREAM in either control-data mode or control-discard
38619		mode with the I_SRDOPT <i>ioctl()</i> command. In control-data mode, <i>read()</i> shall convert any
38620		control part to data and pass it to the application before passing any data part originally present
38621		in the same message. In control-discard mode, <i>read()</i> shall discard message control parts but
38622		return to the process any data part in the message.

In addition, *read()* shall fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of *errno* shall not reflect the result of *read()*, but reflect the prior error. If a hangup occurs on the STREAM being read, *read()* shall continue to operate normally until the STREAM head read queue is empty. Thereafter, it shall return 0.

The *pread()* function shall be equivalent to *read()*

38666	[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
38667	[ENOMEM]	Insufficient memory was available to fulfill the request.
38668	[ENXIO]	A request was made of a nonexistent device, or the request was outside the capabilities of the device.
38669		
38670		The <i>pread()</i> function shall fail, and the file pointer shall remain unchanged, if:
38671	[EINVAL]	The <i>offset</i> argument is invalid. The value is negative.
38672	[EOVERFLOW]	The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the file.
38673		
38674	[ENXIO]	A request was outside the capabilities of the device.
38675	[ESPIPE]	<i>fildev</i> is associated with a pipe or FIFO.

EXAMPLES**Reading Data into a Buffer**

The following example reads data from the file associated with the file descriptor *fd* into the buffer pointed to by *buf*.

```

38680 #include <sys/types.h>
38681 #include <unistd.h>
38682 ...
38683 char buf[20];
38684 size_t nbytes;
38685 ssize_t bytes_read;
38686 int fd;
38687 ...
38688 nbytes = sizeof(buf);
38689 bytes_read = read(fd, buf, nbytes);
38690 ...

```

APPLICATION USAGE

None.

RATIONALE

This volume of IEEE Std 1003.1-200x does not specify the value of the file offset after an error is returned; there are too many cases. For programming errors, such as [EBADF], the concept is meaningless since no file is involved. For errors that are detected immediately, such as [EAGAIN], clearly the pointer should not change. After an interrupt or hardware error, however, an updated value would be very useful and is the behavior of many implementations.

Note that a *read()* of zero bytes does not modify *st_atime*. A *read()* that requests more than zero bytes, but returns zero, shall modify *st_atime*.

Implementations are allowed, but not required, to perform error checking for *read()* requests of zero bytes.

38703

Input and Output

38704

38705

38706

38707

38708

38709

38710

38711

38712

The use of I/O with large byte counts has always presented problems. Ideas such as *lread()* and *lwrite()* (using and returning **longs**) were considered at one time. The current solution is to use abstract types on the ISO C standard function to *read()* and *write()*. The abstract types can be declared so that existing functions work, but can also be declared so that larger types can be represented in future implementations. It is presumed that whatever constraints limit the maximum range of **size_t** also limit portable I/O requests to the same range. This volume of IEEE Std 1003.1-200x also limits the range further by requiring that the byte count be limited so that a signed return value remains meaningful. Since the return type is also a (signed) abstract type, the byte count can be defined by the implementation to be larger than an **int** can hold.

38713

38714

38715

The standard developers considered adding atomicity requirements to a pipe or FIFO, but recognized that due to the nature of pipes and FIFOs there could be no guarantee of atomicity of reads of {PIPE_BUF} or any other size that would be an aid to applications portability.

38716

38717

38718

38719

38720

38721

This volume of IEEE Std 1003.1-200x requires that no action be taken for *read()* or *write()* when *nbyte* is zero. This is not intended to take precedence over detection of errors (such as invalid buffer pointers or file descriptors). This is consistent with the rest of this volume of IEEE Std 1003.1-200x, but the phrasing here could be misread to require detection of the zero case before any other errors. A value of zero is to be considered a correct value, for which the semantics are a no-op.

38722

38723

38724

38725

38726

38727

I/O is intended to be atomic to ordinary files and pipes and FIFOs. Atomic means that all the bytes from a single operation that started out together end up together, without interleaving from other I/O operations. It is a known attribute of terminals that this is not honored, and terminals are explicitly (and implicitly permanently) excepted, making the behavior unspecified. The behavior for other device types is also left unspecified, but the wording is intended to imply that future standards might choose to specify atomicity (or not).

38728

38729

38730

38731

38732

38733

There were recommendations to add format parameters to *read()* and *write()* in order to handle networked transfers among heterogeneous file system and base hardware types. Such a facility may be required for support by the OSI presentation of layer services. However, it was determined that this should correspond with similar C-language facilities, and that is beyond the scope of this volume of IEEE Std 1003.1-200x. The concept was suggested to the developers of the ISO C standard for their consideration as a possible area for future work.

38734

38735

38736

38737

38738

38739

In 4.3 BSD, a *read()* or *write()* that is interrupted by a signal before transferring any data does not by default return an [EINTR] error, but is restarted. In 4.2 BSD, 4.3 BSD, and the Eighth Edition, there is an additional function, *select()*, whose purpose is to pause until specified activity (data to read, space to write, and so on) is detected on specified file descriptors. It is common in applications written for those systems for *select()* to be used before *read()* in situations (such as keyboard input) where interruption of I/O due to a signal is desired.

38740

38741

The issue of which files or file types are interruptible is considered an implementation design issue. This is often affected primarily by hardware and reliability issues.

38742

38743

38744

There are no references to actions taken following an “unrecoverable error”. It is considered beyond the scope of this volume of IEEE Std 1003.1-200x to describe what happens in the case of hardware errors.

38745

38746

38747

38748

38749

38750

38751

38752

Previous versions of IEEE Std 1003.1-200x allowed two very different behaviors with regard to the handling of interrupts. In order to minimize the resulting confusion, it was decided that IEEE Std 1003.1-200x should support only one of these behaviors. Historical practice on AT&T-derived systems was to have *read()* and *write()* return **-1** and set *errno* to [EINTR] when interrupted after some, but not all, of the data requested had been transferred. However, the U.S. Department of Commerce FIPS 151-1 and FIPS 151-2 require the historical BSD behavior, in which *read()* and *write()* return the number of bytes actually transferred before the interrupt. If **-1** is returned when any data is transferred, it is difficult to recover from the error on a seekable

38753 device and impossible on a non-seeking device. Most new implementations support this
 38754 behavior. The behavior required by IEEE Std 1003.1-200x is to return the number of bytes
 38755 transferred.

38756 IEEE Std 1003.1-200x does not specify when an implementation that buffers *read()*s actually
 38757 moves the data into the user-supplied buffer, so an implementation may choose to do this at the
 38758 latest possible moment. Therefore, an interrupt arriving earlier may not cause *read()* to return a
 38759 partial byte count, but rather to return -1 and set *errno* to [EINTR].

38760 Consideration was also given to combining the two previous options, and setting *errno* to
 38761 [EINTR] while returning a short count. However, not only is there no existing practice that
 38762 implements this, it is also contradictory to the idea that when *errno* is set, the function
 38763 responsible shall return -1 .

38764 FUTURE DIRECTIONS

38765 None.

38766 SEE ALSO

38767 *fcntl()*, *ioctl()*, *lseek()*, *open()*, *pipe()*, *readv()*, the Base Definitions volume of
 38768 IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface, `<stropts.h>`, `<sys/uio.h>`,
 38769 `<unistd.h>`

38770 CHANGE HISTORY

38771 First released in Issue 1. Derived from Issue 1 of the SVID.

38772 Issue 5

38773 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
 38774 Threads Extension.

38775 Large File Summit extensions are added.

38776 The *pread()* function is added.

38777 Issue 6

38778 The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are
 38779 marked as part of the XSI STREAMS Option Group.

38780 The following new requirements on POSIX implementations derive from alignment with the
 38781 Single UNIX Specification:

- 38782 • The DESCRIPTION now states that if *read()* is interrupted by a signal after it has
 38783 successfully read some data, it returns the number of bytes read. In Issue 3, it was optional
 38784 whether *read()* returned the number of bytes read, or whether it returned -1 with *errno* set
 38785 to [EINTR]. This is a FIPS requirement.
- 38786 • In the DESCRIPTION, text is added to indicate that for regular files, no data transfer
 38787 occurs past the offset maximum established in the open file description associated with
 38788 *files*. This change is to support large files.
- 38789 • The [EOVERFLOW] mandatory error condition is added.
- 38790 • The [ENXIO] optional error condition is added.

38791 Text referring to sockets is added to the DESCRIPTION.

38792 The following changes were made to align with the IEEE P1003.1a draft standard:

- 38793 • The effect of reading zero bytes is clarified.

38794 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that
 38795 *read()* results are unspecified for typed memory objects.

38796 New RATIONALE is added to explain the atomicity requirements for input and output
 38797 operations.

38798 The following error conditions are added for operations on sockets: [EAGAIN],
38799 [ECONNRESET], [ENOTCONN], and [ETIMEDOUT].

38800 The [EIO] error is made optional.

38801 The following error conditions are added for operations on sockets: [ENOBUFS] and
38802 [ENOMEM].

38803 The *readv()* function is split out into a separate reference page.

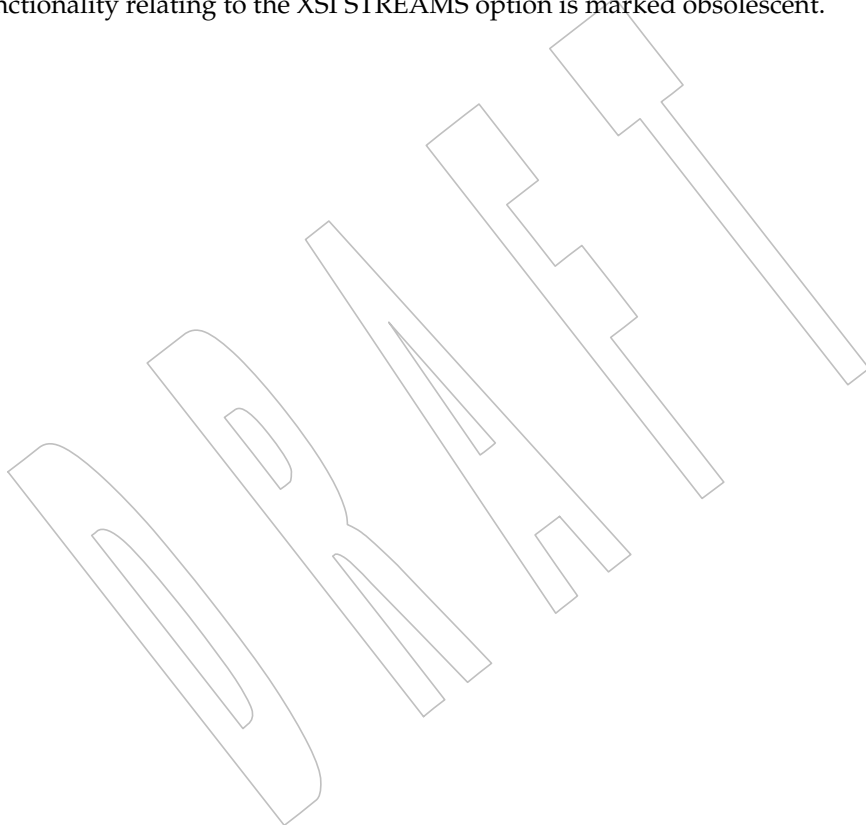
38804 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/108 is applied, updating the [EAGAIN]
38805 error in the ERRORS section from “the process would be delayed” to “the thread would be
38806 delayed”.

38807 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/109 is applied, making an editorial
38808 correction in the RATIONALE section.

38809 **Issue 7**

38810 The *pread()* function is moved from the XSI option to the Base.

38811 Functionality relating to the XSI STREAMS option is marked obsolescent.



38812 NAME

38813 readdir, readdir_r — read a directory

38814 SYNOPSIS

38815 #include <dirent.h>

38816 struct dirent *readdir(DIR *dirp);

38817 int readdir_r(DIR *restrict dirp, struct dirent *restrict entry,

38818 struct dirent **restrict result);

38819 DESCRIPTION

38820 The type **DIR**, which is defined in the **<dirent.h>** header, represents a *directory stream*, which is
 38821 an ordered sequence of all the directory entries in a particular directory. Directory entries
 38822 represent files; files may be removed from a directory or added to a directory asynchronously to
 38823 the operation of *readdir()*.

38824 The *readdir()* function shall return a pointer to a structure representing the directory entry at the
 38825 current position in the directory stream specified by the argument *dirp*, and position the
 38826 directory stream at the next entry. It shall return a null pointer upon reaching the end of the
 38827 directory stream. The structure **dirent** defined in the **<dirent.h>** header describes a directory
 38828 entry.

38829 The *readdir()* function shall not return directory entries containing empty names. If entries for
 38830 dot or dot-dot exist, one entry shall be returned for dot and one entry shall be returned for dot-
 38831 dot; otherwise, they shall not be returned.

38832 The pointer returned by *readdir()* points to data which may be overwritten by another call to
 38833 *readdir()* on the same directory stream. This data is not overwritten by another call to *readdir()*
 38834 on a different directory stream.

38835 If a file is removed from or added to the directory after the most recent call to *opendir()* or
 38836 *rewinddir()*, whether a subsequent call to *readdir()* returns an entry for that file is unspecified.

38837 The *readdir()* function may buffer several directory entries per actual read operation; *readdir()*
 38838 shall mark for update the *st_atime* field of the directory each time the directory is actually read.

38839 After a call to *fork()*, either the parent or child (but not both) may continue processing the
 38840 directory stream using *readdir()*, *rewinddir()*, or *seekdir()*. If both the parent and child processes
 38841 use these functions, the result is undefined.

38842 If the entry names a symbolic link, the value of the *d_ino* member is unspecified.

38843 The *readdir()* function need not be thread-safe. A function that is not required to be thread-safe is
 38844 not required to be reentrant.

38845 The *readdir_r()* function shall initialize the **dirent** structure referenced by *entry* to represent the
 38846 directory entry at the current position in the directory stream referred to by *dirp*, store a pointer
 38847 to this structure at the location referenced by *result*, and position the directory stream at the next
 38848 entry.

38849 The storage pointed to by *entry* shall be large enough for a **dirent** with an array of **char** *d_name*
 38850 members containing at least {NAME_MAX}+1 elements.

38851 Upon successful return, the pointer returned at **result* shall have the same value as the argument
 38852 *entry*. Upon reaching the end of the directory stream, this pointer shall have the value NULL.

38853 The *readdir_r()* function shall not return directory entries containing empty names.

38854 If a file is removed from or added to the directory after the most recent call to *opendir()* or
 38855 *rewinddir()*, whether a subsequent call to *readdir_r()* returns an entry for that file is unspecified.

38856 The *readdir_r()* function may buffer several directory entries per actual read operation; the
 38857 *readdir_r()* function shall mark for update the *st_atime* field of the directory each time the
 38858 directory is actually read.

38859 Applications wishing to check for error situations should set *errno* to 0 before calling *readdir()*. If
 38860 *errno* is set to non-zero on return, an error occurred.

RETURN VALUE

38861 Upon successful completion, *readdir()* shall return a pointer to an object of type **struct dirent**.
 38862 When an error is encountered, a null pointer shall be returned and *errno* shall be set to indicate
 38863 the error. When the end of the directory is encountered, a null pointer shall be returned and
 38864 *errno* is not changed.
 38865

38866 If successful, the *readdir_r()* function shall return zero; otherwise, an error number shall be
 38867 returned to indicate the error.

ERRORS

38868 The *readdir()* and *readdir_r()* functions shall fail if:

38870 [EOVERFLOW] One of the values in the structure to be returned cannot be represented
 38871 correctly.

38872 The *readdir()* and *readdir_r()* functions may fail if:

38873 [EBADF] The *dirp* argument does not refer to an open directory stream.

38874 [ENOENT] The current position of the directory stream is invalid.

EXAMPLES

38875 The following sample program searches the current directory for each of the arguments supplied
 38876 on the command line.
 38877

```

38878 #include <dirent.h>
38879 #include <errno.h>
38880 #include <stdio.h>
38881 #include <string.h>
38882 static void lookup(const char *arg)
38883 {
38884     DIR *dirp;
38885     struct dirent *dp;
38886     if ((dirp = opendir(".")) == NULL) {
38887         perror("couldn't open '.');
38888         return;
38889     }
38890     do {
38891         errno = 0;
38892         if ((dp = readdir(dirp)) != NULL) {
38893             if (strcmp(dp->d_name, arg) != 0)
38894                 continue;
38895             (void) printf("found %s\n", arg);
38896             (void) closedir(dirp);
38897             return;
38898         }
38899     } while (dp != NULL);
38900     if (errno != 0)
38901         perror("error reading directory");

```

```

38902         else
38903             (void) printf("failed to find %s\n", arg);
38904         (void) closedir(dirp);
38905         return;
38906     }
38907
38907 int main(int argc, char *argv[])
38908 {
38909     int i;
38910     for (i = 1; i < argc; i++)
38911         lookup(argv[i]);
38912     return (0);
38913 }

```

APPLICATION USAGE

38914 The *readdir()* function should be used in conjunction with *opendir()*, *closedir()*, and *rewinddir()* to
 38915 examine the contents of the directory.

38917 The *readdir_r()* function is thread-safe and shall return values in a user-supplied buffer instead
 38918 of possibly using a static data area that may be overwritten by each call.

RATIONALE

38919 The returned value of *readdir()* merely *represents* a directory entry. No equivalence should be
 38920 inferred.

38922 Historical implementations of *readdir()* obtain multiple directory entries on a single read
 38923 operation, which permits subsequent *readdir()* operations to operate from the buffered
 38924 information. Any wording that required each successful *readdir()* operation to mark the
 38925 directory *st_atime* field for update would disallow such historical performance-oriented
 38926 implementations.

38927 Since *readdir()* returns NULL when it detects an error and when the end of the directory is
 38928 encountered, an application that needs to tell the difference must set *errno* to zero before the call
 38929 and check it if NULL is returned. Since the function must not change *errno* in the second case
 38930 and must set it to a non-zero value in the first case, a zero *errno* after a call returning NULL
 38931 indicates end-of-directory; otherwise, an error.

38932 Routines to deal with this problem more directly were proposed:

```

38933 int derror (dirp)
38934 DIR *dirp;
38935
38935 void clearerr (dirp)
38936 DIR *dirp;

```

38937 The first would indicate whether an error had occurred, and the second would clear the error
 38938 indication. The simpler method involving *errno* was adopted instead by requiring that *readdir()*
 38939 not change *errno* when end-of-directory is encountered.

38940 An error or signal indicating that a directory has changed while open was considered but
 38941 rejected.

38942 The thread-safe version of the directory reading function returns values in a user-supplied buffer
 38943 instead of possibly using a static data area that may be overwritten by each call. Either the
 38944 {NAME_MAX} compile-time constant or the corresponding *pathconf()* option can be used to
 38945 determine the maximum sizes of returned pathnames.

38946 **FUTURE DIRECTIONS**

38947 None.

38948 **SEE ALSO**38949 *closedir()*, *dirfd()*, *exec*, *fdopendir()*, *fstatat()*, *rewinddir()*, *symlink()*, the Base Definitions volume
38950 of IEEE Std 1003.1-200x, **<dirent.h>**, **<sys/types.h>**38951 **CHANGE HISTORY**

38952 First released in Issue 2.

38953 **Issue 5**

38954 Large File Summit extensions are added.

38955 The *readdir_r()* function is included for alignment with the POSIX Threads Extension.38956 A note indicating that the *readdir()* function need not be reentrant is added to the
38957 DESCRIPTION.38958 **Issue 6**38959 The *readdir_r()* function is marked as part of the Thread-Safe Functions option.38960 The Open Group Corrigendum U026/7 is applied, correcting the prototype for *readdir_r()*.38961 The Open Group Corrigendum U026/8 is applied, clarifying the wording of the successful
38962 return for the *readdir_r()* function.38963 The following new requirements on POSIX implementations derive from alignment with the
38964 Single UNIX Specification:

- 38965 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
38966 required for conforming implementations of previous POSIX specifications, it was not
38967 required for UNIX applications.
- 38968 • A statement is added to the DESCRIPTION indicating the disposition of certain fields in
38969 **struct dirent** when an entry refers to a symbolic link.
- 38970 • The [Eoverflow] mandatory error condition is added. This change is to support large
38971 files.
- 38972 • The [ENOENT] optional error condition is added.

38973 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
38974 its avoidance of possibly using a static data area.38975 The **restrict** keyword is added to the *readdir_r()* prototype for alignment with the
38976 ISO/IEC 9899:1999 standard.38977 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/50 is applied, replacing the EXAMPLES
38978 section with a new example.38979 **Issue 7**

38980 Austin Group Interpretation 1003.1-2001 #059 is applied, updating the ERRORS section.

38981 The *readdir_r()* function is moved from the Thread-Safe Functions option to the Base.

38982 **NAME**

38983 readlink, readlinkat — read the contents of a symbolic link relative to a directory file descriptor

38984 **SYNOPSIS**

```
38985 #include <unistd.h>
38986
38986 ssize_t readlink(const char *restrict path, char *restrict buf,
38987                 size_t bufsize);
38988 ssize_t readlinkat(int fd, const char *path, char *buf,
38989                  size_t bufsize);
```

38990 **DESCRIPTION**

38991 The *readlink()* function shall place the contents of the symbolic link referred to by *path* in the
 38992 buffer *buf* which has size *bufsize*. If the number of bytes in the symbolic link is less than *bufsize*,
 38993 the contents of the remainder of *buf* are unspecified. If the *buf* argument is not large enough to
 38994 contain the link content, the first *bufsize* bytes shall be placed in *buf*.

38995 If the value of *bufsize* is greater than {SSIZE_MAX}, the result is implementation-defined.

38996 The *readlinkat()* function shall be equivalent to the *readlink()* function except in the case where
 38997 *path* specifies a relative path. In this case the symbolic link whose content is read is relative to the
 38998 directory associated with the file descriptor *fd* instead of the current working directory. It is
 38999 unspecified whether directory searches are permitted based on whether the file was opened
 39000 with search permission or on the current permissions of the directory underlying the file
 39001 descriptor.

39002 If *readlinkat()* is passed the special value AT_FDCWD in the *fd* parameter, the current working
 39003 directory is used and the behavior shall be identical to a call to *readlink()*.

39004 **RETURN VALUE**

39005 Upon successful completion, *readlink()* shall return the count of bytes placed in the buffer.
 39006 Otherwise, it shall return a value of -1 , leave the buffer unchanged, and set *errno* to indicate the
 39007 error.

39008 Upon successful completion, the *readlinkat()* function shall return 0. Otherwise, it shall return -1
 39009 and set *errno* to indicate the error.

39010 **ERRORS**

39011 These functions shall fail if:

- | | | |
|-------|----------------|---|
| 39012 | [EACCES] | Search permission is denied for a component of the path prefix of <i>path</i> . |
| 39013 | [EINVAL] | The <i>path</i> argument names a file that is not a symbolic link. |
| 39014 | [EIO] | An I/O error occurred while reading from the file system. |
| 39015 | [ELOOP] | A loop exists in symbolic links encountered during resolution of the <i>path</i>
39016 argument. |
| 39017 | [ENAMETOOLONG] | The length of the <i>path</i> argument exceeds {PATH_MAX} or a pathname
39018 component is longer than {NAME_MAX}. |
| 39020 | [ENOENT] | A component of <i>path</i> does not name an existing file or <i>path</i> is an empty string. |
| 39021 | [ENOTDIR] | A component of the path prefix is not a directory. |

39022 The *readlinkat()* function shall fail if:

- 39023 [EBADF] The *path* argument does not specify an absolute path and the *fd* argument is
39024 neither AT_FDCWD nor a valid file descriptor open for searching.
- 39025 These functions may fail if:
- 39026 [EACCES] Read permission is denied for the directory.
- 39027 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
39028 resolution of the *path* argument.
- 39029 [ENAMETOOLONG]
39030 As a result of encountering a symbolic link in resolution of the *path* argument,
39031 the length of the substituted pathname string exceeded {PATH_MAX}.
- 39032 The *readlinkat()* function may fail if:
- 39033 [ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a
39034 file descriptor associated with a directory.

EXAMPLES**Reading the Name of a Symbolic Link**

The following example shows how to read the name of a symbolic link named `/modules/pass1`.

```
39038 #include <unistd.h>
39039 char buf[1024];
39040 ssize_t len;
39041 ...
39042 if ((len = readlink("/modules/pass1", buf, sizeof(buf)-1)) != -1)
39043     buf[len] = '\0';
```

APPLICATION USAGE

Conforming applications should not assume that the returned contents of the symbolic link are null-terminated.

RATIONALE

Since IEEE Std 1003.1-200x does not require any association of file times with symbolic links, there is no requirement that file times be updated by *readlink()*. The type associated with *bufsiz* is a **size_t** in order to be consistent with both the ISO C standard and the definition of *read()*. The behavior specified for *readlink()* when *bufsiz* is zero represents historical practice. For this case, the standard developers considered a change whereby *readlink()* would return the number of non-null bytes contained in the symbolic link with the buffer *buf* remaining unchanged; however, since the **stat** structure member *st_size* value can be used to determine the size of buffer necessary to contain the contents of the symbolic link as returned by *readlink()*, this proposal was rejected, and the historical practice retained.

The purpose of the *readlinkat()* function is to read the content of symbolic links in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *readlink()*, resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *readlinkat()* function it can be guaranteed that the symbolic link read is located relative to the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

fstatat(), *symlink()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<unistd.h>`

39066
39067
39068
39069
39070
39071
39072
39073
39074
39075
39076
39077
39078
39079
39080
39081
39082
39083
39084
39085
39086

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The return type is changed to **ssize_t**, to align with the IEEE P1003.1a draft standard.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- This function is made mandatory.
- In this function it is possible for the return value to exceed the range of the type **ssize_t** (since **size_t** has a larger range of positive values than **ssize_t**). A sentence restricting the size of the **size_t** object is added to the description to resolve this conflict.

The following changes are made for alignment with the ISO POSIX-1:1996 standard:

- The FUTURE DIRECTIONS section is changed to None.

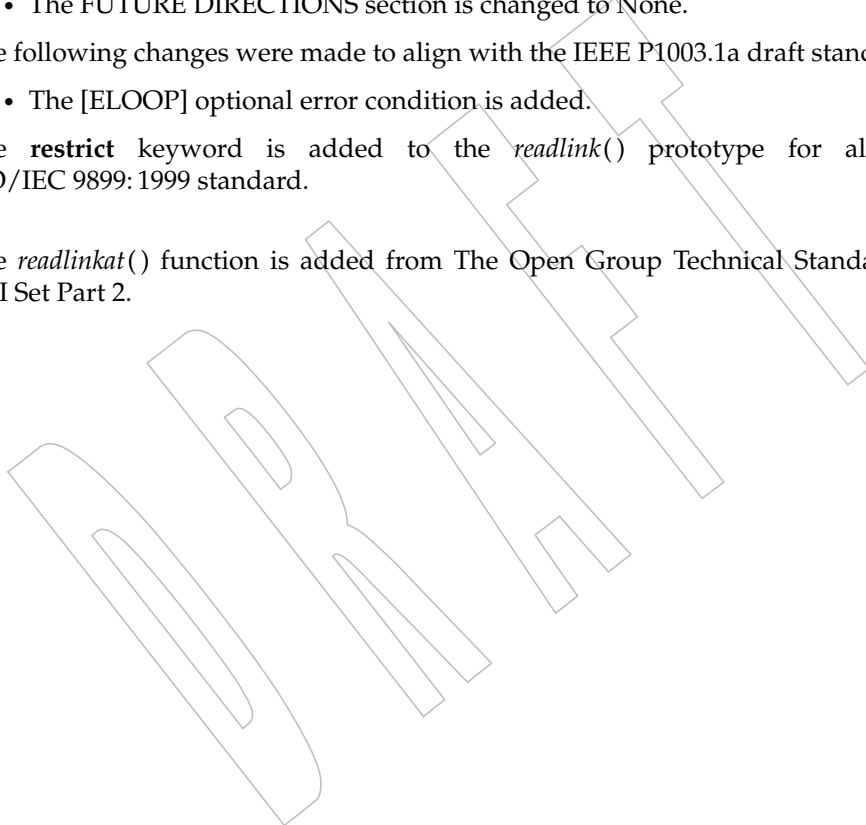
The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

The **restrict** keyword is added to the *readlink()* prototype for alignment with the ISO/IEC 9899:1999 standard.

Issue 7

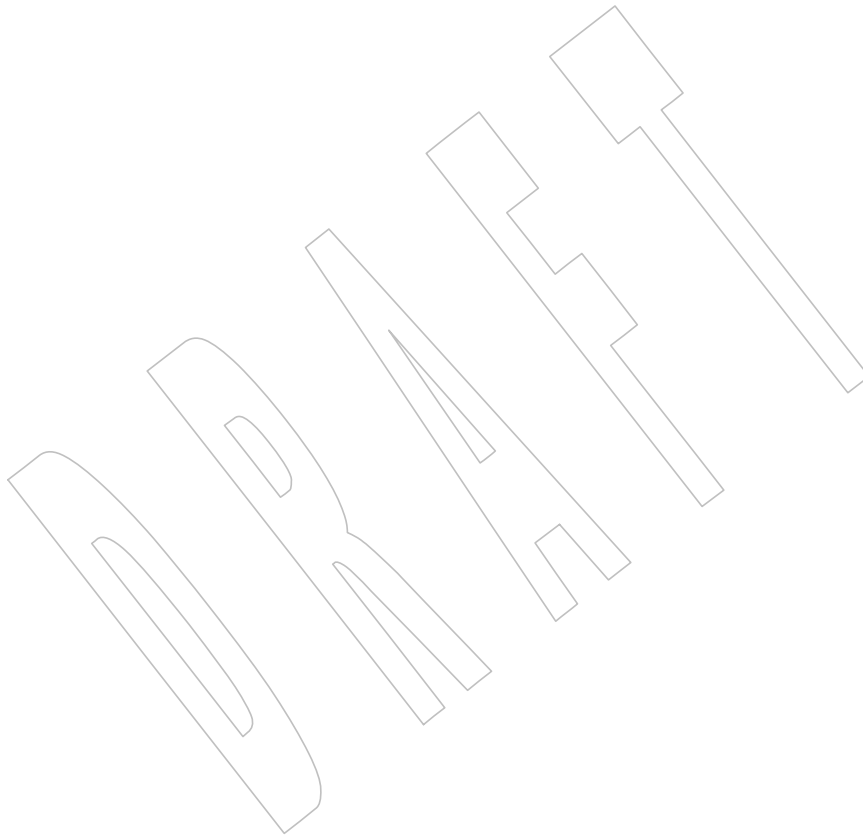
The *readlinkat()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.



39087 **NAME**
39088 readlinkat — read the contents of a symbolic link relative to a directory file descriptor

39089 **SYNOPSIS**
39090 #include <unistd.h>
39091 ssize_t readlinkat(int *fd*, const char **path*, char **buf*,
39092 size_t *bufsize*);

39093 **DESCRIPTION**
39094 Refer to *readlink()*.



39095 **NAME**

39096 readv — read a vector

39097 **SYNOPSIS**

```
39098 XSI #include <sys/uio.h>
39099 ssize_t readv(int fildev, const struct iovec *iov, int iovcnt);
```

39100 **DESCRIPTION**

39101 The *readv()* function shall be equivalent to *read()*, except as described below. The *readv()*
 39102 function shall place the input data into the *iovcnt* buffers specified by the members of the *iov*
 39103 array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt*-1]. The *iovcnt* argument is valid if greater than 0 and less than
 39104 or equal to {IOV_MAX}.

39105 Each *iovec* entry specifies the base address and length of an area in memory where data should
 39106 be placed. The *readv()* function shall always fill an area completely before proceeding to the
 39107 next.

39108 Upon successful completion, *readv()* shall mark for update the *st_atime* field of the file.

39109 **RETURN VALUE**39110 Refer to *read()*.39111 **ERRORS**39112 Refer to *read()*.39113 In addition, the *readv()* function shall fail if:39114 [EINVAL] The sum of the *iov_len* values in the *iov* array overflowed an **ssize_t**.39115 The *readv()* function may fail if:39116 [EINVAL] The *iovcnt* argument was less than or equal to 0, or greater than {IOV_MAX}.39117 **EXAMPLES**39118 **Reading Data into an Array**

39119 The following example reads data from the file associated with the file descriptor *fd* into the
 39120 buffers specified by members of the *iov* array.

```
39121 #include <sys/types.h>
39122 #include <sys/uio.h>
39123 #include <unistd.h>
39124 ...
39125 ssize_t bytes_read;
39126 int fd;
39127 char buf0[20];
39128 char buf1[30];
39129 char buf2[40];
39130 int iovcnt;
39131 struct iovec iov[3];
39132
39133 iov[0].iov_base = buf0;
39134 iov[0].iov_len = sizeof(buf0);
39135 iov[1].iov_base = buf1;
39136 iov[1].iov_len = sizeof(buf1);
39137 iov[2].iov_base = buf2;
```

```
39137     iov[2].iov_len = sizeof(buf2);
39138     ...
39139     iovcnt = sizeof(iov) / sizeof(struct iovec);
39140     bytes_read = readv(fd, iov, iovcnt);
39141     ...
```

APPLICATION USAGE

None.

RATIONALE

Refer to *read()*.

FUTURE DIRECTIONS

None.

SEE ALSO

read(), *writev()*, the Base Definitions volume of IEEE Std 1003.1-200x, <sys/uio.h>

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 6

Split out from the *read()* reference page.

DRAFT

39154 **NAME**39155 **realloc** — memory reallocator39156 **SYNOPSIS**

39157 #include <stdlib.h>

39158 void *realloc(void *ptr, size_t size);

39159 **DESCRIPTION**39160 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
39161 conflict between the requirements described here and the ISO C standard is unintentional. This
39162 volume of IEEE Std 1003.1-200x defers to the ISO C standard.39163 The *realloc()* function shall change the size of the memory object pointed to by *ptr* to the size
39164 specified by *size*. The contents of the object shall remain unchanged up to the lesser of the new
39165 and old sizes. If the new size of the memory object would require movement of the object, the
39166 space for the previous instantiation of the object is freed. If the new size is larger, the contents of
39167 the newly allocated portion of the object are unspecified. If *size* is 0 and *ptr* is not a null pointer,
39168 the object pointed to is freed. If the space cannot be allocated, the object shall remain unchanged.39169 If *ptr* is a null pointer, *realloc()* shall be equivalent to *malloc()* for the specified size.39170 If *ptr* does not match a pointer returned earlier by *calloc()*, *malloc()*, or *realloc()* or if the space has
39171 previously been deallocated by a call to *free()* or *realloc()*, the behavior is undefined.39172 The order and contiguity of storage allocated by successive calls to *realloc()* is unspecified. The
39173 pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to
39174 a pointer to any type of object and then used to access such an object in the space allocated (until
39175 the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object
39176 disjoint from any other object. The pointer returned shall point to the start (lowest byte address)
39177 of the allocated space. If the space cannot be allocated, a null pointer shall be returned.39178 **RETURN VALUE**39179 Upon successful completion with a size not equal to 0, *realloc()* shall return a pointer to the
39180 (possibly moved) allocated space. If *size* is 0, either a null pointer or a unique pointer that can be
39181 successfully passed to *free()* shall be returned. If there is not enough available memory, *realloc()*
39182 shall return a null pointer and set *errno* to [ENOMEM].39183 **ERRORS**39184 The *realloc()* function shall fail if:39185 **CX** [ENOMEM] Insufficient memory is available.39186 **EXAMPLES**

39187 None.

39188 **APPLICATION USAGE**

39189 None.

39190 **RATIONALE**

39191 None.

39192 **FUTURE DIRECTIONS**

39193 None.

39194
39195
39196
39197
39198
39199
39200
39201
39202
39203
39204

SEE ALSO

calloc(), *free()*, *malloc()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdlib.h>

CHANGE HISTORY

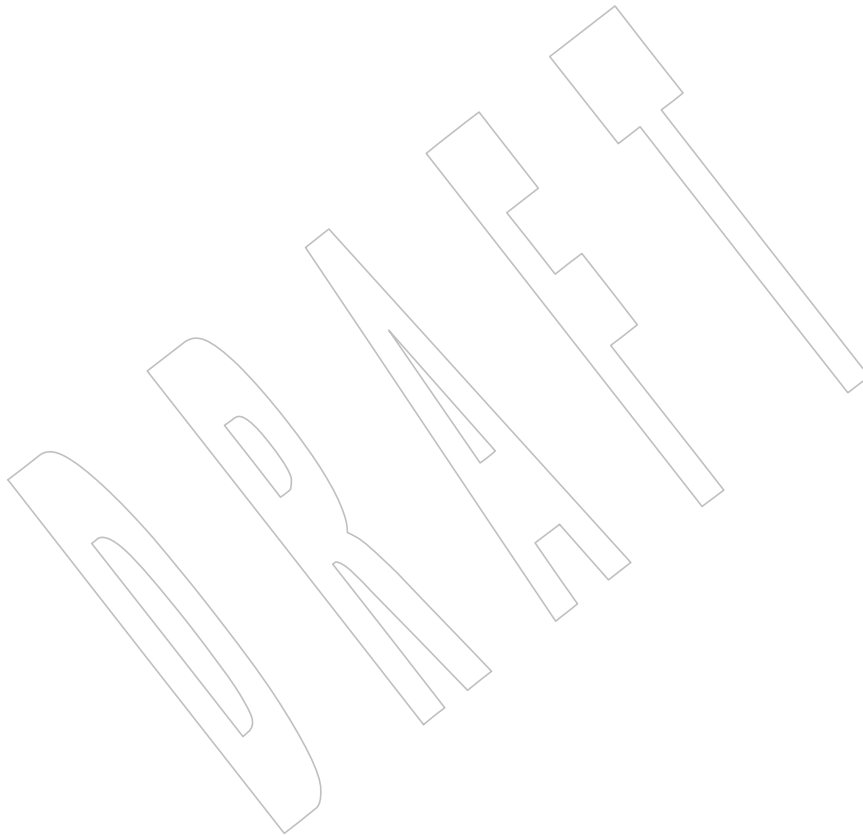
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, if there is not enough available memory, the setting of *errno* to [ENOMEM] is added.
- The [ENOMEM] error condition is added.



39205 **NAME**

39206 realpath — resolve a pathname

39207 **SYNOPSIS**

```
39208 XSI      #include <stdlib.h>
39209
39209 char *realpath(const char *restrict file_name,
39210               char *restrict resolved_name);
```

39211 **DESCRIPTION**

39212 The *realpath()* function shall derive, from the pathname pointed to by *file_name*, an absolute
 39213 pathname that names the same file, whose resolution does not involve '.', '..', or symbolic
 39214 links. The generated pathname shall be stored as a null-terminated string, up to a maximum of
 39215 {PATH_MAX} bytes, in the buffer pointed to by *resolved_name*.

39216 If *resolved_name* is a null pointer, the behavior of *realpath()* is implementation-defined.

39217 **RETURN VALUE**

39218 Upon successful completion, *realpath()* shall return a pointer to the resolved name. Otherwise,
 39219 *realpath()* shall return a null pointer and set *errno* to indicate the error, and the contents of the
 39220 buffer pointed to by *resolved_name* are undefined.

39221 **ERRORS**

39222 The *realpath()* function shall fail if:

- | | | |
|-------|----------------|--|
| 39223 | [EACCES] | Read or search permission was denied for a component of <i>file_name</i> . |
| 39224 | [EINVAL] | The <i>file_name</i> argument is a null pointer. |
| 39225 | [EIO] | An error occurred while reading from the file system. |
| 39226 | [ELOOP] | A loop exists in symbolic links encountered during resolution of the <i>file_name</i> |
| 39227 | | argument. |
| 39228 | [ENAMETOOLONG] | |
| 39229 | | The length of the <i>file_name</i> argument exceeds {PATH_MAX} or a pathname |
| 39230 | | component is longer than {NAME_MAX}. |
| 39231 | [ENOENT] | A component of <i>file_name</i> does not name an existing file or <i>file_name</i> points to |
| 39232 | | an empty string. |
| 39233 | [ENOTDIR] | A component of the path prefix is not a directory. |

39234 The *realpath()* function may fail if:

- | | | |
|-------|----------------|--|
| 39235 | [ELOOP] | More than {SYMLOOP_MAX} symbolic links were encountered during |
| 39236 | | resolution of the <i>file_name</i> argument. |
| 39237 | [ENAMETOOLONG] | |
| 39238 | | Pathname resolution of a symbolic link produced an intermediate result |
| 39239 | | whose length exceeds {PATH_MAX}. |
| 39240 | [ENOMEM] | Insufficient storage space is available. |

39241 **EXAMPLES**39242 **Generating an Absolute Pathname**

39243 The following example generates an absolute pathname for the file identified by the *symlinkpath*
 39244 argument. The generated pathname is stored in the *actualpath* array.

```
39245 #include <stdlib.h>
39246 ...
39247 char *symlinkpath = "/tmp/symlink/file";
39248 char actualpath [PATH_MAX+1];
39249 char *ptr;
39250 ptr = realpath(symlinkpath, actualpath);
```

39251 **APPLICATION USAGE**

39252 None.

39253 **RATIONALE**

39254 Since the maximum pathname length is arbitrary unless `{PATH_MAX}` is defined, an application
 39255 generally cannot supply a *resolved_name* buffer with size `{{PATH_MAX}+1}`.

39256 **FUTURE DIRECTIONS**

39257 In the future, passing a null pointer to *realpath()* for the *resolved_name* argument may be defined
 39258 to have *realpath()* allocate space for the generated pathname.

39259 **SEE ALSO**

39260 *getcwd()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdlib.h>`

39261 **CHANGE HISTORY**

39262 First released in Issue 4, Version 2.

39263 **Issue 5**

39264 Moved from X/OPEN UNIX extension to BASE.

39265 **Issue 6**

39266 The **restrict** keyword is added to the *realpath()* prototype for alignment with the
 39267 ISO/IEC 9899:1999 standard.

39268 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
 39269 [ELOOP] error condition is added.

39270 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/51 is applied, adding new text to the
 39271 DESCRIPTION for the case when *resolved_name* is a null pointer, changing the [EINVAL] error
 39272 text, adding text to the RATIONALE, and adding text to FUTURE DIRECTIONS.

39273 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/110 is applied, updating the ERRORS
 39274 section to refer to the *file_name* argument, rather than a non-existent *path* argument.

39275 **NAME**

39276 recv — receive a message from a connected socket

39277 **SYNOPSIS**

39278 #include <sys/socket.h>

39279 ssize_t recv(int socket, void *buffer, size_t length, int flags);

39280 **DESCRIPTION**39281 The *recv()* function shall receive a message from a connection-mode or connectionless-mode
39282 socket. It is normally used with connected sockets because it does not permit the application to
39283 retrieve the source address of received data.39284 The *recv()* function takes the following arguments:39285 *socket* Specifies the socket file descriptor.39286 *buffer* Points to a buffer where the message should be stored.39287 *length* Specifies the length in bytes of the buffer pointed to by the *buffer* argument.39288 *flags* Specifies the type of message reception. Values of this argument are formed by
39289 logically OR'ing zero or more of the following values:39290 MSG_PEEK Peeks at an incoming message. The data is treated as unread and
39291 the next *recv()* or similar function shall still return this data.39292 MSG_OOB Requests out-of-band data. The significance and semantics of
39293 out-of-band data are protocol-specific.39294 MSG_WAITALL On SOCK_STREAM sockets this requests that the function block
39295 until the full amount of data can be returned. The function may
39296 return the smaller amount of data if the socket is a message-
39297 based socket, if a signal is caught, if the connection is terminated,
39298 if MSG_PEEK was specified, or if an error is pending for the
39299 socket.39300 The *recv()* function shall return the length of the message written to the buffer pointed to by the
39301 *buffer* argument. For message-based sockets, such as SOCK_DGRAM and SOCK_SEQPACKET,
39302 the entire message shall be read in a single operation. If a message is too long to fit in the
39303 supplied buffer, and MSG_PEEK is not set in the *flags* argument, the excess bytes shall be
39304 discarded. For stream-based sockets, such as SOCK_STREAM, message boundaries shall be
39305 ignored. In this case, data shall be returned to the user as soon as it becomes available, and no
39306 data shall be discarded.39307 If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first
39308 message.39309 If no messages are available at the socket and O_NONBLOCK is not set on the socket's file
39310 descriptor, *recv()* shall block until a message arrives. If no messages are available at the socket
39311 and O_NONBLOCK is set on the socket's file descriptor, *recv()* shall fail and set *errno* to
39312 [EAGAIN] or [EWOULDBLOCK].39313 **RETURN VALUE**39314 Upon successful completion, *recv()* shall return the length of the message in bytes. If no
39315 messages are available to be received and the peer has performed an orderly shutdown, *recv()*
39316 shall return 0. Otherwise, -1 shall be returned and *errno* set to indicate the error.

ERRORS

The *recv()* function shall fail if:

[EAGAIN] or [EWOULDBLOCK]

The socket's file descriptor is marked O_NONBLOCK and no data is waiting to be received; or MSG_OOB is set and no out-of-band data is available and either the socket's file descriptor is marked O_NONBLOCK or the socket does not support blocking to await out-of-band data.

[EBADF] The *socket* argument is not a valid file descriptor.

[ECONNRESET] A connection was forcibly closed by a peer.

[EINTR] The *recv()* function was interrupted by a signal that was caught, before any data was available.

[EINVAL] The MSG_OOB flag is set and no out-of-band data is available.

[ENOTCONN] A receive is attempted on a connection-mode socket that is not connected.

[ENOTSOCK] The *socket* argument does not refer to a socket.

[EOPNOTSUPP] The specified flags are not supported for this socket type or protocol.

[ETIMEDOUT] The connection timed out during connection establishment, or due to a transmission timeout on active connection.

The *recv()* function may fail if:

[EIO] An I/O error occurred while reading from or writing to the file system.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENOMEM] Insufficient memory was available to fulfill the request.

EXAMPLES

None.

APPLICATION USAGE

The *recv()* function is equivalent to *recvfrom()* with a zero *address_len* argument, and to *read()* if no flags are used.

The *select()* and *poll()* functions can be used to determine when data is available to be received.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

poll(), *read()*, *recvmsg()*, *recvfrom()*, *select()*,

39353 **NAME**

39354 recvfrom — receive a message from a socket

39355 **SYNOPSIS**

```
39356 #include <sys/socket.h>
39357
39357 ssize_t recvfrom(int socket, void *restrict buffer, size_t length,
39358                 int flags, struct sockaddr *restrict address,
39359                 socklen_t *restrict address_len);
```

39360 **DESCRIPTION**

39361 The *recvfrom()* function shall receive a message from a connection-mode or connectionless-mode
 39362 socket. It is normally used with connectionless-mode sockets because it permits the application
 39363 to retrieve the source address of received data.

39364 The *recvfrom()* function takes the following arguments:

39365	<i>socket</i>	Specifies the socket file descriptor.
39366	<i>buffer</i>	Points to the buffer where the message should be stored.
39367	<i>length</i>	Specifies the length in bytes of the buffer pointed to by the <i>buffer</i> argument.
39368	<i>flags</i>	Specifies the type of message reception. Values of this argument are formed by 39369 logically OR'ing zero or more of the following values:
39370	MSG_PEEK	Peeks at an incoming message. The data is treated as unread 39371 and the next <i>recvfrom()</i> or similar function shall still return 39372 this data.
39373	MSG_OOB	Requests out-of-band data. The significance and semantics 39374 of out-of-band data are protocol-specific.
39375	MSG_WAITALL	On SOCK_STREAM sockets this requests that the function 39376 block until the full amount of data can be returned. The 39377 function may return the smaller amount of data if the socket 39378 is a message-based socket, if a signal is caught, if the 39379 connection is terminated, if MSG_PEEK was specified, or if 39380 an error is pending for the socket.
39381	<i>address</i>	A null pointer, or points to a sockaddr structure in which the sending address 39382 is to be stored. The length and format of the address depend on the address 39383 family of the socket.
39384	<i>address_len</i>	Specifies the length of the sockaddr structure pointed to by the <i>address</i> 39385 argument.

39386 The *recvfrom()* function shall return the length of the message written to the buffer pointed to by
 39387 RS the *buffer* argument. For message-based sockets, such as **SOCK_RAW**, **SOCK_DGRAM**, and
 39388 **SOCK_SEQPACKET**, the entire message shall be read in a single operation. If a message is too
 39389 long to fit in the supplied buffer, and MSG_PEEK is not set in the *flags* argument, the excess
 39390 bytes shall be discarded. For stream-based sockets, such as **SOCK_STREAM**, message
 39391 boundaries shall be ignored. In this case, data shall be returned to the user as soon as it becomes
 39392 available, and no data shall be discarded.

39393 If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first
 39394 message.

39395 Not all protocols provide the source address for messages. If the *address* argument is not a null
 39396 pointer and the protocol provides the source address of messages, the source address of the

39397 received message shall be stored in the **sockaddr** structure pointed to by the *address* argument,
 39398 and the length of this address shall be stored in the object pointed to by the *address_len*
 39399 argument.

39400 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
 39401 the stored address shall be truncated.

39402 If the *address* argument is not a null pointer and the protocol does not provide the source address
 39403 of messages, the value stored in the object pointed to by *address* is unspecified.

39404 If no messages are available at the socket and O_NONBLOCK is not set on the socket's file
 39405 descriptor, *recvfrom()* shall block until a message arrives. If no messages are available at the
 39406 socket and O_NONBLOCK is set on the socket's file descriptor, *recvfrom()* shall fail and set *errno*
 39407 to [EAGAIN] or [EWOULDBLOCK].

39408 RETURN VALUE

39409 Upon successful completion, *recvfrom()* shall return the length of the message in bytes. If no
 39410 messages are available to be received and the peer has performed an orderly shutdown,
 39411 *recvfrom()* shall return 0. Otherwise, the function shall return -1 and set *errno* to indicate the
 39412 error.

39413 ERRORS

39414 The *recvfrom()* function shall fail if:

39415 [EAGAIN] or [EWOULDBLOCK]

39416 The socket's file descriptor is marked O_NONBLOCK and no data is waiting
 39417 to be received; or MSG_OOB is set and no out-of-band data is available and
 39418 either the socket's file descriptor is marked O_NONBLOCK or the socket does
 39419 not support blocking to await out-of-band data.

39420 [EBADF] The *socket* argument is not a valid file descriptor.

39421 [ECONNRESET] A connection was forcibly closed by a peer.

39422 [EINTR] A signal interrupted *recvfrom()* before any data was available.

39423 [EINVAL] The MSG_OOB flag is set and no out-of-band data is available.

39424 [ENOTCONN] A receive is attempted on a connection-mode socket that is not connected.

39425 [ENOTSOCK] The *socket* argument does not refer to a socket.

39426 [EOPNOTSUPP] The specified flags are not supported for this socket type.

39427 [ETIMEDOUT] The connection timed out during connection establishment, or due to a
 39428 transmission timeout on active connection.

39429 The *recvfrom()* function may fail if:

39430 [EIO] An I/O error occurred while reading from or writing to the file system.

39431 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

39432 [ENOMEM] Insufficient memory was available to fulfill the request.

recvfrom()

39433

EXAMPLES

39434

None.

39435

APPLICATION USAGE

39436

The *select()* and *poll()* functions can be used to determine when data is available to be received.

39437

RATIONALE

39438

None.

39439

FUTURE DIRECTIONS

39440

None.

39441

SEE ALSO

39442

poll(), *read()*, *recv()*, *recvmsg()*, *select()*, *send()*, *sendmsg()*, *sendto()*, *shutdown()*, *socket()*, *write()*,

39443

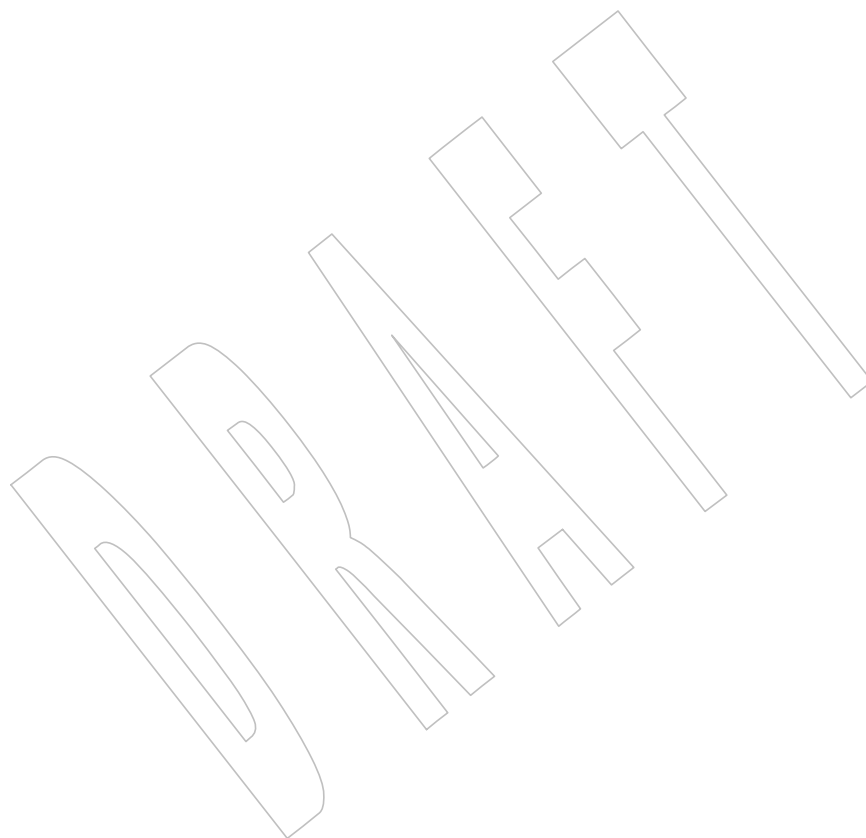
the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

39444

CHANGE HISTORY

39445

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.



NAME

recvmsg — receive a message from a socket

SYNOPSIS

```
#include <sys/socket.h>

ssize_t recvmsg(int socket, struct msghdr *message, int flags);
```

DESCRIPTION

The *recvmsg()* function shall receive a message from a connection-mode or connectionless-mode socket. It is normally used with connectionless-mode sockets because it permits the application to retrieve the source address of received data.

The *recvmsg()* function takes the following arguments:

<i>socket</i>	Specifies the socket file descriptor.
<i>message</i>	Points to a msghdr structure, containing both the buffer to store the source address and the buffers for the incoming message. The length and format of the address depend on the address family of the socket. The <i>msg_flags</i> member is ignored on input, but may contain meaningful values on output.
<i>flags</i>	Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values: <ul style="list-style-type: none"> MSG_OOB Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific. MSG_PEEK Peeks at the incoming message. MSG_WAITALL On SOCK_STREAM sockets this requests that the function block until the full amount of data can be returned. The function may return the smaller amount of data if the socket is a message-based socket, if a signal is caught, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socket.

The *recvmsg()* function shall receive messages from unconnected or connected sockets and shall return the length of the message.

The *recvmsg()* function shall return the total length of the message. For message-based sockets, such as SOCK_DGRAM and SOCK_SEQPACKET, the entire message shall be read in a single operation. If a message is too long to fit in the supplied buffers, and MSG_PEEK is not set in the *flags* argument, the excess bytes shall be discarded, and MSG_TRUNC shall be set in the *msg_flags* member of the **msghdr** structure. For stream-based sockets, such as SOCK_STREAM, message boundaries shall be ignored. In this case, data shall be returned to the user as soon as it becomes available, and no data shall be discarded.

39490 required. The *msg_iov* and *msg_iovlen* fields are used to specify where the received data shall be
 39491 stored. *msg_iov* points to an array of **iovec** structures; *msg_iovlen* shall be set to the dimension of
 39492 this array. In each **iovec** structure, the *iov_base* field specifies a storage area and the *iov_len* field
 39493 gives its size in bytes. Each storage area indicated by *msg_iov* is filled with received data in turn
 39494 until all of the received data is stored or all of the areas have been filled.

39495 Upon successful completion, the *msg_flags* member of the message header shall be the bitwise-
 39496 inclusive OR of all of the following flags that indicate conditions detected for the received
 39497 message:

39498 MSG_EOR End-of-record was received (if supported by the protocol).
 39499 MSG_OOB Out-of-band data was received.
 39500 MSG_TRUNC Normal data was truncated.
 39501 MSG_CTRUNC Control data was truncated.

39502 RETURN VALUE

39503 Upon successful completion, *recvmsg()* shall return the length of the message in bytes. If no
 39504 messages are available to be received and the peer has performed an orderly shutdown,
 39505 *recvmsg()* shall return 0. Otherwise, -1 shall be returned and *errno* set to indicate the error.

39506 ERRORS

39507 The *recvmsg()* function shall fail if:

39508 [EAGAIN] or [EWOULDBLOCK]
 39509 The socket's file descriptor is marked O_NONBLOCK and no data is waiting
 39510 to be received; or MSG_OOB is set and no out-of-band data is available and
 39511 either the socket's file descriptor is marked O_NONBLOCK or the socket does
 39512 not support blocking to await out-of-band data.

39513 [EBADF] The *socket* argument is not a valid open file descriptor.

39514 [ECONNRESET] A connection was forcibly closed by a peer.

39515 [EINTR] This function was interrupted by a signal before any data was available.

39516 [EINVAL] The sum of the *iov_len* values overflows a **ssize_t**, or the MSG_OOB flag is set
 39517 and no out-of-band data is available.

39518 [EMSGSIZE] The *msg_iovlen* member of the **msghdr** structure pointed to by *message* is less
 39519 than or equal to 0, or is greater than {IOV_MAX}.

39520 [ENOTCONN] A receive is attempted on a connection-mode socket that is not connected.

39521 [ENOTSOCK] The *socket* argument does not refer to a socket.

39522 [EOPNOTSUPP] The specified flags are not supported for this socket type.

39523 [ETIMEDOUT] The connection timed out during connection establishment, or due to a
 39524 transmission timeout on active connection.

39525 The *recvmsg()* function may fail if:

39526 [EIO] An I/O error occurred while reading from or writing to the file system.

39527 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

39528 [ENOMEM] Insufficient memory was available to fulfill the request.

39529

EXAMPLES

39530

None.

39531

APPLICATION USAGE

39532

The *select()* and *poll()* functions can be used to determine when data is available to be received.

39533

RATIONALE

39534

None.

39535

FUTURE DIRECTIONS

39536

None.

39537

SEE ALSO

39538

poll(), *recv()*, *recvfrom()*, *select()*, *send()*, *sendmsg()*, *sendto()*, *shutdown()*, *socket()*, the Base

39539

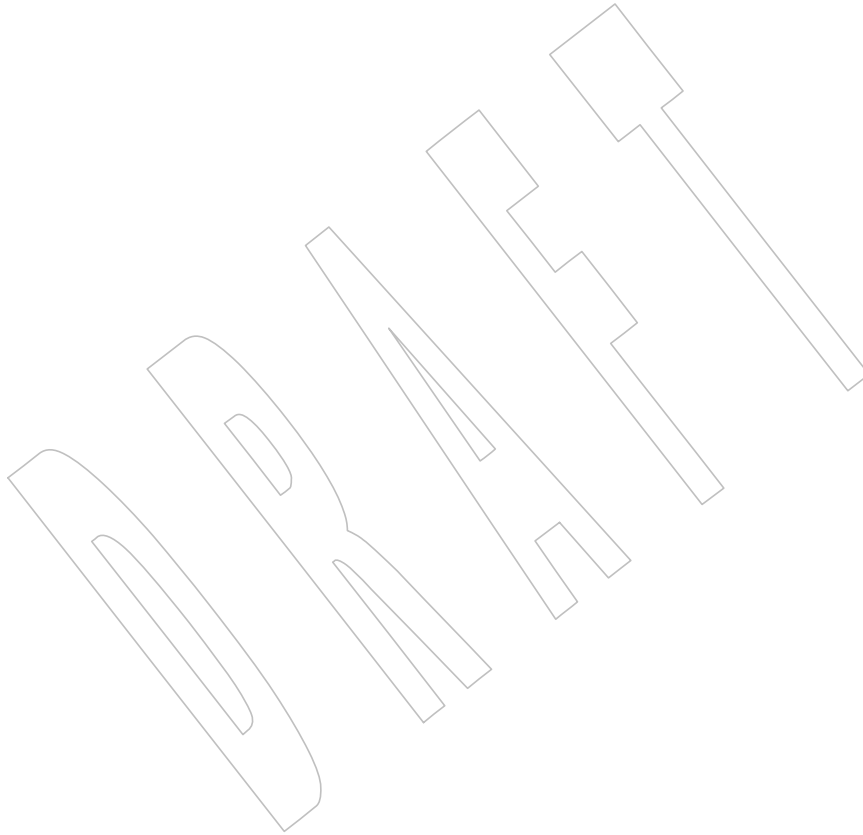
Definitions volume of IEEE Std 1003.1-200x, <sys/socket.h>

39540

CHANGE HISTORY

39541

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.



39542 **NAME**
 39543 regcomp, regerror, regex, regfree — regular expression matching

39544 **SYNOPSIS**
 39545 #include <regex.h>
 39546 int regcomp(regex_t *restrict preg, const char *restrict pattern,
 39547 int cflags);
 39548 size_t regerror(int errcode, const regex_t *restrict preg,
 39549 char *restrict errbuf, size_t errbuf_size);
 39550 int regex(const regex_t *restrict preg, const char *restrict string,
 39551 size_t nmatch, regmatch_t pmatch[restrict], int eflags);
 39552 void regfree(regex_t *preg);

39553 **DESCRIPTION**
 39554 These functions interpret *basic* and *extended* regular expressions as described in the Base
 39555 Definitions volume of IEEE Std 1003.1-200x, Chapter 9, Regular Expressions.

39556 The **regex_t** structure is defined in <regex.h> and contains at least the following member:

Member Type	Member Name	Description
size_t	re_nsub	Number of parenthesized subexpressions.

39559 The **regmatch_t** structure is defined in <regex.h> and contains at least the following members:

Member Type	Member Name	Description
regoff_t	<i>rm_so</i>	Byte offset from start of <i>string</i> to start of substring.
regoff_t	<i>rm_eo</i>	Byte offset from start of <i>string</i> of the first character after the end of substring.

39564 The *regcomp()* function shall compile the regular expression contained in the string pointed to by
 39565 the *pattern* argument and place the results in the structure pointed to by *preg*. The *cflags*
 39566 argument is the bitwise-inclusive OR of zero or more of the following flags, which are defined in
 39567 the <regex.h> header:

39568 REG_EXTENDED Use Extended Regular Expressions.
 39569 REG_ICASE Ignore case in match. (See the Base Definitions volume of
 39570 IEEE Std 1003.1-200x, Chapter 9, Regular Expressions.)
 39571 REG_NOSUB Report only success/fail in *regex()*.
 39572 REG_NEWLINE Change the handling of <newline>s, as described in the text.

39573 The default regular expression type for *pattern* is a Basic Regular Expression. The application can
 39574 specify Extended Regular Expressions using the REG_EXTENDED *cflags* flag.

39575 If the REG_NOSUB flag was not set in *cflags*, then *regcomp()* shall set *re_nsub* to the number of
 39576 parenthesized subexpressions (delimited by "\(\)" in basic regular expressions or "()" in
 39577 extended regular expressions) found in *pattern*.

39578 The *regex()* function compares the null-terminated string specified by *string* with the compiled
 39579 regular expression *preg* initialized by a previous call to *regcomp()*. If it finds a match, *regex()*
 39580 shall return 0; otherwise, it shall return non-zero indicating either no match or an error. The
 39581 *eflags* argument is the bitwise-inclusive OR of zero or more of the following flags, which are
 39582 defined in the <regex.h> header:

39583 REG_NOTBOL The first character of the string pointed to by *string* is not the beginning of the
 39584 line. Therefore, the circumflex character ('^'), when taken as a special
 39585 character, shall not match the beginning of *string*.

39586 REG_NOTEOL The last character of the string pointed to by *string* is not the end of the line.
 39587 Therefore, the dollar sign ('\$'), when taken as a special character, shall not
 39588 match the end of *string*.

39589 If *nmatch* is 0 or REG_NOSUB was set in the *cflags* argument to *regcomp()*, then *regexec()* shall
 39590 ignore the *pmatch* argument. Otherwise, the application shall ensure that the *pmatch* argument
 39591 points to an array with at least *nmatch* elements, and *regexec()* shall fill in the elements of that
 39592 array with offsets of the substrings of *string* that correspond to the parenthesized subexpressions
 39593 of *pattern*: *pmatch[i].rm_so* shall be the byte offset of the beginning and *pmatch[i].rm_eo* shall be
 39594 one greater than the byte offset of the end of substring *i*. (Subexpression *i* begins at the *i*th
 39595 matched open parenthesis, counting from 1.) Offsets in *pmatch[0]* identify the substring that
 39596 corresponds to the entire regular expression. Unused elements of *pmatch* up to *pmatch[nmatch-1]*
 39597 shall be filled with -1. If there are more than *nmatch* subexpressions in *pattern* (*pattern* itself
 39598 counts as a subexpression), then *regexec()* shall still do the match, but shall record only the first
 39599 *nmatch* substrings.

39600 When matching a basic or extended regular expression, any given parenthesized subexpression
 39601 of *pattern* might participate in the match of several different substrings of *string*, or it might not
 39602 match any substring even though the pattern as a whole did match. The following rules shall be
 39603 used to determine which substrings to report in *pmatch* when matching regular expressions:

39604 1. If subexpression *i* in a regular expression is not contained within another subexpression,
 39605 and it participated in the match several times, then the byte offsets in *pmatch[i]* shall
 39606 delimit the last such match.

39607 2. If subexpression *i* is not contained within another subexpression, and it did not
 39608 participate in an otherwise successful match, the byte offsets in *pmatch[i]* shall be -1. A
 39609 subexpression does not participate in the match when:

39610 ' * ' or " \{ \} " appears immediately after the subexpression in a basic regular
 39611 expression, or ' * ', ' ? ', or " { } " appears immediately after the subexpression in
 39612 an extended regular expression, and the subexpression did not match (matched 0
 39613 times)

39614 or:

39615 ' | ' is used in an extended regular expression to select this subexpression or
 39616 another, and the other subexpression matched.

39617 3. If subexpression *i* is contained within another subexpression *j*, and *i* is not contained
 39618 within any other subexpression that is contained within *j*, and a match of subexpression *j*
 39619 is reported in *pmatch[j]*, then the match or non-match of subexpression *i* reported in
 39620 *pmatch[i]* shall be as described in 1. and 2. above, but within the substring reported in
 39621 *pmatch[j]* rather than the whole string. The offsets in *pmatch[i]* are still relative to the start
 39622 of *string*.

39623 4. If subexpression *i* is contained in subexpression *j*, and the byte offsets in *pmatch[j]* are -1,
 39624 then the pointers in *pmatch[i]* shall also be -1.

39625 5. If subexpression *i* matched a zero-length string, then both byte offsets in *pmatch[i]* shall be
 39626 the byte offset of the character or null terminator immediately following the zero-length
 39627 string.

39628 If, when *regexec()* is called, the locale is different from when the regular expression was
 39629 compiled, the result is undefined.

39630 If REG_NEWLINE is not set in *cflags*, then a <newline> in *pattern* or *string* shall be treated as an
 39631 ordinary character. If REG_NEWLINE is set, then <newline> shall be treated as an ordinary
 39632 character except as follows:

- 39633 1. A <newline> in *string* shall not be matched by a period outside a bracket expression or by
 39634 any form of a non-matching list (see the Base Definitions volume of IEEE Std 1003.1-200x,
 39635 Chapter 9, Regular Expressions).
- 39636 2. A circumflex ('^') in *pattern*, when used to specify expression anchoring (see the Base
 39637 Definitions volume of IEEE Std 1003.1-200x, Section 9.3.8, BRE Expression Anchoring),
 39638 shall match the zero-length string immediately after a <newline> in *string*, regardless of
 39639 the setting of REG_NOTBOL.
- 39640 3. A dollar sign ('\$') in *pattern*, when used to specify expression anchoring, shall match the
 39641 zero-length string immediately before a <newline> in *string*, regardless of the setting of
 39642 REG_NOTEOL.

39643 The *regfree()* function frees any memory allocated by *regcomp()* associated with *preg*.

39644 The following constants are defined as error return values:

39645	REG_NOMATCH	<i>regexexec()</i> failed to match.
39646	REG_BADPAT	Invalid regular expression.
39647	REG_ECOLLATE	Invalid collating element referenced.
39648	REG_ECTYPE	Invalid character class type referenced.
39649	REG_EESCAPE	Trailing '\\ ' in pattern.
39650	REG_ESUBREG	Number in "\\digit" invalid or in error.
39651	REG_EBRACK	"[]" imbalance.
39652	REG_EPAREN	"\\(\\)" or "()" imbalance.
39653	REG_EBRACE	"\\{\\}" imbalance.
39654	REG_BADBR	Content of "\\{\\}" invalid: not a number, number too large, more than 39655 two numbers, first larger than second.
39656	REG_ERANGE	Invalid endpoint in range expression.
39657	REG_ESPACE	Out of memory.
39658	REG_BADRPT	'?', '*', or '+' not preceded by valid regular expression.

39659 The *regerror()* function provides a mapping from error codes returned by *regcomp()* and
 39660 *regexexec()* to unspecified printable strings. It generates a string corresponding to the value of the
 39661 *errcode* argument, which the application shall ensure is the last non-zero value returned by
 39662 *regcomp()* or *regexexec()* with the given value of *preg*. If *errcode* is not such a value, the content of
 39663 the generated string is unspecified.

39664 If *preg* is a null pointer, but *errcode* is a value returned by a previous call to *regexexec()* or *regcomp()*,
 39665 the *regerror()* still generates an error string corresponding to the value of *errcode*, but it might not
 39666 be as detailed under some implementations.

39667 If the *errbuf_size* argument is not 0, *regerror()* shall place the generated string into the buffer of
 39668 size *errbuf_size* bytes pointed to by *errbuf*. If the string (including the terminating null) cannot fit
 39669 in the buffer, *regerror()* shall truncate the string and null-terminate the result.

39670 If *errbuf_size* is 0, *regerror()* shall ignore the *errbuf* argument, and return the size of the buffer
 39671 needed to hold the generated string.

39672 If the *preg* argument to *regexexec()* or *regfree()* is not a compiled regular expression returned by

39673 *regcomp()*, the result is undefined. A *preg* is no longer treated as a compiled regular expression
39674 after it is given to *regfree()*.

39675 RETURN VALUE

39676 Upon successful completion, the *regcomp()* function shall return 0. Otherwise, it shall return an
39677 integer value indicating an error as described in <regex.h>, and the content of *preg* is undefined.
39678 If a code is returned, the interpretation shall be as given in <regex.h>.

39679 If *regcomp()* detects an invalid RE, it may return REG_BADPAT, or it may return one of the error
39680 codes that more precisely describes the error.

39681 Upon successful completion, the *regexec()* function shall return 0. Otherwise, it shall return
39682 REG_NOMATCH to indicate no match.

39683 Upon successful completion, the *regerror()* function shall return the number of bytes needed to
39684 hold the entire generated string, including the null termination. If the return value is greater
39685 than *errbuf_size*, the string returned in the buffer pointed to by *errbuf* has been truncated.

39686 The *regfree()* function shall not return a value.

39687 ERRORS

39688 No errors are defined.

39689 EXAMPLES

```
39690 #include <regex.h>
39691 /*
39692  * Match string against the extended regular expression in
39693  * pattern, treating errors as no match.
39694  *
39695  * Return 1 for match, 0 for no match.
39696  */
39697 int
39698 match(const char *string, char *pattern)
39699 {
39700     int    status;
39701     regex_t re;
39702     if (regcomp(&re, pattern, REG_EXTENDED|REG_NOSUB) != 0) {
39703         return(0); /* Report error. */
39704     }
39705     status = regexec(&re, string, (size_t) 0, NULL, 0);
39706     regfree(&re);
39707     if (status != 0) {
39708         return(0); /* Report error. */
39709     }
39710     return(1);
39711 }
```

39712 The following demonstrates how the REG_NOTBOL flag could be used with *regexec()* to find all
39713 substrings in a line that match a pattern supplied by a user. (For simplicity of the example, very
39714 little error checking is done.)

```
39715 (void) regcomp (&re, pattern, 0);
39716 /* This call to regexec() finds the first match on the line. */
39717 error = regexec (&re, &buffer[0], 1, &pm, 0);
39718 while (error == 0) { /* While matches found. */
39719     /* Substring found between pm.rm_so and pm.rm_eo. */
39720     /* This call to regexec() finds the next match. */
```

```

39721         error = regexec (&re, buffer + pm.rm_eo, 1, &pm, REG_NOTBOL);
39722     }

```

APPLICATION USAGE

An application could use:

```

39725     regerror(code, preg, (char *)NULL, (size_t)0)

```

to find out how big a buffer is needed for the generated string, *malloc()* a buffer to hold the string, and then call *regerror()* again to get the string. Alternatively, it could allocate a fixed, static buffer that is big enough to hold most strings, and then use *malloc()* to allocate a larger buffer if it finds that this is too small.

To match a pattern as described in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.13, Pattern Matching Notation, use the *fnmatch()* function.

RATIONALE

The *regexec()* function must fill in all *nmatch* elements of *pmatch*, where *nmatch* and *pmatch* are supplied by the application, even if some elements of *pmatch* do not correspond to subexpressions in *pattern*. The application writer should note that there is probably no reason for using a value of *nmatch* that is larger than *preg->re_nsub+1*.

The REG_NEWLINE flag supports a use of RE matching that is needed in some applications like text editors. In such applications, the user supplies an RE asking the application to find a line that matches the given expression. An anchor in such an RE anchors at the beginning or end of any line. Such an application can pass a sequence of <newline>-separated lines to *regexec()* as a single long string and specify REG_NEWLINE to *regcomp()* to get the desired behavior. The application must ensure that there are no explicit <newline>s in *pattern* if it wants to ensure that any match occurs entirely within a single line.

The REG_NEWLINE flag affects the behavior of *regexec()*, but it is in the *cflags* parameter to *regcomp()* to allow flexibility of implementation. Some implementations will want to generate the same compiled RE in *regcomp()* regardless of the setting of REG_NEWLINE and have *regexec()* handle anchors differently based on the setting of the flag. Other implementations will generate different compiled REs based on the REG_NEWLINE.

The REG_ICASE flag supports the operations taken by the *grep -i* option and the historical implementations of *ex* and *vi*. Including this flag will make it easier for application code to be written that does the same thing as these utilities.

The substrings reported in *pmatch[]* are defined using offsets from the start of the string rather than pointers. This allows type-safe access to both constant and non-constant strings.

The type **regoff_t** is used for the elements of *pmatch[]* to ensure that the application can represent large arrays in memory (important for an application conforming to the Shell and Utilities volume of IEEE Std 1003.1-200x).

The 1992 edition of this standard required **regoff_t** to be at least as wide as **off_t**, to facilitate future extensions in which the string to be searched is taken from a file. However, these future extensions have not appeared. The requirement rules out popular implementations with 32-bit **regoff_t** and 64-bit **off_t**, so it has been withdrawn.

The standard developers rejected the inclusion of a *regsub()* function that would be used to do substitutions for a matched RE. While such a routine would be useful to some applications, its utility would be much more limited than the matching function described here. Both RE parsing and substitution are possible to implement without support other than that required by the ISO C standard, but matching is much more complex than substituting. The only difficult part of substitution, given the information supplied by *regexec()*, is finding the next character in a string when there can be multi-byte characters. That is a much larger issue, and one that needs a more general solution.

39769 The *errno* variable has not been used for error returns to avoid filling the *errno* name space for
 39770 this feature.

39771 The interface is defined so that the matched substrings *rm_sp* and *rm_ep* are in a separate
 39772 **regmatch_t** structure instead of in **regex_t**. This allows a single compiled RE to be used
 39773 simultaneously in several contexts; in *main()* and a signal handler, perhaps, or in multiple
 39774 threads of lightweight processes. (The *preg* argument to *regexec()* is declared with type **const**, so
 39775 the implementation is not permitted to use the structure to store intermediate results.) It also
 39776 allows an application to request an arbitrary number of substrings from an RE. The number of
 39777 subexpressions in the RE is reported in *re_nsub* in *preg*. With this change to *regexec()*,
 39778 consideration was given to dropping the REG_NOSUB flag since the user can now specify this
 39779 with a zero *nmatch* argument to *regexec()*. However, keeping REG_NOSUB allows an
 39780 implementation to use a different (perhaps more efficient) algorithm if it knows in *regcomp()* that
 39781 no subexpressions need be reported. The implementation is only required to fill in *pmatch* if
 39782 *nmatch* is not zero and if REG_NOSUB is not specified. Note that the **size_t** type, as defined in
 39783 the ISO C standard, is unsigned, so the description of *regexec()* does not need to address
 39784 negative values of *nmatch*.

39785 REG_NOTBOL was added to allow an application to do repeated searches for the same pattern
 39786 in a line. If the pattern contains a circumflex character that should match the beginning of a line,
 39787 then the pattern should only match when matched against the beginning of the line. Without
 39788 the REG_NOTBOL flag, the application could rewrite the expression for subsequent matches,
 39789 but in the general case this would require parsing the expression. The need for REG_NOTEOL is
 39790 not as clear; it was added for symmetry.

39791 The addition of the *regerror()* function addresses the historical need for conforming application
 39792 programs to have access to error information more than “Function failed to compile/match your
 39793 RE for unknown reasons”.

39794 This interface provides for two different methods of dealing with error conditions. The specific
 39795 error codes (REG_EBRACE, for example), defined in **<regex.h>**, allow an application to recover
 39796 from an error if it is so able. Many applications, especially those that use patterns supplied by a
 39797 user, will not try to deal with specific error cases, but will just use *regerror()* to obtain a human-
 39798 readable error message to present to the user.

39799 The *regerror()* function uses a scheme similar to *confstr()* to deal with the problem of allocating
 39800 memory to hold the generated string. The scheme used by *strerror()* in the ISO C standard was
 39801 considered unacceptable since it creates difficulties for multi-threaded applications.

39802 The *preg* argument is provided to *regerror()* to allow an implementation to generate a more
 39803 descriptive message than would be possible with *errcode* alone. An implementation might, for
 39804 example, save the character offset of the offending character of the pattern in a field of *preg*, and
 39805 then include that in the generated message string. The implementation may also ignore *preg*.

39806 A REG_FILENAME flag was considered, but omitted. This flag caused *regexec()* to match
 39807 patterns as described in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.13,
 39808 Pattern Matching Notation instead of REs. This service is now provided by the *fnmatch()*
 39809 function.

39810 Notice that there is a difference in philosophy between the ISO POSIX-2:1993 standard and
 39811 IEEE Std 1003.1-200x in how to handle a “bad” regular expression. The ISO POSIX-2:1993
 39812 standard says that many bad constructs “produce undefined results”, or that “the interpretation
 39813 is undefined”. IEEE Std 1003.1-200x, however, says that the interpretation of such REs is
 39814 unspecified. The term “undefined” means that the action by the application is an error, of similar
 39815 severity to passing a bad pointer to a function.

39816 The *regcomp()* and *regexec()* functions are required to accept any null-terminated string as the
 39817 *pattern* argument. If the meaning of the string is “undefined”, the behavior of the function is
 39818 “unspecified”. IEEE Std 1003.1-200x does not specify how the functions will interpret the

39819 pattern; they might return error codes, or they might do pattern matching in some completely
 39820 unexpected way, but they should not do something like abort the process.

FUTURE DIRECTIONS

39821 None.
 39822

SEE ALSO

39823 *fnmatch()*, *glob()*, Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.13, Pattern
 39824 Matching Notation, Base Definitions volume of IEEE Std 1003.1-200x, Chapter 9, Regular
 39825 Expressions, **<regex.h>**, **<sys/types.h>**
 39826

CHANGE HISTORY

39827 First released in Issue 4. Derived from the ISO POSIX-2 standard.
 39828

Issue 5

39829 Moved from POSIX2 C-language Binding to BASE.
 39830

Issue 6

39831 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.
 39832

39833 The following new requirements on POSIX implementations derive from alignment with the
 39834 Single UNIX Specification:

- 39835 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
 39836 required for conforming implementations of previous POSIX specifications, it was not
 39837 required for UNIX applications.

39838 The normative text is updated to avoid use of the term “must” for application requirements.

39839 The REG_ENOSYS constant is removed.

39840 The **restrict** keyword is added to the *regcomp()*, *regerror()*, and *regexec()* prototypes for
 39841 alignment with the ISO/IEC 9899:1999 standard.

Issue 7

39842 SD5-XBD-ERN-60 is applied.
 39843

39844 **NAME**
 39845 remainder, remainderf, remainderl — remainder function

39846 **SYNOPSIS**
 39847 #include <math.h>
 39848 double remainder(double x, double y);
 39849 float remainderf(float x, float y);
 39850 long double remainderl(long double x, long double y);

39851 **DESCRIPTION**

39852 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 39853 conflict between the requirements described here and the ISO C standard is unintentional. This
 39854 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

39855 These functions shall return the floating-point remainder $r=x-ny$ when y is non-zero. The value
 39856 n is the integral value nearest the exact value x/y . When $|n-x/y|=\frac{1}{2}$, the value n is chosen to
 39857 be even.

39858 The behavior of *remainder()* shall be independent of the rounding mode.

39859 **RETURN VALUE**

39860 Upon successful completion, these functions shall return the floating-point remainder $r=x-ny$
 39861 when y is non-zero.

39862 On systems that do not support the IEC 60559 Floating-Point option, if y is zero, it is
 39863 implementation-defined whether a domain error occurs or zero is returned.

39864 MX If x or y is NaN, a NaN shall be returned.

39865 If x is infinite or y is 0 and the other is non-NaN, a domain error shall occur, and either a NaN (if
 39866 supported), or an implementation-defined value shall be returned.

39867 **ERRORS**

39868 These functions shall fail if:

39869 MX **Domain Error** The x argument is $\pm\text{Inf}$, or the y argument is ± 0 and the other argument is non-
 39870 NaN.

39871 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 39872 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 39873 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 39874 shall be raised.

39875 These functions may fail if:

39876 **Domain Error** The y argument is zero.

39877 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 39878 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 39879 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 39880 shall be raised.

remainder()

39881
39882
39883
39884
39885
39886
39887
39888
39889
39890
39891
39892
39893
39894
39895
39896
39897
39898
39899
39900
39901
39902
39903
39904
39905
39906
39907

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

abs(), *div()*, *feclearexcept()*, *fetestexcept()*, *ldiv()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The *remainder()* function is no longer marked as an extension.

The *remainderf()* and *remainderl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

Issue 7

ISO/IEC 9899:1999 standard, Technical Corrigendum 2 #55 (SD5-XSH-ERN-82) is applied.

39908 **NAME**

39909 remove — remove a file

39910 **SYNOPSIS**

39911 #include <stdio.h>

39912 int remove(const char *path);

39913 **DESCRIPTION**39914 CX The functionality described on this reference page is aligned with the ISO C standard. Any
39915 conflict between the requirements described here and the ISO C standard is unintentional. This
39916 volume of IEEE Std 1003.1-200x defers to the ISO C standard.39917 The *remove()* function shall cause the file named by the pathname pointed to by *path* to be no
39918 longer accessible by that name. A subsequent attempt to open that file using that name shall fail,
39919 unless it is created anew.39920 CX If *path* does not name a directory, *remove(path)* shall be equivalent to *unlink(path)*.39921 If *path* names a directory, *remove(path)* shall be equivalent to *rmdir(path)*.39922 **RETURN VALUE**39923 CX Refer to *rmdir()* or *unlink()*.39924 **ERRORS**39925 CX Refer to *rmdir()* or *unlink()*.39926 **EXAMPLES**39927 **Removing Access to a File**39928 The following example shows how to remove access to a file named */home/cnd/old_mods*.

39929 #include <stdio.h>

39930 int status;

39931 ...

39932 status = remove("/home/cnd/old_mods");

39933 **APPLICATION USAGE**

39934 None.

39935 **RATIONALE**

39936 None.

39937 **FUTURE DIRECTIONS**

39938 None.

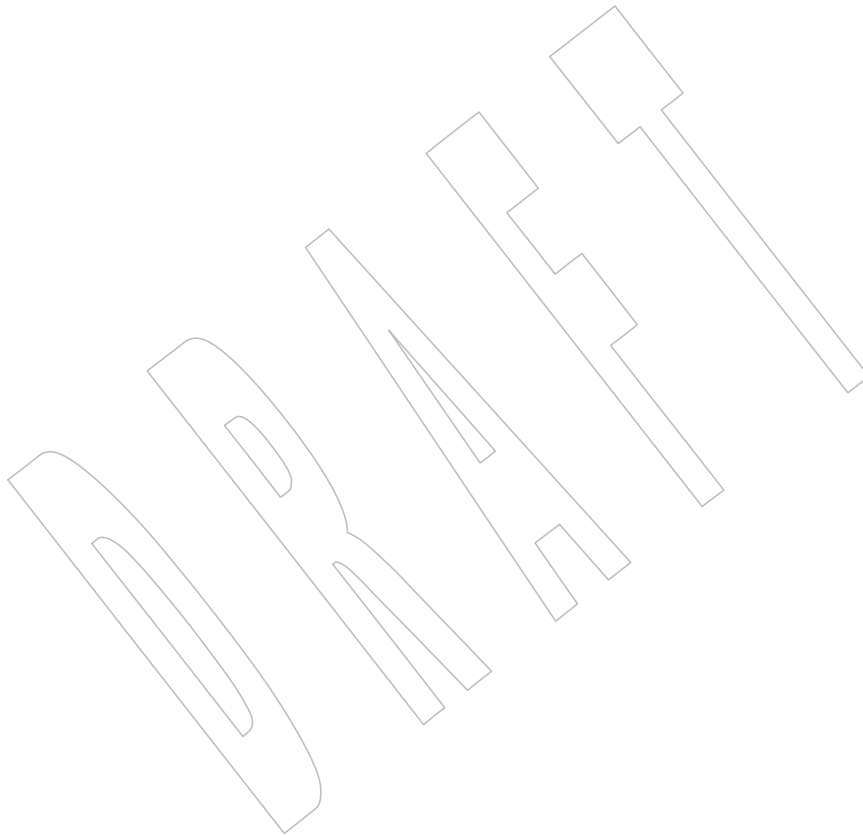
39939 **SEE ALSO**39940 *rmdir()*, *unlink()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>39941 **CHANGE HISTORY**39942 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard and the ISO C
39943 standard.39944 **Issue 6**

39945 Extensions beyond the ISO C standard are marked.

39946 The following new requirements on POSIX implementations derive from alignment with the
39947 Single UNIX Specification:

remove()39948
39949
39950

- The DESCRIPTION, RETURN VALUE, and ERRORS sections are updated so that if *path* is not a directory, *remove()* is equivalent to *unlink()*, and if it is a directory, it is equivalent to *rmdir()*.



39951 **NAME**
39952 `remque` — remove an element from a queue

39953 **SYNOPSIS**

39954 XSI `#include <search.h>`
39955 `void remque(void *element);`

39956 **DESCRIPTION**

39957 Refer to [insque\(\)](#).

39958 **NAME**
 39959 remquo, remquof, remquol — remainder functions

39960 **SYNOPSIS**
 39961 #include <math.h>
 39962 double remquo(double x, double y, int *quo);
 39963 float remquof(float x, float y, int *quo);
 39964 long double remquol(long double x, long double y, int *quo);

39965 **DESCRIPTION**
 39966 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 39967 conflict between the requirements described here and the ISO C standard is unintentional. This
 39968 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

39969 The *remquo()*, *remquof()*, and *remquol()* functions shall compute the same remainder as the
 39970 *remainder()*, *remainderf()*, and *remainderl()* functions, respectively. In the object pointed to by *quo*,
 39971 they store a value whose sign is the sign of x/y and whose magnitude is congruent modulo 2^n
 39972 to the magnitude of the integral quotient of x/y , where n is an implementation-defined integer
 39973 greater than or equal to 3. If y is zero, the value stored in the object pointed to by *quo* is
 39974 unspecified.

39975 An application wishing to check for error situations should set *errno* to zero and call
 39976 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 39977 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 39978 zero, an error has occurred.

39979 **RETURN VALUE**
 39980 These functions shall return $x \text{ REM } y$.
 39981 On systems that do not support the IEC 60559 Floating-Point option, if y is zero, it is
 39982 implementation-defined whether a domain error occurs or zero is returned.

39983 MX If x or y is NaN, a NaN shall be returned.
 39984 If x is $\pm\text{Inf}$ or y is zero and the other argument is non-NaN, a domain error shall occur, and either
 39985 a NaN (if supported), or an implementation-defined value shall be returned.

39986 **ERRORS**
 39987 These functions shall fail if:

39988 MX **Domain Error** The x argument is $\pm\text{Inf}$, or the y argument is ± 0 and the other argument is non-
 39989 NaN.
 39990 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 39991 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 39992 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 39993 shall be raised.

39994 These functions may fail if:
 39995 **Domain Error** The y argument is zero.
 39996 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 39997 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 39998 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 39999 shall be raised.

40000
40001
40002
40003
40004
40005
40006
40007
40008
40009
40010
40011
40012
40013
40014
40015
40016
40017

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

These functions are intended for implementing argument reductions which can exploit a few low-order bits of the quotient. Note that x may be so large in magnitude relative to y that an exact representation of the quotient is not practical.

FUTURE DIRECTIONS

None.

SEE ALSO

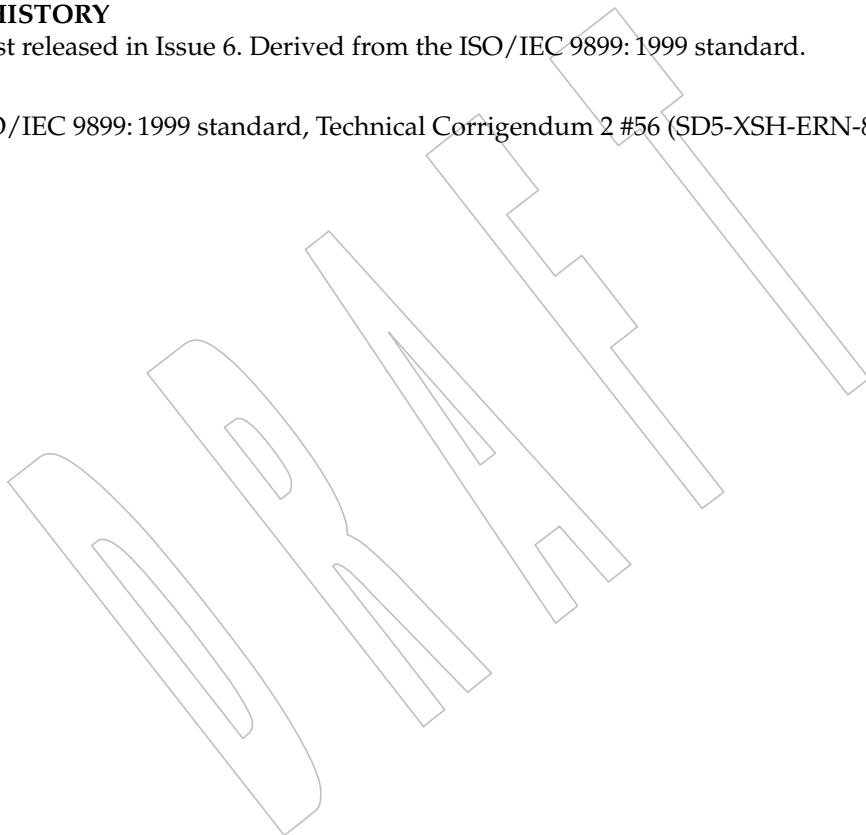
feclearexcept(), *fetestexcept()*, *remainder()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

Issue 7

ISO/IEC 9899:1999 standard, Technical Corrigendum 2 #56 (SD5-XSH-ERN-83) is applied.



40018 **NAME**

40019 rename, renameat — rename file relative to directory file descriptor

40020 **SYNOPSIS**

40021 #include <stdio.h>

40022 int rename(const char *old, const char *new);

40023 CX int renameat(int oldfd, const char *old, int newfd,
40024 const char *new);40025 **DESCRIPTION**40026 CX For *rename()*: The functionality described on this reference page is aligned with the ISO C
40027 standard. Any conflict between the requirements described here and the ISO C standard is
40028 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.40029 The *rename()* function shall change the name of a file. The *old* argument points to the pathname
40030 of the file to be renamed. The *new* argument points to the new pathname of the file.40031 CX If either the *old* or *new* argument names a symbolic link, *rename()* shall operate on the symbolic
40032 link itself, and shall not resolve the last component of the argument. If the *old* argument and the
40033 *new* argument resolve to the same existing file, *rename()* shall return successfully and perform no
40034 other action.40035 If the *old* argument points to the pathname of a file that is not a directory, the *new* argument shall
40036 not point to the pathname of a directory. If the link named by the *new* argument exists, it shall be
40037 removed and *old* renamed to *new*. In this case, a link named *new* shall remain visible to other
40038 processes throughout the renaming operation and refer either to the file referred to by *new* or *old*
40039 before the operation began. Write access permission is required for both the directory containing
40040 *old* and the directory containing *new*.40041 If the *old* argument points to the pathname of a directory, the *new* argument shall not point to the
40042 pathname of a file that is not a directory. If the directory named by the *new* argument exists, it
40043 shall be removed and *old* renamed to *new*. In this case, a link named *new* shall exist throughout
40044 the renaming operation and shall refer either to the directory referred to by *new* or *old* before the
40045 operation began. If *new* names an existing directory, it shall be required to be an empty directory.40046 If either *pathname* argument refers to a path whose final component is either dot or dot-dot,
40047 *rename()* shall fail.40048 If the *old* argument points to a pathname of a symbolic link, the symbolic link shall be renamed.
40049 If the *new* argument points to a pathname of a symbolic link, the symbolic link shall be removed.40050 The *new* pathname shall not contain a path prefix that names *old*. Write access permission is
40051 required for the directory containing *old* and the directory containing *new*. If the *old* argument
40052 points to the pathname of a directory, write access permission may be required for the directory
40053 named by *old*, and, if it exists, the directory named by *new*.40054 If the link named by the *new* argument exists and the file's link count becomes 0 when it is
40055 removed and no process has the file open, the space occupied by the file shall be freed and the
40056 file shall no longer be accessible. If one or more processes have the file open when the last link is
40057 removed, the link shall be removed before *rename()* returns, but the removal of the file contents
40058 shall be postponed until all references to the file are closed.40059 Upon successful completion, *rename()* shall mark for update the *st_ctime* and *st_mtime* fields of
40060 the parent directory of each file.40061 If the *rename()* function fails for any reason other than [EIO], any file named by *new* shall be

40062

unaffected.

40063

40064

40065

40066

40067

40068

40069

The *renameat()* function shall be equivalent to the *rename()* function except in the case where either *old* or *new* specifies a relative path. If *old* is a relative path, the file to be renamed is located relative to the directory associated with the file descriptor *oldfd* instead of the current working directory. If *new* is a relative path, the same happens only relative to the directory associated with *newfd*. It is unspecified whether directory searches are permitted based on whether the file was opened with search permission or on the current permissions of the directory underlying the file descriptor.

40070

40071

If *renameat()* is passed the special value *AT_FDCWD* in the *oldfd* or *newfd* parameter, the current working directory shall be used in the determination of the file for the respective *path* parameter.

40072

RETURN VALUE

40073

40074

40075

CX Upon successful completion, the *rename()* function shall return 0. Otherwise, it shall return *-1*, *errno* shall be set to indicate the error, and neither the file named by *old* nor the file named by *new* shall be changed or created.

40076

40077

CX Upon successful completion, the *renameat()* function shall return 0. Otherwise, it shall return *-1* and set *errno* to indicate the error.

40078

ERRORS

40079

CX The *rename()* and *renameat()* functions shall fail if:

40080

40081

40082

40083

CX **[EACCES]** A component of either path prefix denies search permission; or one of the directories containing *old* or *new* denies write permissions; or, write permission is required and is denied for a directory pointed to by the *old* or *new* arguments.

40084

40085

CX **[EBUSY]** The directory named by *old* or *new* is currently in use by the system or another process, and the implementation considers this an error.

40086

40087

CX **[EEXIST] or [ENOTEMPTY]**

The link named by *new* is a directory that is not an empty directory.

40088

40089

40090

CX **[EINVAL]** The *new* directory pathname contains a path prefix that names the *old* directory, or either *pathname* argument contains a final component that is dot or dot-dot.

40091

CX **[EIO]** A physical I/O error has occurred.

40092

40093

CX **[EISDIR]** The *new* argument points to a directory and the *old* argument points to a file that is not a directory.

40094

40095

CX **[ELOOP]** A loop exists in symbolic links encountered during resolution of the *path* argument.

40096

40097

CX **[EMLINK]** The file named by *old* is a directory, and the link count of the parent directory of *new* would exceed *{LINK_MAX}*.

40098

40099

40100

CX **[ENAMETOOLONG]**

The length of the *old* or *new* argument exceeds *{PATH_MAX}* or a pathname component is longer than *{NAME_MAX}*.

40101

40102

CX **[ENOENT]** The link named by *old* does not name an existing file, or either *old* or *new* points to an empty string.

40103

CX **[ENOSPC]** The directory that would contain *new* cannot be extended.

40104

40105

CX **[ENOTDIR]** A component of either path prefix is not a directory; or the *old* argument names a directory and *new* argument names a non-directory file.

rename()

System Interfaces

40106	XSI	[EPERM] or [EACCES]	
40107			The S_ISVTX flag is set on the directory containing the file referred to by <i>old</i>
40108			and the caller is not the file owner, nor is the caller the directory owner, nor
40109			does the caller have appropriate privileges; or <i>new</i> refers to an existing file, the
40110			S_ISVTX flag is set on the directory containing this file, and the caller is not
40111			the file owner, nor is the caller the directory owner, nor does the caller have
40112			appropriate privileges.
40113	CX	[EROFS]	The requested operation requires writing in a directory on a read-only file
40114			system.
40115	CX	[EXDEV]	The links named by <i>new</i> and <i>old</i> are on different file systems and the
40116			implementation does not support links between file systems.
40117	CX		In addition, the <i>renameat()</i> function shall fail if:
40118		[EBADF]	The <i>old</i> argument does not specify an absolute path and the <i>oldfd</i> argument is
40119			neither AT_FDCWD nor a valid file descriptor open for searching, or the <i>new</i>
40120			argument does not specify an absolute path and the <i>newfd</i> argument is neither
40121			AT_FDCWD nor a valid file descriptor open for searching.
40122	CX		The <i>rename()</i> and <i>renameat()</i> functions may fail if:
40123	OB XSR	[EBUSY]	The file named by the <i>old</i> or <i>new</i> arguments is a named STREAM.
40124	CX	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during
40125			resolution of the <i>path</i> argument.
40126	CX	[ENAMETOOLONG]	
40127			As a result of encountering a symbolic link in resolution of the <i>path</i> argument,
40128			the length of the substituted pathname string exceeded {PATH_MAX}.
40129	CX	[ETXTBSY]	The file to be renamed is a pure procedure (shared text) file that is being
40130			executed.
40131	CX		The <i>renameat()</i> function may fail if:
40132		[ENOTDIR]	The <i>old</i> argument is not an absolute path and <i>oldfd</i> is neither AT_FDCWD nor
40133			a file descriptor associated with a directory, or the <i>new</i> argument is not an
40134			absolute path and <i>newfd</i> is neither AT_FDCWD nor a file descriptor associated
40135			with a directory.

EXAMPLES**Renaming a File**

The following example shows how to rename a file named `/home/cnd/mod1` to `/home/cnd/mod2`.

```
#include <stdio.h>

int status;
...
status = rename("/home/cnd/mod1", "/home/cnd/mod2");
```

APPLICATION USAGE

Some implementations mark for update the *st_ctime* field of renamed files and some do not. Applications which make use of the *st_ctime* field may behave differently with respect to renamed files unless they are designed to allow for either behavior.

RATIONALE

This *rename()* function is equivalent for regular files to that defined by the ISO C standard. Its inclusion here expands that definition to include actions on directories and specifies behavior when the *new* parameter names a file that already exists. That specification requires that the action of the function be atomic.

One of the reasons for introducing this function was to have a means of renaming directories while permitting implementations to prohibit the use of *link()* and *unlink()* with directories, thus constraining links to directories to those made by *mkdir()*.

The specification that if *old* and *new* refer to the same file is intended to guarantee that:

```
rename("x", "x");
```

does not remove the file.

Renaming dot or dot-dot is prohibited in order to prevent cyclical file system paths.

See also the descriptions of [ENOTEMPTY] and [ENAMETOOLONG] in *rmdir()* and [EBUSY] in *unlink()*. For a discussion of [EXDEV], see *link()*.

The purpose of the *renameat()* function is to rename files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *rename()*, resulting in unspecified behavior. By opening file descriptors for the source and target directories and using the *renameat()* function it can be guaranteed that that renamed file is located correctly and the resulting file is in the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

link(), *rmdir()*, *symlink()*, *unlink()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>

CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 5

The [EBUSY] error is added to the optional part of the ERRORS section.

Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] mandatory error condition is added.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.
- The [ETXTBSY] optional error condition is added.

The following changes were made to align with the IEEE P1003.1a draft standard:

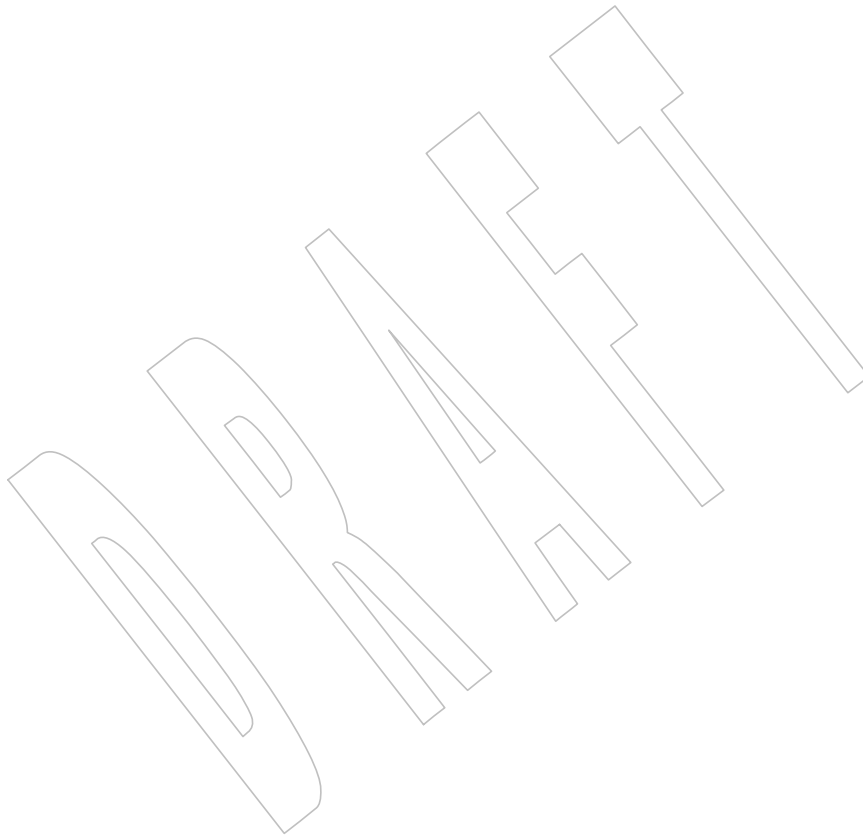
- Details are added regarding the treatment of symbolic links.
- The [ELOOP] optional error condition is added.

The normative text is updated to avoid use of the term “must” for application requirements.

rename()40189
40190
40191
40192
40193**Issue 7**

Austin Group Interpretation 1003.1-2001 #076 is applied, clarifying the behavior if the final component of a path is either dot or dot-dot, and adding the associated [EINVAL] error case.

The *renameat()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.



40194 **NAME**
40195 renameat — rename file relative to directory file descriptor

40196 **SYNOPSIS**

```
40197 CX #include <stdio.h>  
40198 int renameat(int oldfd, const char *old, int newfd,  
40199 const char *new);
```

40200 **DESCRIPTION**

40201 Refer to *rename()*.

40202 **NAME**40203 `rewind` — reset the file position indicator in a stream40204 **SYNOPSIS**40205 `#include <stdio.h>`40206 `void rewind(FILE *stream);`40207 **DESCRIPTION**40208 CX The functionality described on this reference page is aligned with the ISO C standard. Any
40209 conflict between the requirements described here and the ISO C standard is unintentional. This
40210 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

40211 The call:

40212 `rewind(stream)`

40213 shall be equivalent to:

40214 `(void) fseek(stream, 0L, SEEK_SET)`40215 except that *rewind()* shall also clear the error indicator.40216 CX Since *rewind()* does not return a value, an application wishing to detect errors should clear *errno*,
40217 then call *rewind()*, and if *errno* is non-zero, assume an error has occurred.40218 **RETURN VALUE**40219 The *rewind()* function shall not return a value.40220 **ERRORS**40221 CX Refer to *fseek()* with the exception of [EINVAL] which does not apply.40222 **EXAMPLES**

40223 None.

40224 **APPLICATION USAGE**

40225 None.

40226 **RATIONALE**

40227 None.

40228 **FUTURE DIRECTIONS**

40229 None.

40230 **SEE ALSO**40231 *fseek()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdio.h>`40232 **CHANGE HISTORY**

40233 First released in Issue 1. Derived from Issue 1 of the SVID.

40234 **Issue 6**

40235 Extensions beyond the ISO C standard are marked.

NAME

rewinddir — reset the position of a directory stream to the beginning of a directory

SYNOPSIS

```
#include <dirent.h>

void rewinddir(DIR *dirp);
```

DESCRIPTION

The *rewinddir()* function shall reset the position of the directory stream to which *dirp* refers to the beginning of the directory. It shall also cause the directory stream to refer to the current state of the corresponding directory, as a call to *opendir()* would have done. If *dirp* does not refer to a directory stream, the effect is undefined.

After a call to the *fork()* function, either the parent or child (but not both) may continue processing the directory stream using *readdir()*, *rewinddir()*, or *seekdir()*. If both the parent and child processes use these functions, the result is undefined.

RETURN VALUE

The *rewinddir()* function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *rewinddir()* function should be used in conjunction with *opendir()*, *readdir()*, and *closedir()* to examine the contents of the directory. This method is recommended for portability.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

closedir(), *fdopendir()*, *readdir()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>** **<sys/types.h>**

CHANGE HISTORY

First released in Issue 2.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required

40274 **NAME**
 40275 `rint`, `rintf`, `rintl` — round-to-nearest integral value

40276 **SYNOPSIS**
 40277 `#include <math.h>`
 40278 `double rint(double x);`
 40279 `float rintf(float x);`
 40280 `long double rintl(long double x);`

40281 DESCRIPTION

40282 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 40283 conflict between the requirements described here and the ISO C standard is unintentional. This
 40284 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

40285 These functions shall return the integral value (represented as a **double**) nearest x in the
 40286 direction of the current rounding mode. The current rounding mode is implementation-defined.

40287 If the current rounding mode rounds toward negative infinity, then `rint()` shall be equivalent to
 40288 `floor()`. If the current rounding mode rounds toward positive infinity, then `rint()` shall be
 40289 equivalent to `ceil()`.

40290 These functions differ from the `nearbyint()`, `nearbyintf()`, and `nearbyintl()` functions only in that
 40291 they may raise the inexact floating-point exception if the result differs in value from the
 40292 argument.

40293 An application wishing to check for error situations should set `errno` to zero and call
 40294 `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `errno` is non-zero or
 40295 `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-
 40296 zero, an error has occurred.

40297 RETURN VALUE

40298 Upon successful completion, these functions shall return the integer (represented as a double
 40299 precision number) nearest x in the direction of the current rounding mode.

40300 MX If x is NaN, a NaN shall be returned.

40301 If x is ± 0 or $\pm \text{Inf}$, x shall be returned.

40302 XSI If the correct value would cause overflow, a range error shall occur and `rint()`, `rintf()`, and `rintl()`
 40303 shall return the value of the macro `±HUGE_VAL`, `±HUGE_VALF`, and `±HUGE_VALL` (with the
 40304 same sign as x), respectively.

40305 ERRORS

40306 These functions shall fail if:

40307 XSI **Range Error** The result would cause an overflow.

40308 If the integer expression `(math_errhandling & MATH_ERRNO)` is non-zero,
 40309 then `errno` shall be set to [ERANGE]. If the integer expression
 40310 `(math_errhandling & MATH_ERREXCEPT)` is non-zero, then the overflow
 40311 floating-point exception shall be raised.

40312
40313
40314
40315
40316
40317
40318
40319
40320
40321
40322
40323
40324
40325
40326
40327
40328
40329
40330
40331
40332
40333
40334
40335
40336

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

abs(), *ceil()*, *feclearexcept()*, *fetestexcept()*, *floor()*, *isnan()*, *nearbyint()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The *rintf()* and *rintl()* functions are added.
- The *rint()* function is no longer marked as an extension.
- The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

40337 **NAME**40338 `rmdir` — remove a directory40339 **SYNOPSIS**40340 `#include <unistd.h>`40341 `int rmdir(const char *path);`40342 **DESCRIPTION**40343 The `rmdir()` function shall remove a directory whose name is given by *path*. The directory shall
40344 be removed only if it is an empty directory.40345 If the directory is the root directory or the current working directory of any process, it is
40346 unspecified whether the function succeeds, or whether it shall fail and set *errno* to [EBUSY].40347 If *path* names a symbolic link, then `rmdir()` shall fail and set *errno* to [ENOTDIR].40348 If the *path* argument refers to a path whose final component is either dot or dot-dot, `rmdir()` shall
40349 fail.40350 If the directory's link count becomes 0 and no process has the directory open, the space occupied
40351 by the directory shall be freed and the directory shall no longer be accessible. If one or more
40352 processes have the directory open when the last link is removed, the dot and dot-dot entries, if
40353 present, shall be removed before `rmdir()` returns and no new entries may be created in the
40354 directory, but the directory shall not be removed until all references to the directory are closed.40355 If the directory is not an empty directory, `rmdir()` shall fail and set *errno* to [EEXIST] or
40356 [ENOTEMPTY].40357 Upon successful completion, the `rmdir()` function shall mark for update the *st_ctime* and
40358 *st_mtime* fields of the parent directory.40359 **RETURN VALUE**40360 Upon successful completion, the function `rmdir()` shall return 0. Otherwise, -1 shall be returned,
40361 and *errno* set to indicate the error. If -1 is returned, the named directory shall not be changed.40362 **ERRORS**40363 The `rmdir()` function shall fail if:40364 [EACCES] Search permission is denied on a component of the path prefix, or write
40365 permission is denied on the parent directory of the directory to be removed.40366 [EBUSY] The directory to be removed is currently in use by the system or some process
40367 and the implementation considers this to be an error.40368 [EEXIST] or [ENOTEMPTY] The *path* argument names a directory that is not an empty directory, or there
40369 are hard links to the directory other than dot or a single entry in dot-dot.
4037040371 [EINVAL] The *path* argument contains a last component that is dot.

40372 [EIO] A physical I/O error has occurred.

40373 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
40374 argument.40375 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname
40376 component is longer than {NAME_MAX}.
40377

- 40378 [ENOENT] A component of *path* does not name an existing file, or the *path* argument
40379 names a nonexistent directory or points to an empty string.
- 40380 [ENOTDIR] A component of *path* is not a directory.
- 40381 XSI [EPERM] or [EACCES]
40382 The S_ISVTX flag is set on the parent directory of the directory to be removed
40383 and the caller is not the owner of the directory to be removed, nor is the caller
40384 the owner of the parent directory, nor does the caller have the appropriate
40385 privileges.
- 40386 [EROFS] The directory entry to be removed resides on a read-only file system.
- 40387 The *rmdir()* function may fail if:
- 40388 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
40389 resolution of the *path* argument.
- 40390 [ENAMETOOLONG]
40391 As a result of encountering a symbolic link in resolution of the *path* argument,
40392 the length of the substituted pathname string exceeded {PATH_MAX}.

EXAMPLES

Removing a Directory

The following example shows how to remove a directory named `/home/cnd/mod1`.

```
#include <unistd.h>

int status;
...
status = rmdir("/home/cnd/mod1");
```

APPLICATION USAGE

None.

RATIONALE

The *rmdir()* and *rename()* functions originated in 4.2 BSD, and they used [ENOTEMPTY] for the condition when the directory to be removed does not exist or *new* already exists. When the 1984 /usr/group standard was published, it contained [EEXIST] instead. When these functions were adopted into System V, the 1984 /usr/group standard was used as a reference. Therefore, several existing applications and implementations support/use both forms, and no agreement could be reached on either value. All implementations are required to supply both [EEXIST] and [ENOTEMPTY] in `<errno.h>` with distinct values, so that applications can use both values in C-language **case** statements.

The meaning of deleting *pathname/dot* is unclear, because the name of the file (directory) in the parent directory to be removed is not clear, particularly in the presence of multiple links to a directory.

The POSIX.1-1990 standard was silent with regard to the behavior of *rmdir()* when there are multiple hard links to the directory being removed. The requirement to set *errno* to [EEXIST] or [ENOTEMPTY] clarifies the behavior in this case.

If the current working directory of the process is being removed, that should be an allowed error.

Virtually all existing implementations detect [ENOTEMPTY] or the case of dot-dot. The text in [Section 2.3](#) about returning any one of the possible errors permits that behavior to continue. The [ELOOP] error may be returned if more than {SYMLOOP_MAX} symbolic links are encountered during resolution of the *path* argument.

40423
40424
40425
40426
40427
40428
40429
40430
40431
40432
40433
40434
40435
40436
40437
40438**FUTURE DIRECTIONS**

None.

SEE ALSO

Section 2.3 (on page 21), *mkdir()*, *remove()*, *rename()*, *unlink()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**unistd.h**>

CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to indicate the results of naming a symbolic link in *path*.
- The [EIO] mandatory error condition is added.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

40439 **NAME**

40440 round, roundf, roundl — round to the nearest integer value in a floating-point format

40441 **SYNOPSIS**

```
40442 #include <math.h>
40443
40443 double round(double x);
40444 float roundf(float x);
40445 long double roundl(long double x);
```

40446 **DESCRIPTION**

40447 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 40448 conflict between the requirements described here and the ISO C standard is unintentional. This
 40449 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

40450 These functions shall round their argument to the nearest integer value in floating-point format,
 40451 rounding halfway cases away from zero, regardless of the current rounding direction.

40452 An application wishing to check for error situations should set *errno* to zero and call
 40453 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 40454 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 40455 zero, an error has occurred.

40456 **RETURN VALUE**

40457 Upon successful completion, these functions shall return the rounded integer value.

40458 MX If *x* is NaN, a NaN shall be returned.40459 If *x* is ± 0 or $\pm \text{Inf}$, *x* shall be returned.

40460 XSI If the correct value would cause overflow, a range error shall occur and *round()*, *roundf()*, and
 40461 *roundl()* shall return the value of the macro $\pm \text{HUGE_VAL}$, $\pm \text{HUGE_VALF}$, and $\pm \text{HUGE_VALL}$
 40462 (with the same sign as *x*), respectively.

40463 **ERRORS**

40464 These functions may fail if:

40465 XSI **Range Error** The result overflows.

40466 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 40467 then *errno* shall be set to [ERANGE]. If the integer expression
 40468 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
 40469 floating-point exception shall be raised.

40470 **EXAMPLES**

40471 None.

40472 **APPLICATION USAGE**

40473 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
 40474 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

40475 **RATIONALE**

40476 None.

40477 **FUTURE DIRECTIONS**

40478 None.

round()*System Interfaces*

40479

SEE ALSO

40480

feclearexcept(), *fetestexcept()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

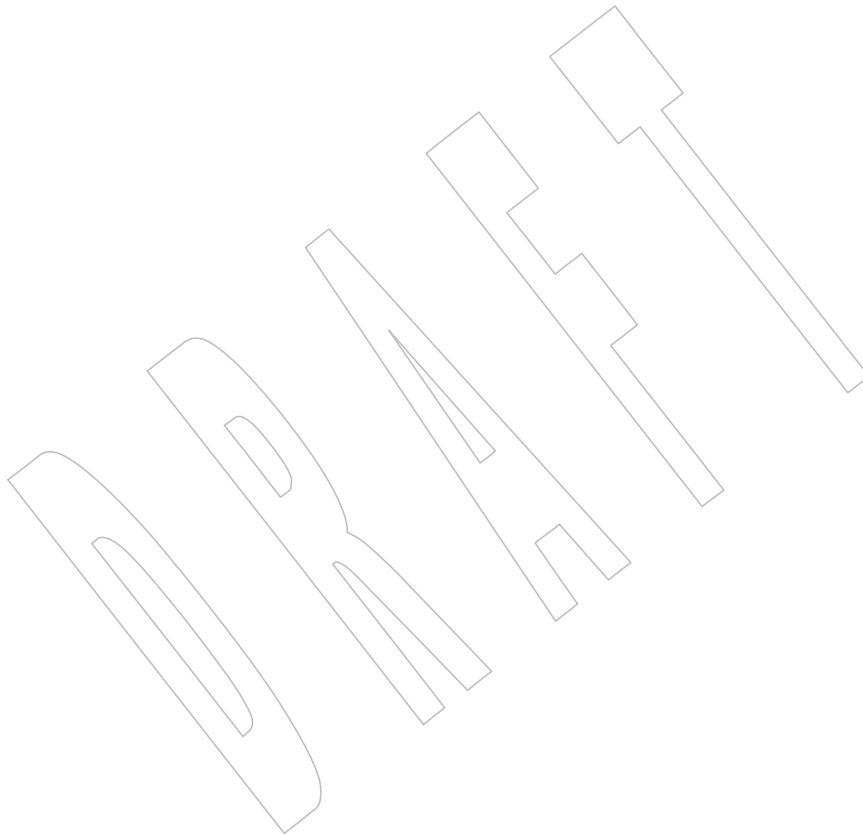
40481

40482

CHANGE HISTORY

40483

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.



40484 **NAME**

40485 scalbln, scalblnf, scalblnl, scalbn, scalbnf, scalbnl — compute exponent using FLT_RADIX

40486 **SYNOPSIS**

40487 #include <math.h>

40488 double scalbln(double x, long n);

40489 float scalblnf(float x, long n);

40490 long double scalblnl(long double x, long n);

40491 double scalbn(double x, int n);

40492 float scalbnf(float x, int n);

40493 long double scalbnl(long double x, int n);

40494 **DESCRIPTION**

40495 CX The functionality described on this reference page is aligned with the ISO C standard. Any

40496 conflict between the requirements described here and the ISO C standard is unintentional. This

40497 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

40498 These functions shall compute $x * FLT_RADIX^n$ efficiently, not normally by computing

40499 FLT_RADIX^n explicitly.

40500 An application wishing to check for error situations should set *errno* to zero and call

40501 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or

40502 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-

40503 zero, an error has occurred.

40504 **RETURN VALUE**

40505 Upon successful completion, these functions shall return $x * FLT_RADIX^n$.

40506 If the result would cause overflow, a range error shall occur and these functions shall return

40507 $\pm HUGE_VAL$, $\pm HUGE_VALF$, and $\pm HUGE_VALL$ (according to the sign of *x*) as appropriate for

40508 the return type of the function.

40509 If the correct value would cause underflow, and is not representable, a range error may occur,

40510 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

40511 MX If *x* is NaN, a NaN shall be returned.

40512 If *x* is ± 0 or $\pm Inf$, *x* shall be returned.

40513 If *n* is 0, *x* shall be returned.

40514 If the correct value would cause underflow, and is representable, a range error may occur and

40515 the correct value shall be returned.

40516 **ERRORS**

40517 These functions shall fail if:

40518 Range Error The result overflows.

40519 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,

40520 then *errno* shall be set to [ERANGE]. If the integer expression

40521 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow

40522 floating-point exception shall be raised.

40523 These functions may fail if:

40524 Range Error The result underflows.

40525 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,

40526 then *errno* shall be set to [ERANGE]. If the integer expression

40527 *(math_errhandling & MATH_ERREXCEPT)* is non-zero, then the underflow
40528 floating-point exception shall be raised.

EXAMPLES

40529 None.
40530

APPLICATION USAGE

40531 On error, the expressions *(math_errhandling & MATH_ERRNO)* and *(math_errhandling &*
40532 *MATH_ERREXCEPT)* are independent of each other, but at least one of them must be non-zero.
40533

RATIONALE

40534 These functions are named so as to avoid conflicting with the historical definition of the *scalb()*
40535 function from the Single UNIX Specification. The difference is that the *scalb()* function has a
40536 second argument of **double** instead of **int**. The *scalb()* function is not part of the ISO C standard.
40537 The three functions whose second type is **long** are provided because the factor required to scale
40538 from the smallest positive floating-point value to the largest finite one, on many
40539 implementations, is too large to represent in the minimum-width **int** format.
40540

FUTURE DIRECTIONS

40541 None.
40542

SEE ALSO

40543 *feclearexcept()*, *fetestexcept()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18,
40544 Treatment of Error Conditions for Mathematical Functions, **<math.h>**
40545

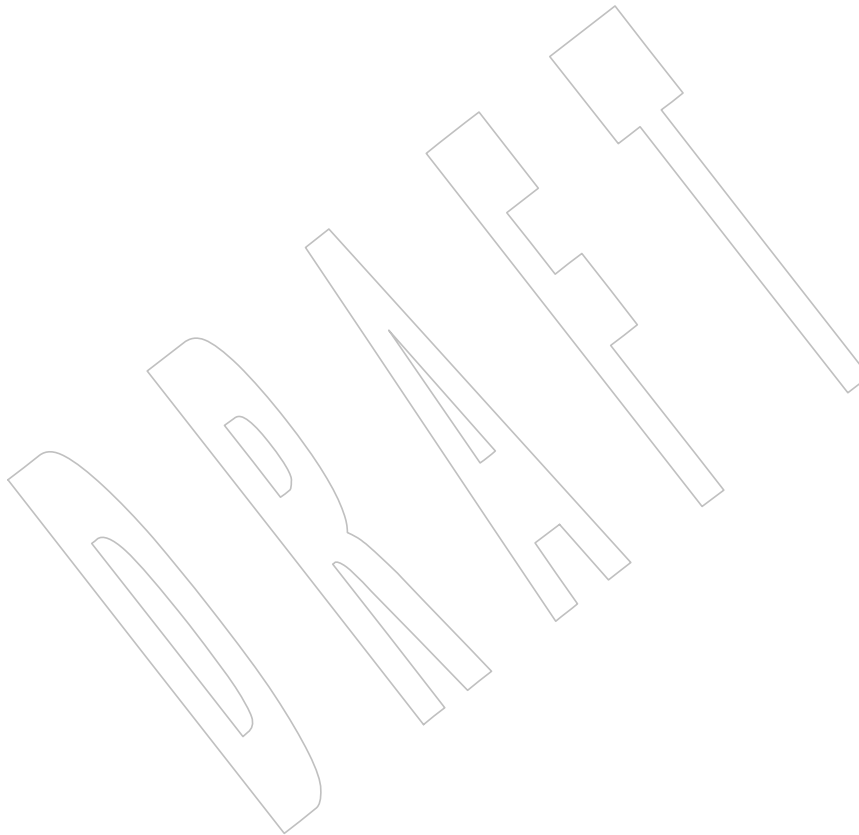
CHANGE HISTORY

40546 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.
40547

40548 **NAME**
40549 scandir — scan a directory

40550 **SYNOPSIS**
40551 #include <dirent.h>
40552 int scandir(const char *dir, struct dirent ***namelist,
40553 int (*sel)(const struct dirent *),
40554 int (*compar)(const struct dirent **, const struct dirent **));

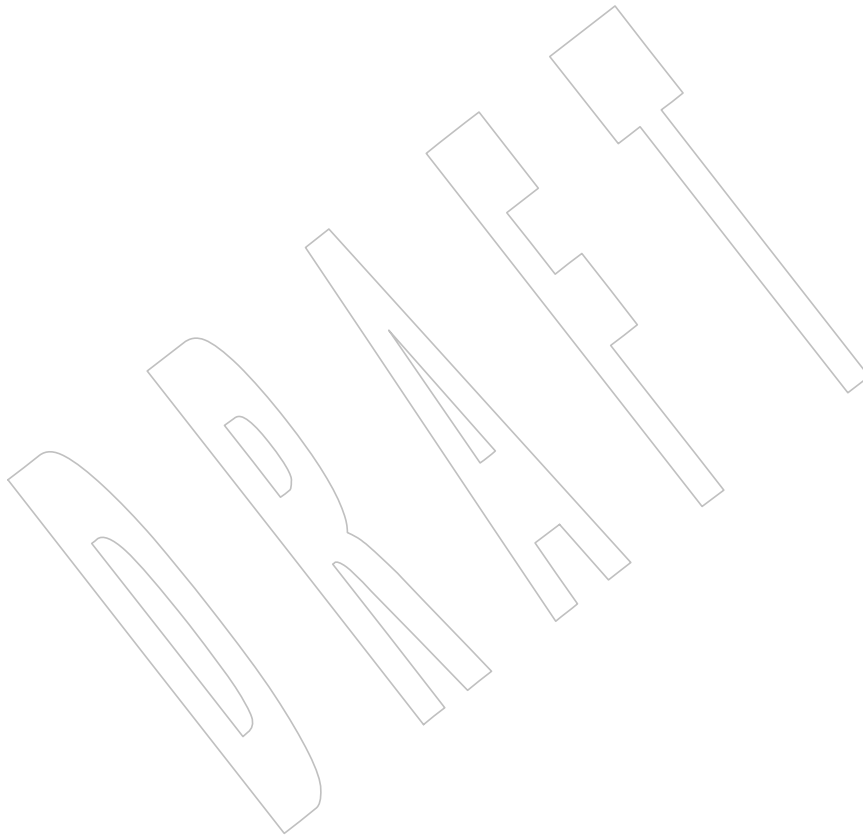
40555 **DESCRIPTION**
40556 Refer to *alphasort()*.



40557 **NAME**
40558 scanf — convert formatted input

40559 **SYNOPSIS**
40560 #include <stdio.h>
40561 int scanf(const char *restrict *format*, ...);

40562 **DESCRIPTION**
40563 Refer to *fscanf()*.



40564 **NAME**40565 sched_get_priority_max, sched_get_priority_min — get priority limits (**REALTIME**)40566 **SYNOPSIS**

```
40567 PS|TPS #include <sched.h>
40568 int sched_get_priority_max(int policy);
40569 int sched_get_priority_min(int policy);
```

40570 **DESCRIPTION**

40571 The *sched_get_priority_max()* and *sched_get_priority_min()* functions shall return the appropriate
 40572 maximum or minimum, respectively, for the scheduling policy specified by *policy*.

40573 The value of *policy* shall be one of the scheduling policy values defined in **<sched.h>**.

40574 **RETURN VALUE**

40575 If successful, the *sched_get_priority_max()* and *sched_get_priority_min()* functions shall return the
 40576 appropriate maximum or minimum values, respectively. If unsuccessful, they shall return a
 40577 value of -1 and set *errno* to indicate the error.

40578 **ERRORS**

40579 The *sched_get_priority_max()* and *sched_get_priority_min()* functions shall fail if:

40580 [EINVAL] The value of the *policy* parameter does not represent a defined scheduling
 40581 policy.

40582 **EXAMPLES**

40583 None.

40584 **APPLICATION USAGE**

40585 None.

40586 **RATIONALE**

40587 None.

40588 **FUTURE DIRECTIONS**

40589 None.

40590 **SEE ALSO**

40591 *sched_getparam()*, *sched_setparam()*, *sched_getscheduler()*, *sched_rr_get_interval()*,
 40592 *sched_setscheduler()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**

40593 **CHANGE HISTORY**

40594 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

40595 **Issue 6**

40596 These functions are marked as part of the Process Scheduling option.

40597 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 40598 implementation does not support the Process Scheduling option.

40599 The [ESRCH] error condition has been removed since these functions do not take a *pid*
 40600 argument.

40601 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/52 is applied, changing the PS margin
 40602 code in the SYNOPSIS to PS|TPS.

40603 **NAME**40604 sched_getparam — get scheduling parameters (**REALTIME**)40605 **SYNOPSIS**

```
40606 PS #include <sched.h>
40607 int sched_getparam(pid_t pid, struct sched_param *param);
```

40608 **DESCRIPTION**

40609 The *sched_getparam()* function shall return the scheduling parameters of a process specified by
 40610 *pid* in the **sched_param** structure pointed to by *param*.

40611 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
 40612 parameters for the process whose process ID is equal to *pid* shall be returned.

40613 If *pid* is zero, the scheduling parameters for the calling process shall be returned. The behavior of
 40614 the *sched_getparam()* function is unspecified if the value of *pid* is negative.

40615 **RETURN VALUE**

40616 Upon successful completion, the *sched_getparam()* function shall return zero. If the call to
 40617 *sched_getparam()* is unsuccessful, the function shall return a value of -1 and set *errno* to indicate
 40618 the error.

40619 **ERRORS**

40620 The *sched_getparam()* function shall fail if:

40621 [EPERM] The requesting process does not have permission to obtain the scheduling
 40622 parameters of the specified process.

40623 [ESRCH] No process can be found corresponding to that specified by *pid*.

40624 **EXAMPLES**

40625 None.

40626 **APPLICATION USAGE**

40627 None.

40628 **RATIONALE**

40629 None.

40630 **FUTURE DIRECTIONS**

40631 None.

40632 **SEE ALSO**

40633 [sched_getscheduler\(\)](#), [sched_setparam\(\)](#), [sched_setscheduler\(\)](#), the Base Definitions volume of
 40634 IEEE Std 1003.1-200x, **<sched.h>**

40635 **CHANGE HISTORY**

40636 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

40637 **Issue 6**

40638 The *sched_getparam()* function is marked as part of the Process Scheduling option.

40639 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 40640 implementation does not support the Process Scheduling option.

40641 **NAME**
 40642 sched_getscheduler — get scheduling policy (**REALTIME**)

40643 **SYNOPSIS**

```
40644 PS #include <sched.h>
40645 int sched_getscheduler(pid_t pid);
```

40646 **DESCRIPTION**

40647 The *sched_getscheduler()* function shall return the scheduling policy of the process specified by
 40648 *pid*. If the value of *pid* is negative, the behavior of the *sched_getscheduler()* function is
 40649 unspecified.

40650 The values that can be returned by *sched_getscheduler()* are defined in the **<sched.h>** header.

40651 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
 40652 policy shall be returned for the process whose process ID is equal to *pid*.

40653 If *pid* is zero, the scheduling policy shall be returned for the calling process.

40654 **RETURN VALUE**

40655 Upon successful completion, the *sched_getscheduler()* function shall return the scheduling policy
 40656 of the specified process. If unsuccessful, the function shall return -1 and set *errno* to indicate the
 40657 error.

40658 **ERRORS**

40659 The *sched_getscheduler()* function shall fail if:

- | | | |
|-------|---------|---|
| 40660 | [EPERM] | The requesting process does not have permission to determine the scheduling |
| 40661 | | policy of the specified process. |
| 40662 | [ESRCH] | No process can be found corresponding to that specified by <i>pid</i> . |

40663 **EXAMPLES**

40664 None.

40665 **APPLICATION USAGE**

40666 None.

40667 **RATIONALE**

40668 None.

40669 **FUTURE DIRECTIONS**

40670 None.

40671 **SEE ALSO**

40672 *sched_getparam()*, *sched_setparam()*, *sched_setscheduler()*, the Base Definitions volume of
 40673 IEEE Std 1003.1-200x, **<sched.h>**

40674 **CHANGE HISTORY**

40675 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

40676 **Issue 6**

40677 The *sched_getscheduler()* function is marked as part of the Process Scheduling option.

40678 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 40679 implementation does not support the Process Scheduling option.

40680 **NAME**40681 sched_rr_get_interval — get execution time limits (**REALTIME**)40682 **SYNOPSIS**

40683 PS|TPS #include <sched.h>

40684 int sched_rr_get_interval(pid_t pid, struct timespec *interval);

40685 **DESCRIPTION**

40686 The *sched_rr_get_interval()* function shall update the **timespec** structure referenced by the
 40687 *interval* argument to contain the current execution time limit (that is, time quantum) for the
 40688 process specified by *pid*. If *pid* is zero, the current execution time limit for the calling process
 40689 shall be returned.

40690 **RETURN VALUE**

40691 If successful, the *sched_rr_get_interval()* function shall return zero. Otherwise, it shall return a
 40692 value of -1 and set *errno* to indicate the error.

40693 **ERRORS**40694 The *sched_rr_get_interval()* function shall fail if:40695 [ESRCH] No process can be found corresponding to that specified by *pid*.40696 **EXAMPLES**

40697 None.

40698 **APPLICATION USAGE**

40699 None.

40700 **RATIONALE**

40701 None.

40702 **FUTURE DIRECTIONS**

40703 None.

40704 **SEE ALSO**

40705 *sched_getparam()*, *sched_get_priority_max()*, *sched_getscheduler()*, *sched_setparam()*,
 40706 *sched_setscheduler()*, the Base Definitions volume of IEEE Std 1003.1-200x, <sched.h>

40707 **CHANGE HISTORY**

40708 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

40709 **Issue 6**40710 The *sched_rr_get_interval()* function is marked as part of the Process Scheduling option.

40711 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 40712 implementation does not support the Process Scheduling option.

40713 IEEE Std 1003.1-2001/Cor 1-2002, XSH/TC1/D6/53 is applied, changing the PS margin code in
 40714 the SYNOPSIS to PS|TPS.

40715 **NAME**40716 sched_setparam — set scheduling parameters (**REALTIME**)40717 **SYNOPSIS**

```
40718 PS #include <sched.h>
40719 int sched_setparam(pid_t pid, const struct sched_param *param);
```

40720 **DESCRIPTION**

40721 The *sched_setparam()* function shall set the scheduling parameters of the process specified by *pid*
 40722 to the values specified by the **sched_param** structure pointed to by *param*. The value of the
 40723 *sched_priority* member in the **sched_param** structure shall be any integer within the inclusive
 40724 priority range for the current scheduling policy of the process specified by *pid*. Higher
 40725 numerical values for the priority represent higher priorities. If the value of *pid* is negative, the
 40726 behavior of the *sched_setparam()* function is unspecified.

40727 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
 40728 parameters shall be set for the process whose process ID is equal to *pid*.

40729 If *pid* is zero, the scheduling parameters shall be set for the calling process.

40730 The conditions under which one process has permission to change the scheduling parameters of
 40731 another process are implementation-defined.

40732 Implementations may require the requesting process to have the appropriate privilege to set its
 40733 own scheduling parameters or those of another process.

40734 See [Scheduling Policies](#) for a description on how this function affects the scheduling of the
 40735 threads within the target process.

40736 SS If the current scheduling policy for the target process is not SCHED_FIFO, SCHED_RR, or
 40737 SCHED_SPORADIC, the result is implementation-defined; this case includes the
 40738 SCHED_OTHER policy.

40739 SS The specified *sched_ss_repl_period* shall be greater than or equal to the specified
 40740 *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.

40741 The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the
 40742 function to succeed; if not, the function shall fail.

40743 This function is not atomic with respect to other threads in the process. Threads may continue to
 40744 execute while this function call is in the process of changing the scheduling policy for the
 40745 underlying kernel-scheduled entities used by the process contention scope threads.

40746 **RETURN VALUE**

40747 If successful, the *sched_setparam()* function shall return zero.

40748 If the call to *sched_setparam()* is unsuccessful, the priority shall remain unchanged, and the
 40749 function shall return a value of -1 and set *errno* to indicate the error.

40750 **ERRORS**

40751 The *sched_setparam()* function shall fail if:

40752 [EINVAL] One or more of the requested scheduling parameters is outside the range
 40753 defined for the scheduling policy of the specified *pid*.

40754 [EPERM] The requesting process does not have permission to set the scheduling
 40755 parameters for the specified process, or does not have the appropriate
 40756 privilege to invoke *sched_setparam()*.

sched_setparam()

40757 [ESRCH] No process can be found corresponding to that specified by *pid*.

EXAMPLES

40758 None.
40759

APPLICATION USAGE

40760 None.
40761

RATIONALE

40762 None.
40763

FUTURE DIRECTIONS

40764 None.
40765

SEE ALSO

40766 [Scheduling Policies](#) (on page 44), [sched_getparam\(\)](#), [sched_getscheduler\(\)](#), [sched_setscheduler\(\)](#), the
40767 Base Definitions volume of IEEE Std 1003.1-200x, <[sched.h](#)>
40768

CHANGE HISTORY

40769 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.
40770

Issue 6

40771 The `sched_setparam()` function is marked as part of the Process Scheduling option.
40772

40773 The [ENOSYS] error condition has been removed as stubs need not be provided if an
40774 implementation does not support the Process Scheduling option.

40775 The following new requirements on POSIX implementations derive from alignment with the
40776 Single UNIX Specification:

- 40777 • In the DESCRIPTION, the effect of this function on a thread's scheduling parameters is
40778 added.
- 40779 • Sections describing two-level scheduling and atomicity of the function are added.

40780 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

40781 IEEE PASC Interpretation 1003.1 #100 is applied.

Issue 7

40782 Austin Group Interpretation 1003.1-2001 #061 is applied, updating the DESCRIPTION.
40783

40784 **NAME**
 40785 sched_setscheduler — set scheduling policy and parameters (**REALTIME**)

40786 **SYNOPSIS**

```
40787 PS #include <sched.h>
40788 int sched_setscheduler(pid_t pid, int policy,
40789 const struct sched_param *param);
```

40790 **DESCRIPTION**

40791 The *sched_setscheduler()* function shall set the scheduling policy and scheduling parameters of
 40792 the process specified by *pid* to *policy* and the parameters specified in the **sched_param** structure
 40793 pointed to by *param*, respectively. The value of the *sched_priority* member in the **sched_param**
 40794 structure shall be any integer within the inclusive priority range for the scheduling policy
 40795 specified by *policy*. If the value of *pid* is negative, the behavior of the *sched_setscheduler()*
 40796 function is unspecified.

40797 The possible values for the *policy* parameter are defined in the **<sched.h>** header.

40798 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
 40799 policy and scheduling parameters shall be set for the process whose process ID is equal to *pid*.

40800 If *pid* is zero, the scheduling policy and scheduling parameters shall be set for the calling
 40801 process.

40802 The conditions under which one process has the appropriate privilege to change the scheduling
 40803 parameters of another process are implementation-defined.

40804 Implementations may require that the requesting process have permission to set its own
 40805 scheduling parameters or those of another process. Additionally, implementation-defined
 40806 restrictions may apply as to the appropriate privileges required to set the scheduling policy of
 40807 the process, or the scheduling policy of another process, to a particular value.

40808 The *sched_setscheduler()* function shall be considered successful if it succeeds in setting the
 40809 scheduling policy and scheduling parameters of the process specified by *pid* to the values
 40810 specified by *policy* and the structure pointed to by *param*, respectively.

40811 See [Scheduling Policies](#) for a description on how this function affects the scheduling of the
 40812 threads within the target process.

40813 SS If the current scheduling policy for the target process is not SCHED_FIFO, SCHED_RR, or
 40814 SCHED_SPORADIC, the result is implementation-defined; this case includes the
 40815 SCHED_OTHER policy.

40816 SS The specified *sched_ss_repl_period* shall be greater than or equal to the specified
 40817 *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.

40818 The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the
 40819 function to succeed; if not, the function shall fail.

40820 This function is not atomic with respect to other threads in the process. Threads may continue to
 40821 execute while this function call is in the process of changing the scheduling policy and
 40822 associated scheduling parameters for the underlying kernel-scheduled entities used by the
 40823 process contention scope threads.

sched_setscheduler()*System Interfaces***RETURN VALUE**

40824
40825 Upon successful completion, the function shall return the former scheduling policy of the
40826 specified process. If the *sched_setscheduler()* function fails to complete successfully, the policy
40827 and scheduling parameters shall remain unchanged, and the function shall return a value of -1
40828 and set *errno* to indicate the error.

ERRORS

40829 The *sched_setscheduler()* function shall fail if:

- 40831 [EINVAL] The value of the *policy* parameter is invalid, or one or more of the parameters
40832 contained in *param* is outside the valid range for the specified scheduling
40833 policy.
- 40834 [EPERM] The requesting process does not have permission to set either or both of the
40835 scheduling parameters or the scheduling policy of the specified process.
- 40836 [ESRCH] No process can be found corresponding to that specified by *pid*.

EXAMPLES

40837 None.
40838

APPLICATION USAGE

40839 None.
40840

RATIONALE

40841 None.
40842

FUTURE DIRECTIONS

40843 None.
40844

SEE ALSO

40845 [Scheduling Policies](#) (on page 44), [sched_getparam\(\)](#), [sched_getscheduler\(\)](#), [sched_setparam\(\)](#), the
40846 Base Definitions volume of IEEE Std 1003.1-200x, [<sched.h>](#)
40847

CHANGE HISTORY

40848 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.
40849

Issue 6

40850 The *sched_setscheduler()* function is marked as part of the Process Scheduling option.
40851

40852 The [ENOSYS] error condition has been removed as stubs need not be provided if an
40853 implementation does not support the Process Scheduling option.

40854 The following new requirements on POSIX implementations derive from alignment with the
40855 Single UNIX Specification:

- 40856 • In the DESCRIPTION, the effect of this function on a thread's scheduling parameters is
40857 added.
- 40858 • Sections describing two-level scheduling and atomicity of the function are added.

40859 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.
40860

Issue 7

40861 Austin Group Interpretation 1003.1-2001 #061 is applied, updating the DESCRIPTION.

NAME

sched_yield — yield the processor

SYNOPSIS

```
#include <sched.h>

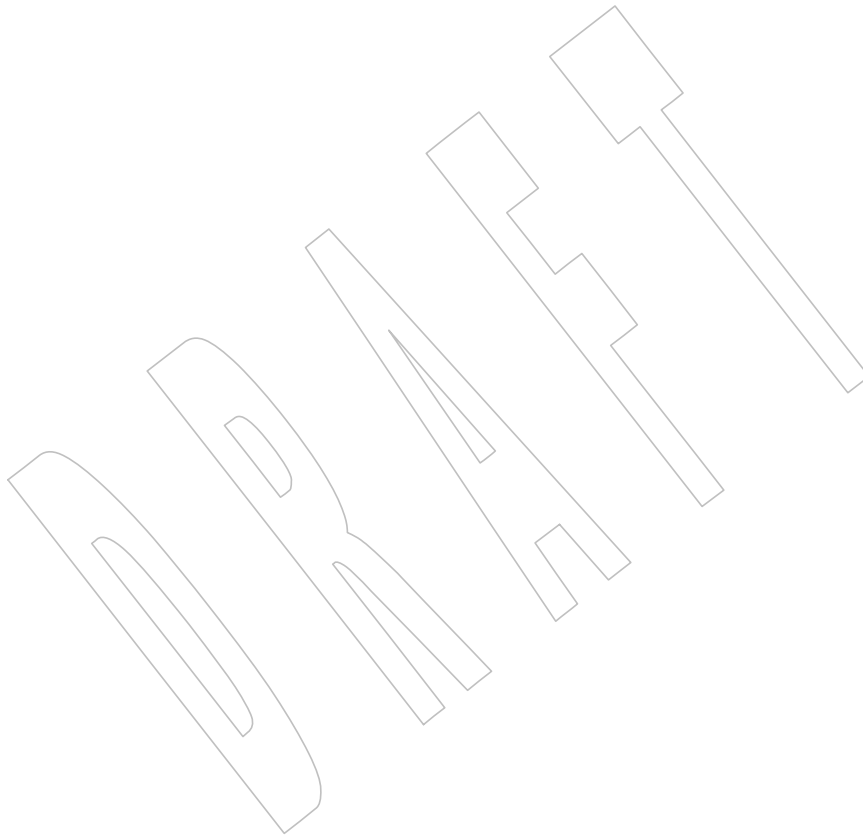
int sched_yield(void);
```

DESCRIPTION

The *sched_yield()* function shall force the running thread to relinquish the processor until it again becomes the head of its thread list. It takes no arguments.

RETURN VALUE

The *sched_yield()* function shall return 0 if it completes successfully; otherwise, it shall return a value $\neq 0$.



seed48()

40897 **NAME**
40898 seed48 — seed a uniformly distributed pseudo-random non-negative long integer generator

SYNOPSIS

40900 XSI #include <stdlib.h>
40901 unsigned short *seed48(unsigned short seed16v[3]);

DESCRIPTION

40902 Refer to *drand48()*.
40903

40904 **NAME**
 40905 seekdir — set the position of a directory stream

40906 **SYNOPSIS**

```
40907 XSI #include <dirent.h>
40908 void seekdir(DIR *dirp, long loc);
```

40909 **DESCRIPTION**

40910 The *seekdir()* function shall set the position of the next *readdir()* operation on the directory
 40911 stream specified by *dirp* to the position specified by *loc*. The value of *loc* should have been
 40912 returned from an earlier call to *telldir()*. The new position reverts to the one associated with the
 40913 directory stream when *telldir()* was performed.

40914 If the value of *loc* was not obtained from an earlier call to *telldir()*, or if a call to *rewinddir()*
 40915 occurred between the call to *telldir()* and the call to *seekdir()*, the results of subsequent calls to
 40916 *readdir()* are unspecified.

40917 **RETURN VALUE**

40918 The *seekdir()* function shall not return a value.

40919 **ERRORS**

40920 No errors are defined.

40921 **EXAMPLES**

40922 None.

40923 **APPLICATION USAGE**

40924 None.

40925 **RATIONALE**

40926 The original standard developers perceived that there were restrictions on the use of the
 40927 *seekdir()* and *telldir()* functions related to implementation details, and for that reason these
 40928 functions need not be supported on all POSIX-conforming systems. They are required on
 40929 implementations supporting the XSI option.

40930 One of the perceived problems of implementation is that returning to a given point in a directory
 40931 is quite difficult to describe formally, in spite of its intuitive appeal, when systems that use B-
 40932 trees, hashing functions, or other similar mechanisms to order their directories are considered.
 40933 The definition of *seekdir()* and *telldir()* does not specify whether, when using these interfaces, a
 40934 given directory entry will be seen at all, or more than once.

40935 On systems not supporting these functions, their capability can sometimes be accomplished by
 40936 saving a filename found by *readdir()* and later using *rewinddir()* and a loop on *readdir()* to
 40937 relocate the position from which the filename was saved.

40938 **FUTURE DIRECTIONS**

40939 None.

40940 **SEE ALSO**

40941 *fdopendir()*, *readdir()*, *telldir()*, the Base Definitions volume of IEEE Std 1003.1-200x, *<dirent.h>*,
 40942 *<stdio.h>*, *<sys/types.h>*

40943 **CHANGE HISTORY**

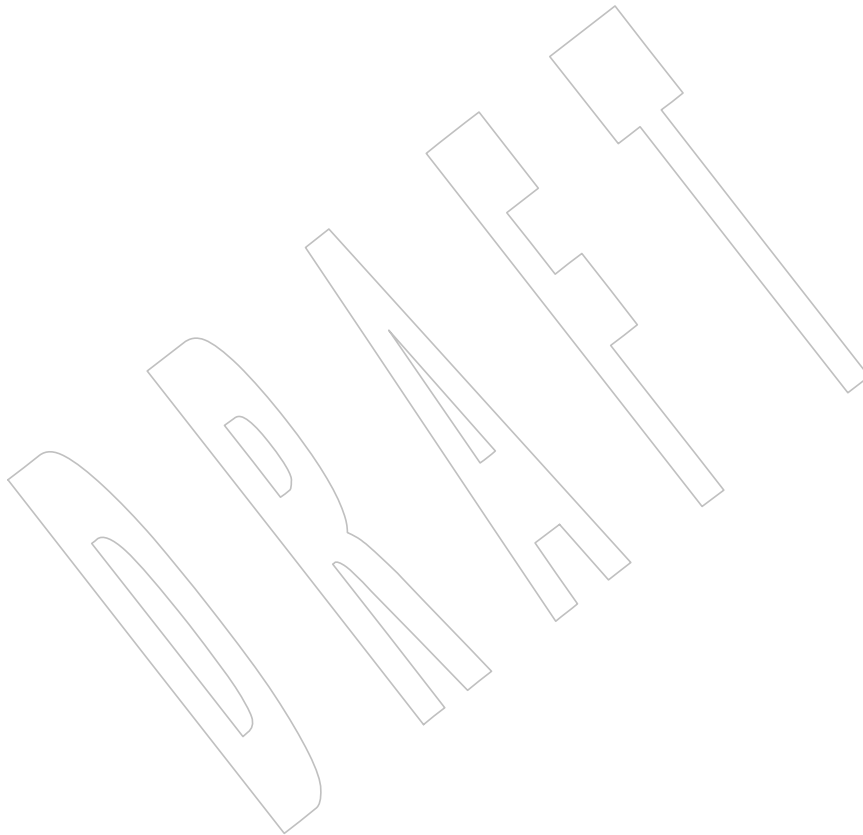
40944 First released in Issue 2.

40945

Issue 6

40946

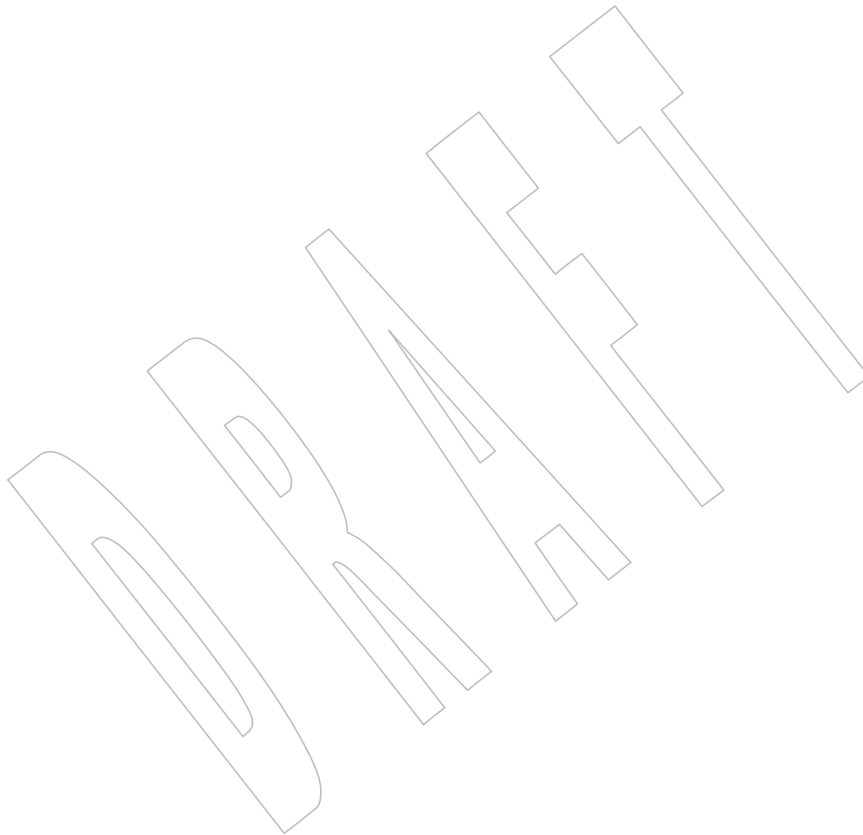
In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.



40947 **NAME**
40948 select — synchronous I/O multiplexing

40949 **SYNOPSIS**
40950 #include <sys/select.h>
40951 int select(int *nfds*, fd_set *restrict *readfds*,
40952 fd_set *restrict *writefds*, fd_set *restrict *errorfds*,
40953 struct timeval *restrict *timeout*);

40954 **DESCRIPTION**
40955 Refer to *pselect()*.



40956 **NAME**40957 `sem_close` — close a named semaphore40958 **SYNOPSIS**40959 `#include <semaphore.h>`40960 `int sem_close(sem_t *sem);`40961 **DESCRIPTION**

40962 The `sem_close()` function shall indicate that the calling process is finished using the named
 40963 semaphore indicated by `sem`. The effects of calling `sem_close()` for an unnamed semaphore (one
 40964 created by `sem_init()`) are undefined. The `sem_close()` function shall deallocate (that is, make
 40965 available for reuse by a subsequent `sem_open()` by this process) any system resources allocated
 40966 by the system for use by this process for this semaphore. The effect of subsequent use of the
 40967 semaphore indicated by `sem` by this process is undefined. If the semaphore has not been
 40968 removed with a successful call to `sem_unlink()`, then `sem_close()` has no effect on the state of the
 40969 semaphore. If the `sem_unlink()` function has been successfully invoked for `name` after the most
 40970 recent call to `sem_open()` with `O_CREAT` for this semaphore, then when all processes that have
 40971 opened the semaphore close it, the semaphore is no longer accessible.

40972 **RETURN VALUE**

40973 Upon successful completion, a value of zero shall be returned. Otherwise, a value of `-1` shall be
 40974 returned and `errno` set to indicate the error.

40975 **ERRORS**40976 The `sem_close()` function may fail if:40977 [EINVAL] The `sem` argument is not a valid semaphore descriptor.40978 **EXAMPLES**

40979 None.

40980 **APPLICATION USAGE**

40981 The `sem_close()` function is part of the Semaphores option and need not be available on all
 40982 implementations.

40983 **RATIONALE**

40984 None.

40985 **FUTURE DIRECTIONS**

40986 None.

40987 **SEE ALSO**

40988 `semctl()`, `semget()`, `semop()`, `sem_init()`, `sem_open()`, `sem_unlink()`, the Base Definitions volume of
 40989 IEEE Std 1003.1-200x, `<semaphore.h>`

40990 **CHANGE HISTORY**

40991 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

40992 **Issue 6**40993 The `sem_close()` function is marked as part of the Semaphores option.

40994 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 40995 implementation does not support the Semaphores option.

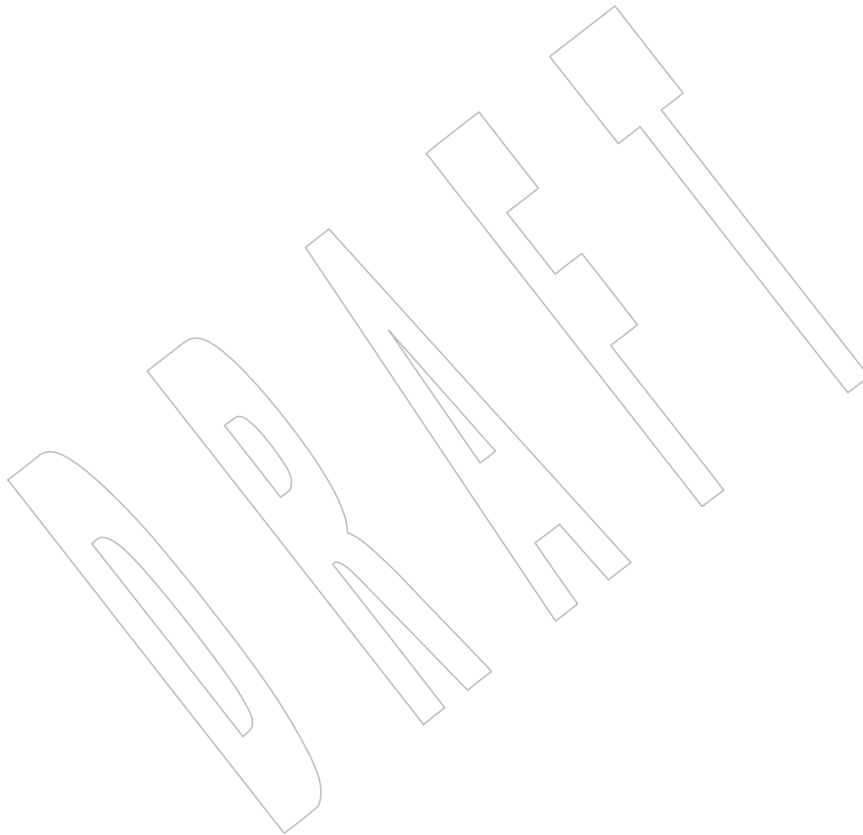
40996 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/113 is applied, updating the ERRORS
 40997 section so that the [EINVAL] error becomes optional.

40998

Issue 7

40999

The *sem_close()* function is moved from the Semaphores option to the Base.



41000 **NAME**
 41001 `sem_destroy` — destroy an unnamed semaphore

41002 **SYNOPSIS**
 41003 `#include <semaphore.h>`
 41004 `int sem_destroy(sem_t *sem);`

41005 **DESCRIPTION**
 41006 The `sem_destroy()` function shall destroy the unnamed semaphore indicated by `sem`. Only a
 41007 semaphore that was created using `sem_init()` may be destroyed using `sem_destroy()`; the effect of
 41008 calling `sem_destroy()` with a named semaphore is undefined. The effect of subsequent use of the
 41009 semaphore `sem` is undefined until `sem` is reinitialized by another call to `sem_init()`.

41010 It is safe to destroy an initialized semaphore upon which no threads are currently blocked. The
 41011 effect of destroying a semaphore upon which other threads are currently blocked is undefined.

41012 **RETURN VALUE**
 41013 Upon successful completion, a value of zero shall be returned. Otherwise, a value of `-1` shall be
 41014 returned and `errno` set to indicate the error.

41015 **ERRORS**
 41016 The `sem_destroy()` function may fail if:
 41017 [EINVAL] The `sem` argument is not a valid semaphore.
 41018 [EBUSY] There are currently processes blocked on the semaphore.

41019 **EXAMPLES**
 41020 None.

41021 **APPLICATION USAGE**
 41022 The `sem_destroy()` function is part of the Semaphores option and need not be available on all
 41023 implementations.

41024 **RATIONALE**
 41025 None.

41026 **FUTURE DIRECTIONS**
 41027 None.

41028 **SEE ALSO**
 41029 [semctl\(\)](#), [semget\(\)](#), [semop\(\)](#), [sem_init\(\)](#), [sem_open\(\)](#), the Base Definitions volume of
 41030 IEEE Std 1003.1-200x, [<semaphore.h>](#)

41031 **CHANGE HISTORY**
 41032 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

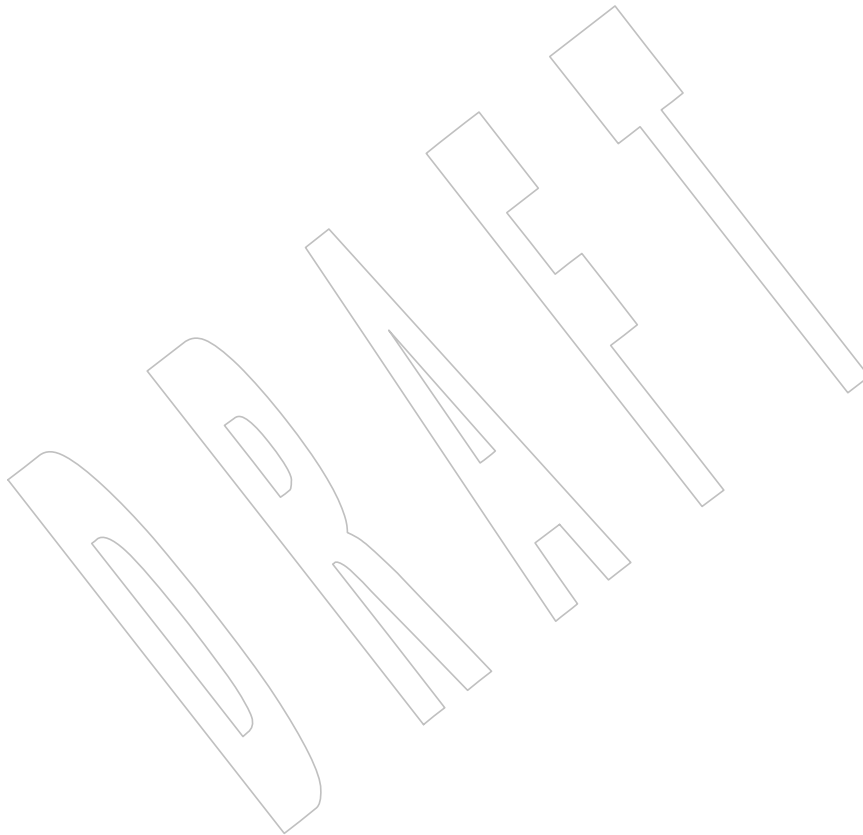
41033 **Issue 6**
 41034 The `sem_destroy()` function is marked as part of the Semaphores option.
 41035 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 41036 implementation does not support the Semaphores option.
 41037 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/114 is applied, updating the ERRORS
 41038 section so that the [EINVAL] error becomes optional.

41039

Issue 7

41040

The *dem_destroy()* function is moved from the Semaphores option to the Base.



41041 **NAME**41042 `sem_getvalue` — get the value of a semaphore41043 **SYNOPSIS**41044 `#include <semaphore.h>`41045 `int sem_getvalue(sem_t *restrict sem, int *restrict sval);`41046 **DESCRIPTION**

41047 The `sem_getvalue()` function shall update the location referenced by the `sval` argument to have
 41048 the value of the semaphore referenced by `sem` without affecting the state of the semaphore. The
 41049 updated value represents an actual semaphore value that occurred at some unspecified time
 41050 during the call, but it need not be the actual value of the semaphore when it is returned to the
 41051 calling process.

41052 If `sem` is locked, then the object to which `sval` points shall either be set to zero or to a negative
 41053 number whose absolute value represents the number of processes waiting for the semaphore at
 41054 some unspecified time during the call.

41055 **RETURN VALUE**

41056 Upon successful completion, the `sem_getvalue()` function shall return a value of zero. Otherwise,
 41057 it shall return a value of `-1` and set `errno` to indicate the error.

41058 **ERRORS**

41059 The `sem_getvalue()` function may fail if:

41060 [EINVAL] The `sem` argument does not refer to a valid semaphore.

41061 **EXAMPLES**

41062 None.

41063 **APPLICATION USAGE**

41064 The `sem_getvalue()` function is part of the Semaphores option and need not be available on all
 41065 implementations.

41066 **RATIONALE**

41067 None.

41068 **FUTURE DIRECTIONS**

41069 None.

41070 **SEE ALSO**

41071 [semctl\(\)](#), [semget\(\)](#), [semop\(\)](#), [sem_post\(\)](#), [sem_timedwait\(\)](#), [sem_trywait\(\)](#), [sem_wait\(\)](#), the Base
 41072 Definitions volume of IEEE Std 1003.1-200x, [<semaphore.h>](#)

41073 **CHANGE HISTORY**

41074 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

41075 **Issue 6**

41076 The `sem_getvalue()` function is marked as part of the Semaphores option.

41077 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 41078 implementation does not support the Semaphores option.

41079 The `sem_timedwait()` function is added to the SEE ALSO section for alignment with IEEE Std
 41080 1003.1d-1999.

41081 The **restrict** keyword is added to the `sem_getvalue()` prototype for alignment with the
 41082 ISO/IEC 9899:1999 standard.

41083 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/54 is applied.

41084

41085

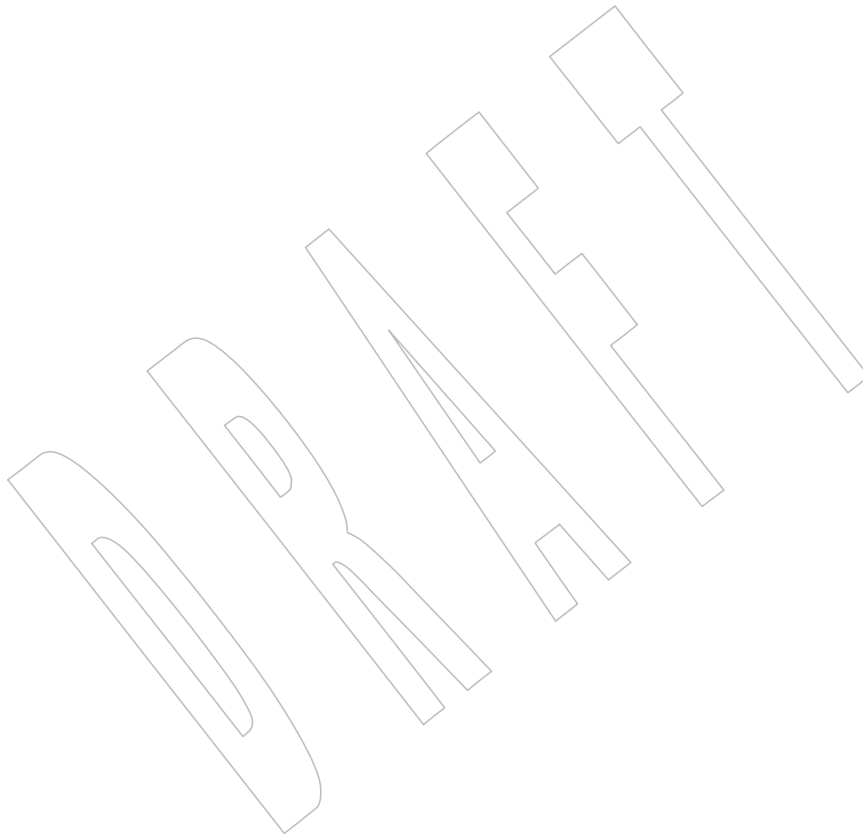
IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/115 is applied, updating the ERRORS section so that the [EINVAL] error becomes optional.

41086

Issue 7

41087

The *sem_getvalue()* function is moved from the Semaphores option to the Base.



41088 NAME

41089 `sem_init` — initialize an unnamed semaphore

41090 SYNOPSIS

41091 `#include <semaphore.h>`

41092 `int sem_init(sem_t *sem, int pshared, unsigned value);`

41093 DESCRIPTION

41094 The `sem_init()` function shall initialize the unnamed semaphore referred to by `sem`. The value of
 41095 the initialized semaphore shall be `value`. Following a successful call to `sem_init()`, the semaphore
 41096 may be used in subsequent calls to `sem_wait()`, `sem_timedwait()`, `sem_trywait()`, `sem_post()`, and
 41097 `sem_destroy()`. This semaphore shall remain usable until the semaphore is destroyed.

41098 If the `pshared` argument has a non-zero value, then the semaphore is shared between processes;
 41099 in this case, any process that can access the semaphore `sem` can use `sem` for performing
 41100 `sem_wait()`, `sem_timedwait()`, `sem_trywait()`, `sem_post()`, and `sem_destroy()` operations.

41101 Only `sem` itself may be used for performing synchronization. The result of referring to copies of
 41102 `sem` in calls to `sem_wait()`, `sem_timedwait()`, `sem_trywait()`, `sem_post()`, and `sem_destroy()` is
 41103 undefined.

41104 If the `pshared` argument is zero, then the semaphore is shared between threads of the process; any
 41105 thread in this process can use `sem` for performing `sem_wait()`, `sem_timedwait()`, `sem_trywait()`,
 41106 `sem_post()`, and `sem_destroy()` operations. The use of the semaphore by threads other than those
 41107 created in the same process is undefined.

41108 Attempting to initialize an already initialized semaphore results in undefined behavior.

41109 RETURN VALUE

41110 Upon successful completion, the `sem_init()` function shall initialize the semaphore in `sem`.
 41111 Otherwise, it shall return `-1` and set `errno` to indicate the error.

41112 ERRORS

41113 The `sem_init()` function shall fail if:

41114 [EINVAL] The `value` argument exceeds `{SEM_VALUE_MAX}`.

41115 [ENOSPC] A resource required to initialize the semaphore has been exhausted, or the
 41116 limit on semaphores (`{SEM_NSEMS_MAX}`) has been reached.

41117 [EPERM] The process lacks the appropriate privileges to initialize the semaphore.

41118 EXAMPLES

41119 None.

41120 APPLICATION USAGE

41121 The `sem_init()` function is part of the Semaphores option and need not be available on all
 41122 implementations.

41123 RATIONALE

41124 Although this volume of IEEE Std 1003.1-200x fails to specify a successful return value, it is
 41125 likely that a later version may require the implementation to return a value of zero if the call to
 41126 `sem_init()` is successful.

41127 FUTURE DIRECTIONS

41128 None.

41129
41130
41131
41132
41133
41134
41135
41136
41137
41138
41139
41140
41141
41142
41143

SEE ALSO

sem_destroy(), *sem_post()*, *sem_timedwait()*, *sem_trywait()*, *sem_wait()*, the Base Definitions volume of IEEE Std 1003.1-200x, <semaphore.h>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sem_init()* function is marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

The *sem_timedwait()* function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/116 is applied, updating the DESCRIPTION to add the *sem_timedwait()* function for alignment with IEEE Std 1003.1d-1999.

Issue 7

The *sem_init()* function is moved from the Semaphores option to the Base.

DRAFT

41144 NAME

41145 sem_open — initialize and open a named semaphore

41146 SYNOPSIS

41147 #include <semaphore.h>

41148 sem_t *sem_open(const char *name, int oflag, ...);

41149 DESCRIPTION

41150 The *sem_open()* function shall establish a connection between a named semaphore and a process.
 41151 Following a call to *sem_open()* with semaphore name *name*, the process may reference the
 41152 semaphore associated with *name* using the address returned from the call. This semaphore may
 41153 be used in subsequent calls to *sem_wait()*, *sem_timedwait()*, *sem_trywait()*, *sem_post()*, and
 41154 *sem_close()*. The semaphore remains usable by this process until the semaphore is closed by a
 41155 successful call to *sem_close()*, *_exit()*, or one of the *exec* functions.

41156 The *oflag* argument controls whether the semaphore is created or merely accessed by the call to
 41157 *sem_open()*. The following flag bits may be set in *oflag*:

41158 **O_CREAT** This flag is used to create a semaphore if it does not already exist. If **O_CREAT** is
 41159 set and the semaphore already exists, then **O_CREAT** has no effect, except as noted
 41160 under **O_EXCL**. Otherwise, *sem_open()* creates a named semaphore. The **O_CREAT**
 41161 flag requires a third and a fourth argument: *mode*, which is of type **mode_t**, and
 41162 *value*, which is of type **unsigned**. The semaphore is created with an initial value of
 41163 *value*. Valid initial values for semaphores are less than or equal to
 41164 {SEM_VALUE_MAX}.

41165 The user ID of the semaphore is set to the effective user ID of the process; the
 41166 group ID of the semaphore is set to a system default group ID or to the effective
 41167 group ID of the process. The permission bits of the semaphore are set to the value
 41168 of the *mode* argument except those set in the file mode creation mask of the process.
 41169 When bits in *mode* other than the file permission bits are specified, the effect is
 41170 unspecified.

41171 After the semaphore named *name* has been created by *sem_open()* with the
 41172 **O_CREAT** flag, other processes can connect to the semaphore by calling
 41173 *sem_open()* with the same value of *name*.

41174 **O_EXCL** If **O_EXCL** and **O_CREAT** are set, *sem_open()* fails if the semaphore *name* exists.
 41175 The check for the existence of the semaphore and the creation of the semaphore if it
 41176 does not exist are atomic with respect to other processes executing *sem_open()* with
 41177 **O_EXCL** and **O_CREAT** set. If **O_EXCL** is set and **O_CREAT** is not set, the effect is
 41178 undefined.

41179 If flags other than **O_CREAT** and **O_EXCL** are specified in the *oflag* parameter, the
 41180 effect is unspecified.

41181 The *name* argument points to a string naming a semaphore object. It is unspecified whether the
 41182 name appears in the file system and is visible to functions that take pathnames as arguments.
 41183 The *name* argument conforms to the construction rules for a pathname, except that the
 41184 interpretation of slash characters other than the leading slash character in *name* is
 41185 implementation-defined, and that the length limits for the *name* argument are implementation-
 41186 defined and need not be the same as the path name limits {PATH_MAX} and {NAME_MAX}. If
 41187 *name* begins with the slash character, then processes calling *sem_open()* with the same value of
 41188 *name* shall refer to the same semaphore object, as long as that name has not been removed. If
 41189 *name* does not begin with the slash character, the effect is implementation-defined.

41190 If a process makes multiple successful calls to *sem_open()* with the same value for *name*, the same
 41191 semaphore address shall be returned for each such successful call, provided that there have been
 41192 no calls to *sem_unlink()* for this semaphore, and at least one previous successful *sem_open()* call
 41193 for this semaphore has not been matched with a *sem_close()* call.

41194 References to copies of the semaphore produce undefined results.

41195 RETURN VALUE

41196 Upon successful completion, the *sem_open()* function shall return the address of the semaphore.
 41197 Otherwise, it shall return a value of SEM_FAILED and set *errno* to indicate the error. The symbol
 41198 SEM_FAILED is defined in the `<semaphore.h>` header. No successful return from *sem_open()*
 41199 shall return the value SEM_FAILED.

41200 ERRORS

41201 If any of the following conditions occur, the *sem_open()* function shall return SEM_FAILED and
 41202 set *errno* to the corresponding value:

41203 [EACCES] The named semaphore exists and the permissions specified by *oflag* are
 41204 denied, or the named semaphore does not exist and permission to create the
 41205 named semaphore is denied.

41206 [EEXIST] O_CREAT and O_EXCL are set and the named semaphore already exists.

41207 [EINTR] The *sem_open()* operation was interrupted by a signal.

41208 [EINVAL] The *sem_open()* operation is not supported for the given name, or O_CREAT
 41209 was specified in *oflag* and *value* was greater than {SEM_VALUE_MAX}.

41210 [EMFILE] Too many semaphore descriptors or file descriptors are currently in use by this
 41211 process.

41212 [ENFILE] Too many semaphores are currently open in the system.

41213 [ENOENT] O_CREAT is not set and the named semaphore does not exist.

41214 [ENOMEM] There is insufficient memory for the creation of the new named semaphore.

41215 [ENOSPC] There is insufficient space on a storage device for the creation of the new
 41216 named semaphore.

41217 If any of the following conditions occur, the *sem_open()* function may return SEM_FAILED and
 41218 set *errno* to the corresponding value:

41219 [ENAMETOOLONG]

41220 The length of the *name* argument exceeds {_POSIX_PATH_MAX} on systems
 41221 XSI that do not support the XSI option or exceeds {_XOPEN_PATH_MAX} on XSI
 41222 systems, or has a pathname component that is longer than
 41223 XSI {_POSIX_NAME_MAX} on systems that do not support the XSI option or
 41224 longer than {_XOPEN_NAME_MAX} on XSI systems.

41225 EXAMPLES

41226 None.

41227 APPLICATION USAGE

41228 The *sem_open()* function is part of the Semaphores option and need not be available on all
 41229 implementations.

41230 RATIONALE

41231 Early drafts required an error return value of -1 with the type `sem_t *` for the *sem_open()*
 41232 function, which is not guaranteed to be portable across implementations. The revised text
 41233 provides the symbolic error code SEM_FAILED to eliminate the type conflict.

41234
41235
41236
41237
41238
41239
41240
41241
41242
41243
41244
41245
41246
41247
41248
41249
41250
41251
41252
41253
41254
41255
41256
41257
41258

FUTURE DIRECTIONS

None.

SEE ALSO

semctl(), *semget()*, *semop()*, *sem_close()*, *sem_post()*, *sem_timedwait()*, *sem_trywait()*, *sem_unlink()*, *sem_wait()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**semaphore.h**>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sem_open()* function is marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

The *sem_timedwait()* function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/117 is applied, updating the DESCRIPTION to add the *sem_timedwait()* function for alignment with IEEE Std 1003.1d-1999.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/118 is applied, updating the DESCRIPTION to describe the conditions to return the same semaphore address on a call to *sem_open()*. The words “and at least one previous successful *sem_open()* call for this semaphore has not been matched with a *sem_close()* call” are added.

Issue 7

Austin Group Interpretation 1003.1-2001 #066 is applied, updating the [ENOSPC] error case and adding the [ENOMEM] error case.

Austin Group Interpretation 1003.1-2001 #077 is applied, clarifying the *name* argument and adding [ENAMETOOLONG] as a “may fail” error.

The *sem_open()* function is moved from the Semaphores option to the Base.

41259 **NAME**

41260 sem_post — unlock a semaphore

41261 **SYNOPSIS**

41262 #include <semaphore.h>

41263 int sem_post(sem_t *sem);

41264 **DESCRIPTION**41265 The *sem_post()* function shall unlock the semaphore referenced by *sem* by performing a
41266 semaphore unlock operation on that semaphore.41267 If the semaphore value resulting from this operation is positive, then no threads were blocked
41268 waiting for the semaphore to become unlocked; the semaphore value is simply incremented.41269 If the value of the semaphore resulting from this operation is zero, then one of the threads
41270 blocked waiting for the semaphore shall be allowed to return successfully from its call to
41271 PS *sem_wait()*. If the Process Scheduling option is supported, the thread to be unblocked shall be
41272 chosen in a manner appropriate to the scheduling policies and parameters in effect for the
41273 blocked threads. In the case of the schedulers SCHED_FIFO and SCHED_RR, the highest
41274 priority waiting thread shall be unblocked, and if there is more than one highest priority thread
41275 blocked waiting for the semaphore, then the highest priority thread that has been waiting the
41276 longest shall be unblocked. If the Process Scheduling option is not defined, the choice of a thread
41277 to unblock is unspecified.41278 SS If the Process Sporadic Server option is supported, and the scheduling policy is
41279 SCHED_SPORADIC, the semantics are as per SCHED_FIFO above.41280 The *sem_post()* function shall be reentrant with respect to signals and may be invoked from a
41281 signal-catching function.41282 **RETURN VALUE**41283 If successful, the *sem_post()* function shall return zero; otherwise, the function shall return -1
41284 and set *errno* to indicate the error.41285 **ERRORS**41286 The *sem_post()* function may fail if:41287 [EINVAL] The *sem* argument does not refer to a valid semaphore.41288 **EXAMPLES**

41289 None.

41290 **APPLICATION USAGE**41291 The *sem_post()* function is part of the Semaphores option and need not be available on all
41292 implementations.41293 **RATIONALE**

41294 None.

41295 **FUTURE DIRECTIONS**

41296 None.

41297 **SEE ALSO**41298 *semctl()*, *semget()*, *semop()*, *sem_timedwait()*, *sem_trywait()*, *sem_wait()*, the Base Definitions
41299 volume of IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, <semaphore.h>

41300

CHANGE HISTORY

41301

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

41302

Issue 6

41303

The *sem_post()* function is marked as part of the Semaphores option.

41304

41305

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

41306

41307

The *sem_timedwait()* function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

41308

41309

SCHED_SPORADIC is added to the list of scheduling policies for which the thread that is to be unblocked is specified for alignment with IEEE Std 1003.1d-1999.

41310

41311

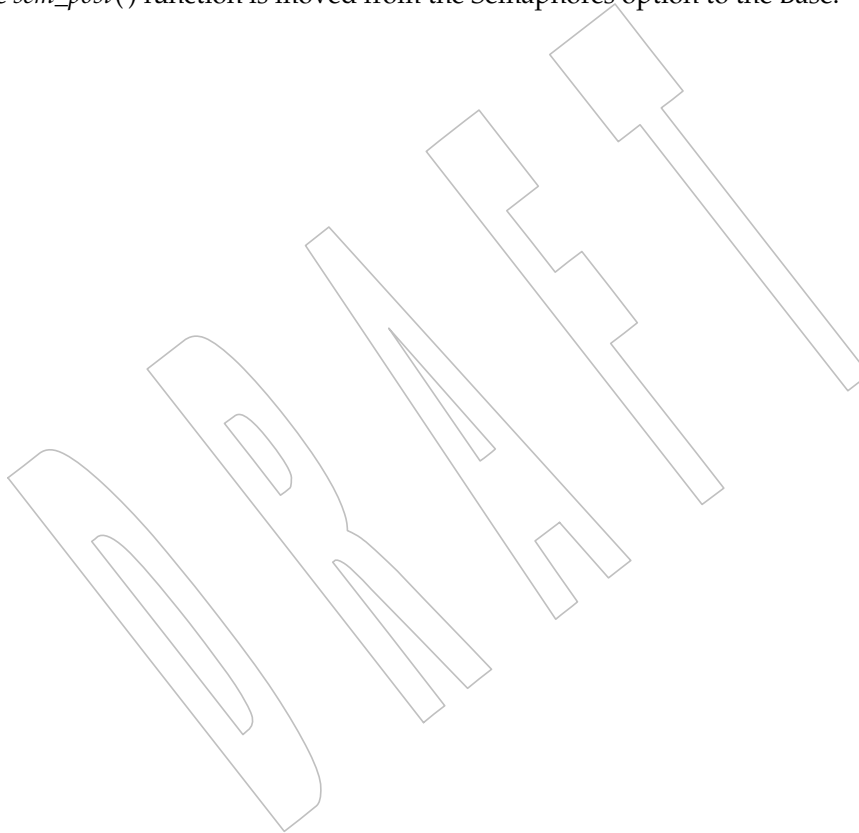
IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/119 is applied, updating the ERRORS section so that the [EINVAL] error becomes optional.

41312

Issue 7

41313

The *sem_post()* function is moved from the Semaphores option to the Base.



41314 **NAME**
 41315 `sem_timedwait` — lock a semaphore

41316 **SYNOPSIS**
 41317 `#include <semaphore.h>`
 41318 `#include <time.h>`
 41319 `int sem_timedwait(sem_t *restrict sem,`
 41320 `const struct timespec *restrict abs_timeout);`

41321 **DESCRIPTION**
 41322 The `sem_timedwait()` function shall lock the semaphore referenced by `sem` as in the `sem_wait()`
 41323 function. However, if the semaphore cannot be locked without waiting for another process or
 41324 thread to unlock the semaphore by performing a `sem_post()` function, this wait shall be
 41325 terminated when the specified timeout expires.

41326 The timeout shall expire when the absolute time specified by `abs_timeout` passes, as measured by
 41327 the clock on which timeouts are based (that is, when the value of that clock equals or exceeds
 41328 `abs_timeout`), or if the absolute time specified by `abs_timeout` has already been passed at the time
 41329 of the call.

41330 The timeout shall be based on the `CLOCK_REALTIME` clock. The resolution of the timeout shall
 41331 be the resolution of the clock on which it is based. The `timespec` data type is defined as a
 41332 structure in the `<time.h>` header.

41333 Under no circumstance shall the function fail with a timeout if the semaphore can be locked
 41334 immediately. The validity of the `abs_timeout` need not be checked if the semaphore can be locked
 41335 immediately.

41336 **RETURN VALUE**
 41337 The `sem_timedwait()` function shall return zero if the calling process successfully performed the
 41338 semaphore lock operation on the semaphore designated by `sem`. If the call was unsuccessful, the
 41339 state of the semaphore shall be unchanged, and the function shall return a value of `-1` and set
 41340 `errno` to indicate the error.

41341 **ERRORS**
 41342 The `sem_timedwait()` function shall fail if:
 41343 [EINVAL] The process or thread would have blocked, and the `abs_timeout` parameter
 41344 specified a nanoseconds field value less than zero or greater than or equal to
 41345 1 000 million.

41346 [ETIMEDOUT] The semaphore could not be locked before the specified timeout expired.

41347 The `sem_timedwait()` function may fail if:

41348 [EDEADLK] A deadlock condition was detected.

41349 [EINTR] A signal interrupted this function.

41350 [EINVAL] The `sem` argument does not refer to a valid semaphore.

sem_timedwait()41351
41352
41353
41354
41355
41356
41357
41358
41359
41360
41361
41362
41363
41364
41365
41366
41367
41368
41369**EXAMPLES**

None.

APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sem_post(), *sem_trywait()*, *sem_wait()*, *semctl()*, *semget()*, *semop()*, *time()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/120 is applied, updating the ERRORS section so that the [EINVAL] error becomes optional.

Issue 7

The *sem_timedwait()* function is moved from the Semaphores option to the Base.

Functionality relating to the Timers option is moved to the Base.

41370 **NAME**41371 `sem_trywait, sem_wait` — lock a semaphore41372 **SYNOPSIS**41373 `#include <semaphore.h>`41374 `int sem_trywait(sem_t *sem);`41375 `int sem_wait(sem_t *sem);`41376 **DESCRIPTION**41377 The `sem_trywait()` function shall lock the semaphore referenced by `sem` only if the semaphore is
41378 currently not locked; that is, if the semaphore value is currently positive. Otherwise, it shall not
41379 lock the semaphore.41380 The `sem_wait()` function shall lock the semaphore referenced by `sem` by performing a semaphore
41381 lock operation on that semaphore. If the semaphore value is currently zero, then the calling
41382 thread shall not return from the call to `sem_wait()` until it either locks the semaphore or the call is
41383 interrupted by a signal.41384 Upon successful return, the state of the semaphore shall be locked and shall remain locked until
41385 the `sem_post()` function is executed and returns successfully.41386 The `sem_wait()` function is interruptible by the delivery of a signal.41387 **RETURN VALUE**41388 The `sem_trywait()` and `sem_wait()` functions shall return zero if the calling process successfully
41389 performed the semaphore lock operation on the semaphore designated by `sem`. If the call was
41390 unsuccessful, the state of the semaphore shall be unchanged, and the function shall return a
41391 value of `-1` and set `errno` to indicate the error.41392 **ERRORS**41393 The `sem_trywait()` function shall fail if:41394 [EAGAIN] The semaphore was already locked, so it cannot be immediately locked by the
41395 `sem_trywait()` operation.41396 The `sem_trywait()` and `sem_wait()` functions may fail if:

41397 [EDEADLK] A deadlock condition was detected.

41398 [EINTR] A signal interrupted this function.

41399 [EINVAL] The `sem` argument does not refer to a valid semaphore.41400 **EXAMPLES**

41401 None.

41402 **APPLICATION USAGE**41403 Applications using these functions may be subject to priority inversion, as discussed in the Base
41404 Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.41405 The `sem_trywait()` and `sem_wait()` functions are part of the Semaphores option and need not be
41406 provided on all implementations.41407 **RATIONALE**

41408 None.

41409 **FUTURE DIRECTIONS**

41410 None.

41411 **SEE ALSO**41412 *semctl()*, *semget()*, *semop()*, *sem_post()*, *sem_timedwait()*, the Base Definitions volume of
41413 IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, <**semaphore.h**>41414 **CHANGE HISTORY**

41415 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

41416 **Issue 6**41417 The *sem_trywait()* and *sem_wait()* functions are marked as part of the Semaphores option.41418 The [ENOSYS] error condition has been removed as stubs need not be provided if an
41419 implementation does not support the Semaphores option.41420 The *sem_timedwait()* function is added to the SEE ALSO section for alignment with IEEE Std
41421 1003.1d-1999.41422 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/121 is applied, updating the ERRORS
41423 section so that the [EINVAL] error becomes optional.41424 **Issue 7**41425 SD5-XSH-ERN-54 is applied, removing the *sem_wait()* function from the “shall fail” error cases.41426 The *sem_trywait()* and *sem_wait()* functions are moved from the Semaphores option to the Base.

41427 **NAME**

41428 sem_unlink — remove a named semaphore

41429 **SYNOPSIS**

41430 #include <semaphore.h>

41431 int sem_unlink(const char *name);

41432 **DESCRIPTION**

41433 The *sem_unlink()* function shall remove the semaphore named by the string *name*. If the
 41434 semaphore named by *name* is currently referenced by other processes, then *sem_unlink()* shall
 41435 have no effect on the state of the semaphore. If one or more processes have the semaphore open
 41436 when *sem_unlink()* is called, destruction of the semaphore is postponed until all references to the
 41437 semaphore have been destroyed by calls to *sem_close()*, *_exit()*, or *exec*. Calls to *sem_open()*
 41438 to recreate or reconnect to the semaphore refer to a new semaphore after *sem_unlink()* is called. The
 41439 *sem_unlink()* call shall not block until all references have been destroyed; it shall return
 41440 immediately.

41441 **RETURN VALUE**

41442 Upon successful completion, the *sem_unlink()* function shall return a value of 0. Otherwise, the
 41443 semaphore shall not be changed and the function shall return a value of -1 and set *errno* to
 41444 indicate the error.

41445 **ERRORS**41446 The *sem_unlink()* function shall fail if:

41447 [EACCES] Permission is denied to unlink the named semaphore.

41448 [ENOENT] The named semaphore does not exist.

41449 The *sem_unlink()* function may fail if:

41450 [ENAMETOOLONG]

41451 The length of the *name* argument exceeds `{_POSIX_PATH_MAX}` on systems
 41452 XSI that do not support the XSI option or exceeds `{_XOPEN_PATH_MAX}` on XSI
 41453 systems, or has a pathname component that is longer than
 41454 XSI `{_POSIX_NAME_MAX}` on systems that do not support the XSI option or
 41455 longer than `{_XOPEN_NAME_MAX}` on XSI systems. A call to *sem_unlink()*
 41456 with a *name* argument that contains the same semaphore name as was
 41457 previously used in a successful *sem_open()* call shall not give an
 41458 [ENAMETOOLONG] error.

41459 **EXAMPLES**

41460 None.

41461 **APPLICATION USAGE**

41462 The *sem_unlink()* function is part of the Semaphores option and need not be available on all
 41463 implementations.

41464 **RATIONALE**

41465 None.

41466 **FUTURE DIRECTIONS**

41467 None.

sem_unlink()

41468

SEE ALSO

41469

semctl(), *semget()*, *semop()*, *sem_close()*, *sem_open()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**

41470

41471

CHANGE HISTORY

41472

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

41473

Issue 6

41474

The *sem_unlink()* function is marked as part of the Semaphores option.

41475

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

41476

41477

Issue 7

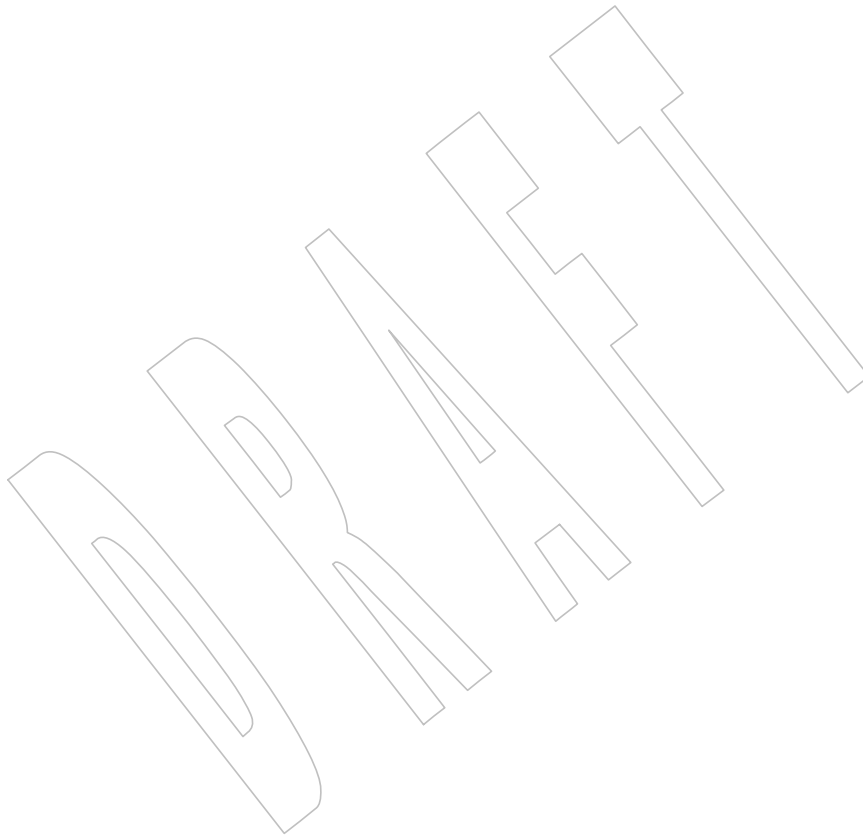
41478

Austin Group Interpretation 1003.1-2001 #077 is applied, changing [ENAMETOOLONG] from a “shall fail” to a “may fail” error.

41479

41480

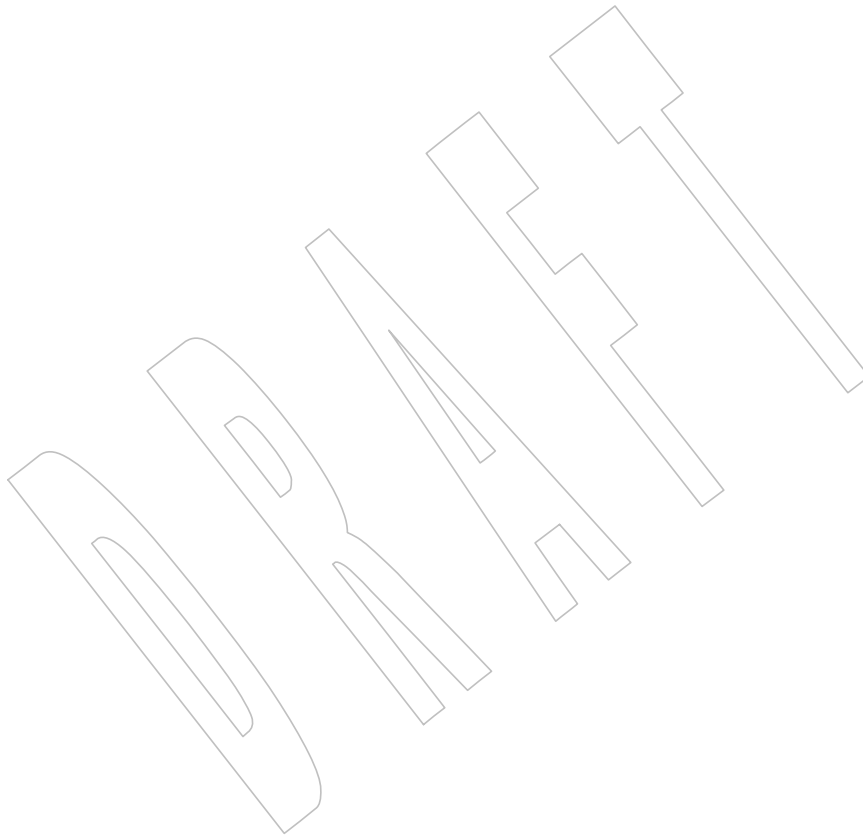
The *sem_unlink()* function is moved from the Semaphores option to the Base.



41481 **NAME**
41482 `sem_wait` — lock a semaphore

41483 **SYNOPSIS**
41484 `#include <semaphore.h>`
41485 `int sem_wait(sem_t *sem);`

41486 **DESCRIPTION**
41487 Refer to *sem_trywait()*.



41488 NAME

41489 semctl — XSI semaphore control operations

41490 SYNOPSIS

```
41491 XSI #include <sys/sem.h>
41492 int semctl(int semid, int semnum, int cmd, ...);
```

41493 DESCRIPTION

41494 The *semctl()* function operates on XSI semaphores (see the Base Definitions volume of
41495 IEEE Std 1003.1-200x, Section 4.15, Semaphore). It is unspecified whether this function
41496 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
41497 page 40).

41498 The *semctl()* function provides a variety of semaphore control operations as specified by *cmd*.
41499 The fourth argument is optional and depends upon the operation requested. If required, it is of
41500 type **union semun**, which the application shall explicitly declare:

```
41501 union semun {
41502     int val;
41503     struct semid_ds *buf;
41504     unsigned short *array;
41505 } arg;
```

41506 The following semaphore control operations as specified by *cmd* are executed with respect to the
41507 semaphore specified by *semid* and *semnum*. The level of permission required for each operation
41508 is shown with each command; see Section 2.7 (on page 39). The symbolic names for the values
41509 of *cmd* are defined in the `<sys/sem.h>` header:

41510	GETVAL	Return the value of <i>semval</i> ; see <code><sys/sem.h></code> . Requires read permission.
41511	SETVAL	Set the value of <i>semval</i> to <i>arg.val</i> , where <i>arg</i> is the value of the fourth argument to <i>semctl()</i> . When this command is successfully executed, the <i>semadj</i> value corresponding to the specified semaphore in all processes is cleared. Requires alter permission; see Section 2.7 (on page 39).
41512		
41513		
41514		
41515	GETPID	Return the value of <i>sempid</i> . Requires read permission.
41516	GETNCNT	Return the value of <i>semmcnt</i> . Requires read permission.
41517	GETZCNT	Return the value of <i>semzcnt</i> . Requires read permission.

41518 The following values of *cmd* operate on each *semval* in the set of semaphores:

41519	GETALL	Return the value of <i>semval</i> for each semaphore in the semaphore set and place into the array pointed to by <i>arg.array</i> , where <i>arg</i> is the fourth argument to <i>semctl()</i> . Requires read permission.
41520		
41521		
41522	SETALL	Set the value of <i>semval</i> for each semaphore in the semaphore set according to the array pointed to by <i>arg.array</i> , where <i>arg</i> is the fourth argument to <i>semctl()</i> . When this command is successfully executed, the <i>semadj</i> values corresponding to each specified semaphore in all processes are cleared. Requires alter permission.
41523		
41524		
41525		
41526		

41527 The following values of *cmd* are also available:

41528	IPC_STAT	Place the current value of each member of the semid_ds data structure associated with <i>semid</i> into the structure pointed to by <i>arg.buf</i> , where <i>arg</i> is the fourth argument to <i>semctl()</i> . The contents of this structure are defined in
41529		
41530		

41531		< sys/sem.h >. Requires read permission.
41532	IPC_SET	Set the value of the following members of the semid_ds data structure associated with <i>semid</i> to the corresponding value found in the structure pointed to by <i>arg.buf</i> , where <i>arg</i> is the fourth argument to <i>semctl()</i> :
41533		
41534		
41535		<i>sem_perm.uid</i>
41536		<i>sem_perm.gid</i>
41537		<i>sem_perm.mode</i>
41538		The mode bits specified in Section 2.7.1 are copied into the corresponding bits of the <i>sem_perm.mode</i> associated with <i>semid</i> . The stored values of any other bits are unspecified.
41539		
41540		
41541		This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of <i>sem_perm.cuid</i> or <i>sem_perm.uid</i> in the semid_ds data structure associated with <i>semid</i> .
41542		
41543		
41544		
41545	IPC_RMID	Remove the semaphore identifier specified by <i>semid</i> from the system and destroy the set of semaphores and semid_ds data structure associated with it. This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of <i>sem_perm.cuid</i> or <i>sem_perm.uid</i> in the semid_ds data structure associated with <i>semid</i> .
41546		
41547		
41548		
41549		
41550		
41551	RETURN VALUE	
41552		If successful, the value returned by <i>semctl()</i> depends on <i>cmd</i> as follows:
41553	GETVAL	The value of <i>semval</i> .
41554	GETPID	The value of <i>sempid</i> .
41555	GETNCNT	The value of <i>semmcnt</i> .
41556	GETZCNT	The value of <i>semzcnt</i> .
41557	All others	0.
41558		Otherwise, <i>semctl()</i> shall return -1 and set <i>errno</i> to indicate the error.
41559	ERRORS	
41560		The <i>semctl()</i> function shall fail if:
41561	[EACCES]	Operation permission is denied to the calling process; see Section 2.7 (on page 39).
41562		
41563	[EINVAL]	The value of <i>semid</i> is not a valid semaphore identifier, or the value of <i>semnum</i> is less than 0 or greater than or equal to <i>sem_nsems</i> , or the value of <i>cmd</i> is not a valid command.
41564		
41565		
41566	[EPERM]	The argument <i>cmd</i> is equal to IPC_RMID or IPC_SET and the effective user ID of the calling process is not equal to that of a process with appropriate privileges and it is not equal to the value of <i>sem_perm.cuid</i> or <i>sem_perm.uid</i> in the data structure associated with <i>semid</i> .
41567		
41568		
41569		
41570	[ERANGE]	The argument <i>cmd</i> is equal to SETVAL or SETALL and the value to which <i>semval</i> is to be set is greater than the system-imposed maximum.
41571		

41572

EXAMPLES

41573

None.

41574

APPLICATION USAGE

41575

The fourth parameter in the SYNOPSIS section is now specified as ". . ." in order to avoid a clash with the ISO C standard when referring to the union *semun* (as defined in Issue 3) and for backwards-compatibility.

41576

41577

41578

The POSIX Realtime Extension defines alternative interfaces for interprocess communication. Application developers who need to use IPC should design their applications so that modules using the IPC routines described in [Section 2.7](#) can be easily modified to use the alternative interfaces.

41579

41580

41581

41582

RATIONALE

41583

None.

41584

FUTURE DIRECTIONS

41585

None.

41586

SEE ALSO

41587

[Section 2.7](#) (on page 39), [Section 2.8](#) (on page 40), [semget\(\)](#), [semop\(\)](#), [sem_close\(\)](#), [sem_destroy\(\)](#), [sem_getvalue\(\)](#), [sem_init\(\)](#), [sem_open\(\)](#), [sem_post\(\)](#), [sem_unlink\(\)](#), [sem_wait\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, [<sys/sem.h>](#)

41588

41589

41590

CHANGE HISTORY

41591

First released in Issue 2. Derived from Issue 2 of the SVID.

41592

Issue 5

41593

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to the APPLICATION USAGE section.

41594

41595 **NAME**

41596 semget — get set of XSI semaphores

41597 **SYNOPSIS**

```
41598 XSI #include <sys/sem.h>
41599 int semget(key_t key, int nsems, int semflg);
```

41600 **DESCRIPTION**

41601 The *semget()* function operates on XSI semaphores (see the Base Definitions volume of
 41602 IEEE Std 1003.1-200x, Section 4.15, Semaphore). It is unspecified whether this function
 41603 interoperates with the realtime interprocess communication facilities defined in [Section 2.8](#) (on
 41604 page 40).

41605 The *semget()* function shall return the semaphore identifier associated with *key*.

41606 A semaphore identifier with its associated **semid_ds** data structure and its associated set of
 41607 *nsems* semaphores (see **<sys/sem.h>**) is created for *key* if one of the following is true:

- 41608 • The argument *key* is equal to `IPC_PRIVATE`.
- 41609 • The argument *key* does not already have a semaphore identifier associated with it and
 41610 (*semflg* & `IPC_CREAT`) is non-zero.

41611 Upon creation, the **semid_ds** data structure associated with the new semaphore identifier is
 41612 initialized as follows:

- 41613 • In the operation permissions structure *sem_perm.cuid*, *sem_perm.uid*, *sem_perm.cgid*, and
 41614 *sem_perm.gid* shall be set equal to the effective user ID and effective group ID, respectively,
 41615 of the calling process.
- 41616 • The low-order 9 bits of *sem_perm.mode* shall be set equal to the low-order 9 bits of *semflg*.
- 41617 • The variable *sem_nsems* shall be set equal to the value of *nsems*.
- 41618 • The variable *sem_otime* shall be set equal to 0 and *sem_ctime* shall be set equal to the current
 41619 time.
- 41620 • The data structure associated with each semaphore in the set need not be initialized. The
 41621 *semctl()* function with the command `SETVAL` or `SETALL` can be used to initialize each
 41622 semaphore.

41623 **RETURN VALUE**

41624 Upon successful completion, *semget()* shall return a non-negative integer, namely a semaphore
 41625 identifier; otherwise, it shall return `-1` and set *errno* to indicate the error.

41626 **ERRORS**

41627 The *semget()* function shall fail if:

- | | | |
|-------|----------|---|
| 41628 | [EACCES] | A semaphore identifier exists for <i>key</i> , but operation permission as specified by the low-order 9 bits of <i>semflg</i> would not be granted; see Section 2.7 (on page 39). |
| 41629 | | |
| 41630 | | |
| 41631 | [EEXIST] | A semaphore identifier exists for the argument <i>key</i> but $((semflg \& IPC_CREAT) \&\& (semflg \& IPC_EXCL))$ is non-zero. |
| 41632 | | |
| 41633 | [EINVAL] | The value of <i>nsems</i> is either less than or equal to 0 or greater than the system-imposed limit, or a semaphore identifier exists for the argument <i>key</i> , but the number of semaphores in the set associated with it is less than <i>nsems</i> and <i>nsems</i> is not equal to 0. |
| 41634 | | |
| 41635 | | |
| 41636 | | |

41637	[ENOENT]	A semaphore identifier does not exist for the argument <i>key</i> and (<i>semflg</i> &IPC_CREAT) is equal to 0.
41638		
41639	[ENOSPC]	A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores system-wide would be exceeded.
41640		

41641 **EXAMPLES**41642 **Creating a Semaphore Identifier**

41643 The following example gets a unique semaphore key using the *ftok()* function, then gets a
 41644 semaphore ID associated with that key using the *semget()* function (the first call also tests to
 41645 make sure the semaphore exists). If the semaphore does not exist, the program creates it, as
 41646 shown by the second call to *semget()*. In creating the semaphore for the queuing process, the
 41647 program attempts to create one semaphore with read/write permission for all. It also uses the
 41648 IPC_EXCL flag, which forces *semget()* to fail if the semaphore already exists.

41649 After creating the semaphore, the program uses a call to *semop()* to initialize it to the values in
 41650 the *sbuf* array. The number of processes that can execute concurrently without queuing is
 41651 initially set to 2. The final call to *semget()* creates a semaphore identifier that can be used later in
 41652 the program.

```

41653 #include <sys/types.h>
41654 #include <stdio.h>
41655 #include <sys/ipc.h>
41656 #include <sys/sem.h>
41657 #include <sys/stat.h>
41658 #include <errno.h>
41659 #include <unistd.h>
41660 #include <stdlib.h>
41661 #include <pwd.h>
41662 #include <fcntl.h>
41663 #include <limits.h>
41664 ...
41665 key_t semkey;
41666 int semid, pfd, fv;
41667 struct sembuf sbuf;
41668 char *lgn;
41669 char filename[PATH_MAX+1];
41670 struct stat outstat;
41671 struct passwd *pw;
41672 ...
41673 /* Get unique key for semaphore. */
41674 if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
41675     perror("IPC error: ftok"); exit(1);
41676 }
41677
41678 /* Get semaphore ID associated with this key. */
41679 if ((semid = semget(semkey, 0, 0)) == -1) {
41680     /* Semaphore does not exist - Create. */
41681     if ((semid = semget(semkey, 1, IPC_CREAT | IPC_EXCL | S_IRUSR |
41682         S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)) != -1)
41683     {
41684         /* Initialize the semaphore. */
41685         sbuf.sem_num = 0;
41686         sbuf.sem_op = 2; /* This is the number of runs
                           without queuing. */

```

```

41687         sbuf.sem_flg = 0;
41688         if (semop(semid, &sbuf, 1) == -1) {
41689             perror("IPC error: semop"); exit(1);
41690         }
41691     }
41692     else if (errno == EEXIST) {
41693         if ((semid = semget(semkey, 0, 0)) == -1) {
41694             perror("IPC error 1: semget"); exit(1);
41695         }
41696     }
41697     else {
41698         perror("IPC error 2: semget"); exit(1);
41699     }
41700 }
41701 ...

```

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication. Application developers who need to use IPC should design their applications so that modules using the IPC routines described in [Section 2.7](#) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 2.7](#) (on page 39), [Section 2.8](#) (on page 40), [semctl\(\)](#), [semop\(\)](#), [sem_close\(\)](#), [sem_destroy\(\)](#), [sem_getvalue\(\)](#), [sem_init\(\)](#), [sem_open\(\)](#), [sem_post\(\)](#), [sem_unlink\(\)](#), [sem_wait\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, [<sys/sem.h>](#)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

Issue 6

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/122 is applied, updating the DESCRIPTION from "each semaphore in the set shall not be initialized" to "each semaphore in the set need not be initialized".

41724 **NAME**

41725 semop — XSI semaphore operations

41726 **SYNOPSIS**

```
41727 XSI #include <sys/sem.h>
41728 int semop(int semid, struct sembuf *sops, size_t nsops);
```

41729 **DESCRIPTION**

41730 The *semop()* function operates on XSI semaphores (see the Base Definitions volume of
 41731 IEEE Std 1003.1-200x, Section 4.15, Semaphore). It is unspecified whether this function
 41732 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
 41733 page 40).

41734 The *semop()* function shall perform atomically a user-defined array of semaphore operations on
 41735 the set of semaphores associated with the semaphore identifier specified by the argument *semid*.

41736 The argument *sops* is a pointer to a user-defined array of semaphore operation structures. The
 41737 implementation shall not modify elements of this array unless the application uses
 41738 implementation-defined extensions.

41739 The argument *nsops* is the number of such structures in the array.

41740 Each structure, **sembuf**, includes the following members:

Member Type	Member Name	Description
short	<i>sem_num</i>	Semaphore number.
short	<i>sem_op</i>	Semaphore operation.
short	<i>sem_flg</i>	Operation flags.

41745 Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore
 41746 specified by *semid* and *sem_num*.

41747 The variable *sem_op* specifies one of three semaphore operations:

- 41748 1. If *sem_op* is a negative integer and the calling process has alter permission, one of the
 41749 following shall occur:
 - 41750 • If *semval*(see <sys/sem.h>) is greater than or equal to the absolute value of *sem_op*,
 41751 the absolute value of *sem_op* is subtracted from *semval*. Also, if (*sem_flg*
 41752 &SEM_UNDO) is non-zero, the absolute value of *sem_op* shall be added to the
 41753 *semadj* value of the calling process for the specified semaphore.
 - 41754 • If *semval* is less than the absolute value of *sem_op* and (*sem_flg* &IPC_NOWAIT) is
 41755 non-zero, *semop()* shall return immediately.
 - 41756 • If *semval* is less than the absolute value of *sem_op* and (*sem_flg* &IPC_NOWAIT) is 0,
 41757 *semop()* shall increment the *semncnt* associated with the specified semaphore and
 41758 suspend execution of the calling thread until one of the following conditions occurs:
 - 41759 — The value of *semval* becomes greater than or equal to the absolute value of
 41760 *sem_op*. When this occurs, the value of *semncnt* associated with the specified
 41761 semaphore shall be decremented, the absolute value of *sem_op* shall be
 41762 subtracted from *semval* and, if (*sem_flg* &SEM_UNDO) is non-zero, the
 41763 absolute value of *sem_op* shall be added to the *semadj* value of the calling
 41764 process for the specified semaphore.

- 41765 — The *semid* for which the calling thread is awaiting action is removed from the
 41766 system. When this occurs, *errno* shall be set equal to [EIDRM] and -1 shall be
 41767 returned.
- 41768 — The calling thread receives a signal that is to be caught. When this occurs, the
 41769 value of *semncnt* associated with the specified semaphore shall be
 41770 decremented, and the calling thread shall resume execution in the manner
 41771 prescribed in *sigaction()*.
- 41772 2. If *sem_op* is a positive integer and the calling process has alter permission, the value of
 41773 *sem_op* shall be added to *semval* and, if (*sem_flg* & SEM_UNDO) is non-zero, the value of
 41774 *sem_op* shall be subtracted from the *semadj* value of the calling process for the specified
 41775 semaphore.
- 41776 3. If *sem_op* is 0 and the calling process has read permission, one of the following shall occur:
- 41777 • If *semval* is 0, *semop()* shall return immediately.
 - 41778 • If *semval* is non-zero and (*sem_flg* & IPC_NOWAIT) is non-zero, *semop()* shall return
 41779 immediately.
 - 41780 • If *semval* is non-zero and (*sem_flg* & IPC_NOWAIT) is 0, *semop()* shall increment the
 41781 *semzcnt* associated with the specified semaphore and suspend execution of the
 41782 calling thread until one of the following occurs:
 - 41783 — The value of *semval* becomes 0, at which time the value of *semzcnt* associated
 41784 with the specified semaphore shall be decremented.
 - 41785 — The *semid* for which the calling thread is awaiting action is removed from the
 41786 system. When this occurs, *errno* shall be set equal to [EIDRM] and -1 shall be
 41787 returned.
 - 41788 — The calling thread receives a signal that is to be caught. When this occurs, the
 41789 value of *semzcnt* associated with the specified semaphore shall be
 41790 decremented, and the calling thread shall resume execution in the manner
 41791 prescribed in *sigaction()*.

41792 Upon successful completion, the value of *sempid* for each semaphore specified in the array
 41793 pointed to by *sops* shall be set equal to the process ID of the calling process.

41794 RETURN VALUE

41795 Upon successful completion, *semop()* shall return 0; otherwise, it shall return -1 and set *errno* to
 41796 indicate the error.

41797 ERRORS

41798 The *semop()* function shall fail if:

- | | | |
|-------|----------|--|
| 41799 | [E2BIG] | The value of <i>nsops</i> is greater than the system-imposed maximum. |
| 41800 | [EACCES] | Operation permission is denied to the calling process; see Section 2.7 (on page
41801 39). |
| 41802 | [EAGAIN] | The operation would result in suspension of the calling process but (<i>sem_flg</i>
41803 & IPC_NOWAIT) is non-zero. |
| 41804 | [EFBIG] | The value of <i>sem_num</i> is less than 0 or greater than or equal to the number of
41805 semaphores in the set associated with <i>semid</i> . |
| 41806 | [EIDRM] | The semaphore identifier <i>semid</i> is removed from the system. |
| 41807 | [EINTR] | The <i>semop()</i> function was interrupted by a signal. |

41808	[EINVAL]	The value of <i>semid</i> is not a valid semaphore identifier, or the number of individual semaphores for which the calling process requests a SEM_UNDO would exceed the system-imposed limit.
41809		
41810		
41811	[ENOSPC]	The limit on the number of individual processes requesting a SEM_UNDO would be exceeded.
41812		
41813	[ERANGE]	An operation would cause a <i>semval</i> to overflow the system-imposed limit, or an operation would cause a <i>semadj</i> value to overflow the system-imposed limit.
41814		
41815		

EXAMPLES

Setting Values in Semaphores

The following example sets the values of the two semaphores associated with the *semid* identifier to the values contained in the *sb* array.

```

41820 #include <sys/sem.h>
41821 ...
41822 int semid;
41823 struct sembuf sb[2];
41824 int nsops = 2;
41825 int result;

41826 /* Adjust value of semaphore in the semaphore array semid. */
41827 sb[0].sem_num = 0;
41828 sb[0].sem_op = -1;
41829 sb[0].sem_flg = SEM_UNDO | IPC_NOWAIT;
41830 sb[1].sem_num = 1;
41831 sb[1].sem_op = 1;
41832 sb[1].sem_flg = 0;

41833 result = semop(semid, sb, nsops);

```

Creating a Semaphore Identifier

The following example gets a unique semaphore key using the *ftok()* function, then gets a semaphore ID associated with that key using the *semget()* function (the first call also tests to make sure the semaphore exists). If the semaphore does not exist, the program creates it, as shown by the second call to *semget()*. In creating the semaphore for the queuing process, the program attempts to create one semaphore with read/write permission for all. It also uses the IPC_EXCL flag, which forces *semget()* to fail if the semaphore already exists.

After creating the semaphore, the program uses a call to *semop()* to initialize it to the values in the *sbuf* array. The number of processes that can execute concurrently without queuing is initially set to 2. The final call to *semget()* creates a semaphore identifier that can be used later in the program.

The final call to *semop()* acquires the semaphore and waits until it is free; the SEM_UNDO option releases the semaphore when the process exits, waiting until there are less than two processes running concurrently.

```

41848 #include <sys/types.h>
41849 #include <stdio.h>
41850 #include <sys/ipc.h>
41851 #include <sys/sem.h>
41852 #include <sys/stat.h>
41853 #include <errno.h>

```

```

41854     #include <unistd.h>
41855     #include <stdlib.h>
41856     #include <pwd.h>
41857     #include <fcntl.h>
41858     #include <limits.h>
41859     ...
41860     key_t semkey;
41861     int semid, pfd, fv;
41862     struct sembuf sbuf;
41863     char *lgn;
41864     char filename[PATH_MAX+1];
41865     struct stat outstat;
41866     struct passwd *pw;
41867     ...
41868     /* Get unique key for semaphore. */
41869     if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
41870         perror("IPC error: ftok"); exit(1);
41871     }
41872     /* Get semaphore ID associated with this key. */
41873     if ((semid = semget(semkey, 0, 0)) == -1) {
41874         /* Semaphore does not exist - Create. */
41875         if ((semid = semget(semkey, 1, IPC_CREAT | IPC_EXCL | S_IRUSR |
41876             S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)) != -1)
41877         {
41878             /* Initialize the semaphore. */
41879             sbuf.sem_num = 0;
41880             sbuf.sem_op = 2; /* This is the number of runs without queuing.
41881             sbuf.sem_flg = 0;
41882             if (semop(semid, &sbuf, 1) == -1) {
41883                 perror("IPC error: semop"); exit(1);
41884             }
41885         }
41886         else if (errno == EEXIST) {
41887             if ((semid = semget(semkey, 0, 0)) == -1) {
41888                 perror("IPC error 1: semget"); exit(1);
41889             }
41890         }
41891         else {
41892             perror("IPC error 2: semget"); exit(1);
41893         }
41894     }
41895     ...
41896     sbuf.sem_num = 0;
41897     sbuf.sem_op = -1;
41898     sbuf.sem_flg = SEM_UNDO;
41899     if (semop(semid, &sbuf, 1) == -1) {
41900         perror("IPC Error: semop"); exit(1);
41901     }

```

APPLICATION USAGE

41902 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
 41903 Application developers who need to use IPC should design their applications so that modules
 41904 using the IPC routines described in [Section 2.7](#) can be easily modified to use the alternative
 41905 interfaces.
 41906

semop()

41907

RATIONALE

41908

None.

41909

FUTURE DIRECTIONS

41910

None.

41911

SEE ALSO

41912

Section 2.7 (on page 39), Section 2.8 (on page 40), *exec*, *exit()*, *fork()*, *semctl()*, *semget()*, *sem_close()*, *sem_destroy()*, *sem_getvalue()*, *sem_init()*, *sem_open()*, *sem_post()*, *sem_unlink()*, *sem_wait()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<sys/ipc.h>`, `<sys/sem.h>`, `<sys/types.h>`

41913

41914

41915

41916

CHANGE HISTORY

41917

First released in Issue 2. Derived from Issue 2 of the SVID.

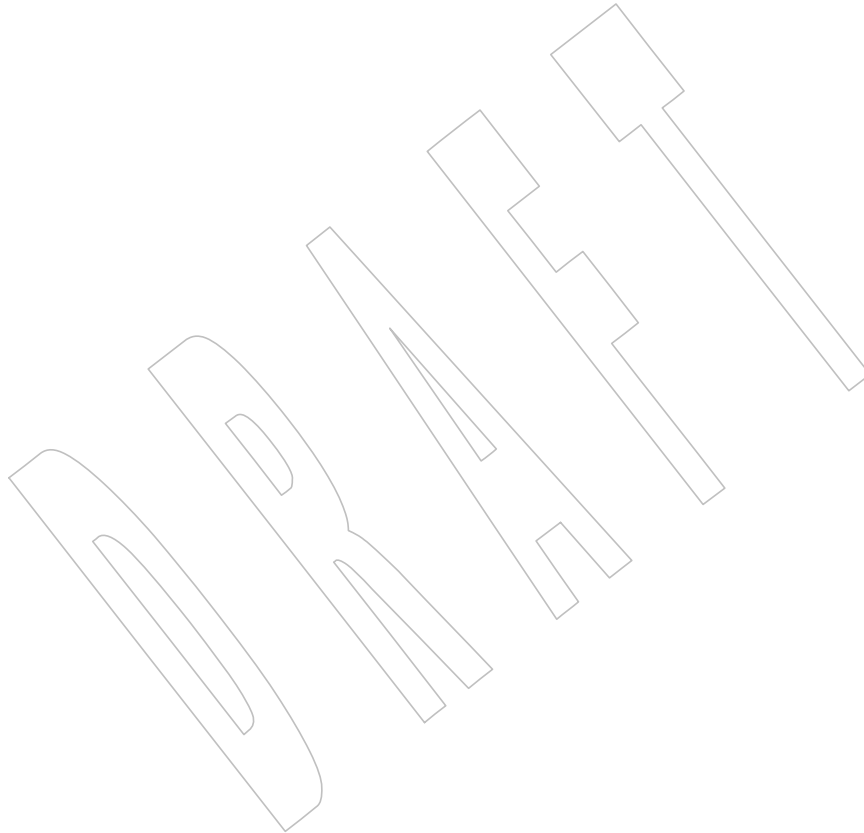
41918

Issue 5

41919

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

41920



NAME

send — send a message on a socket

SYNOPSIS

```
#include <sys/socket.h>

ssize_t send(int socket, const void *buffer, size_t length, int flags);
```

DESCRIPTION

The *send()* function shall initiate transmission of a message from the specified socket to its peer. The *send()* function shall send a message only when the socket is connected. If the socket is a connectionless-mode socket, the message shall be sent to the pre-specified peer address.

The *send()* function takes the following arguments:

<i>socket</i>	Specifies the socket file descriptor.
<i>buffer</i>	Points to the buffer containing the message to send.
<i>length</i>	Specifies the length of the message in bytes.
<i>flags</i>	Specifies the type of message transmission. Values of this argument are formed by logically OR'ing zero or more of the following flags:
MSG_EOR	Terminates a record (if supported by the protocol).
MSG_OOB	Sends out-of-band data on sockets that support out-of-band communications. The significance and semantics of out-of-band data are protocol-specific.
MSG_NOSIGNAL	Requests not to send the SIGPIPE signal if an attempt to send is made on a stream-oriented socket that is no longer connected. The [EPIPE] error shall still be returned.

The length of the message to be sent is specified by the *length* argument. If the message is too long to pass through the underlying protocol, *send()* shall fail and no data shall be transmitted.

Successful completion of a call to *send()* does not guarantee delivery of the message. A return value of -1 indicates only locally-detected errors.

send()

- 41963 [EBADF] The *socket* argument is not a valid file descriptor.
- 41964 [ECONNRESET] A connection was forcibly closed by a peer.
- 41965 [EDESTADDRREQ]
- 41966 The socket is not connection-mode and no peer address is set.
- 41967 [EINTR] A signal interrupted *send()* before any data was transmitted.
- 41968 [EMSGSIZE] The message is too large to be sent all at once, as the socket requires.
- 41969 [ENOTCONN] The socket is not connected.
- 41970 [ENOTSOCK] The *socket* argument does not refer to a socket.
- 41971 [EOPNOTSUPP] The *socket* argument is associated with a socket that does not support one or
- 41972 more of the values set in *flags*.
- 41973 [EPIPE] The socket is shut down for writing, or the socket is connection-mode and is
- 41974 no longer connected. In the latter case, and if the socket is of type
- 41975 SOCK_STREAM, the SIGPIPE signal is generated to the calling thread.
- 41976 The *send()* function may fail if:
- 41977 [EACCES] The calling process does not have the appropriate privileges.
- 41978 [EIO] An I/O error occurred while reading from or writing to the file system.
- 41979 [ENETDOWN] The local network interface used to reach the destination is down.
- 41980 [ENETUNREACH]
- 41981 No route to the network is present.
- 41982 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

EXAMPLES

41983 None.

41984

APPLICATION USAGE

41985 The *send()* function is equivalent to *sendto()* with a null pointer *dest_len* argument, and to *write()*

41986 if no flags are used.

41987

RATIONALE

41988 None.

41989

FUTURE DIRECTIONS

41990 None.

41991

SEE ALSO

41992 [connect\(\)](#), [getsockopt\(\)](#), [poll\(\)](#), [recv\(\)](#), [recvfrom\(\)](#), [recvmsg\(\)](#), [select\(\)](#), [sendmsg\(\)](#), [sendto\(\)](#),

41993 [setsockopt\(\)](#), [shutdown\(\)](#), [socket\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x,

41994 [<sys/socket.h>](#)

41995

CHANGE HISTORY

41996 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

41997

Issue 7

41998 Austin Group Interpretation 1003.1-2001 #035 is applied, updating the DESCRIPTION to clarify

41999 the behavior when the socket is a connectionless-mode socket.

42000

42001 The MSG_NOSIGNAL flag is added from The Open Group Technical Standard, 2006, Extended

42002 API Set Part 2.

42003 **NAME**
 42004 sendmsg — send a message on a socket using a message structure

42005 **SYNOPSIS**
 42006 #include <sys/socket.h>

42007 ssize_t sendmsg(int socket, const struct msghdr *message, int flags);

42008 **DESCRIPTION**

42009 The *sendmsg()* function shall send a message through a connection-mode or connectionless-
 42010 mode socket. If the socket is a connectionless-mode socket, the message shall be sent to the
 42011 address specified by **msghdr** if no pre-specified peer address has been set. If a peer address has
 42012 been pre-specified, either the message shall be sent to the address specified in **msghdr**
 42013 (overriding the pre-specified peer address), or the function shall return -1 and set *errno* to
 42014 [EISCONN]. If the socket is connection-mode, the destination address in **msghdr** shall be
 42015 ignored.

42016 The *sendmsg()* function takes the following arguments:

42017	<i>socket</i>	Specifies the socket file descriptor.
42018 42019 42020	<i>message</i>	Points to a msghdr structure, containing both the destination address and the buffers for the outgoing message. The length and format of the address depend on the address family of the socket. The <i>msg_flags</i> member is ignored.
42021 42022	<i>flags</i>	Specifies the type of message transmission. The application may specify 0 or the following flag:
42023	MSG_EOR	Terminates a record (if supported by the protocol).
42024 42025 42026	MSG_OOB	Sends out-of-band data on sockets that support out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
42027 42028 42029 42030	MSG_NOSIGNAL	Requests not to send the SIGPIPE signal if an attempt to send is made on a stream-oriented socket that is no longer connected. The [EPIPE] error shall still be returned.

42031 The *msg_iov* and *msg_iovlen* fields of *message* specify zero or more buffers containing the data to
 42032 be sent. *msg_iov* points to an array of **iovec** structures; *msg_iovlen* shall be set to the dimension of
 42033 this array. In each **iovec** structure, the *iov_base* field specifies a storage area and the *iov_len* field
 42034 gives its size in bytes. Some of these sizes can be zero. The data from each storage area indicated
 42035 by *msg_iov* is sent in turn.

42036 Successful completion of a call to *sendmsg()* does not guarantee delivery of the message. A
 42037 return value of -1 indicates only locally-detected errors.

42038 If space is not available at the sending socket to hold the message to be transmitted and the
 42039 socket file descriptor does not have O_NONBLOCK set, the *sendmsg()* function shall block until
 42040 space is available. If space is not available at the sending socket to hold the message to be
 42041 transmitted and the socket file descriptor does have O_NONBLOCK set, the *sendmsg()* function
 42042 shall fail.

42043 If the socket protocol supports broadcast and the specified address is a broadcast address for the
 42044 socket protocol, *sendmsg()* shall fail if the SO_BROADCAST option is not set for the socket.

42045 The socket in use may require the process to have appropriate privileges to use the *sendmsg()*
 42046 function.

RETURN VALUE

42047
42048 Upon successful completion, *sendmsg()* shall return the number of bytes sent. Otherwise, `-1`
42049 shall be returned and *errno* set to indicate the error.

ERRORS

42050 The *sendmsg()* function shall fail if:

42051 [EAGAIN] or [EWOULDBLOCK]

42052 The socket's file descriptor is marked `O_NONBLOCK` and the requested
42054 operation would block.

42055 [EAFNOSUPPORT]

42056 Addresses in the specified address family cannot be used with this socket.

42057 [EBADF] The *socket* argument is not a valid file descriptor.

42058 [ECONNRESET] A connection was forcibly closed by a peer.

42059 [EINTR] A signal interrupted *sendmsg()* before any data was transmitted.

42060 [EINVAL] The sum of the *iov_len* values overflows an `ssize_t`.

42061 [EMSGSIZE] The message is too large to be sent all at once (as the socket requires), or the
42062 *msg_iovlen* member of the `msghdr` structure pointed to by *message* is less than
42063 or equal to 0 or is greater than `{IOV_MAX}`.

42064 [ENOTCONN] The socket is connection-mode but is not connected.

42065 [ENOTSOCK] The *socket* argument does not refer to a socket.

42066 [EOPNOTSUPP] The *socket* argument is associated with a socket that does not support one or
42067 more of the values set in *flags*.

42068 [EPIPE] The socket is shut down for writing, or the socket is connection-mode and is
42069 no longer connected. In the latter case, and if the socket is of type
42070 `SOCK_STREAM`, the `SIGPIPE` signal is generated to the calling thread.

42071 If the address family of the socket is `AF_UNIX`, then *sendmsg()* shall fail if:

42072 [EIO] An I/O error occurred while reading from or writing to the file system.

42073 [ELOOP] A loop exists in symbolic links encountered during resolution of the pathname
42074 in the socket address.

42075 [ENAMETOOLONG]

42076 A component of a pathname exceeded `{NAME_MAX}` characters, or an entire
42077 pathname exceeded `{PATH_MAX}` characters.

42078 [ENOENT] A component of the pathname does not name an existing file or the path name
42079 is an empty string.

42080 [ENOTDIR] A component of the path prefix of the pathname in the socket address is not a
42081 directory.

42082 The *sendmsg()* function may fail if:

42083 [EACCES] Search permission is denied for a component of the path prefix; or write access
42084 to the named socket is denied.

42085 [EDESTADDRREQ]

42086 The socket is not connection-mode and does not have its peer address set, and
42087 no destination address was specified.

42088	[EHOSTUNREACH]	
42089		The destination host cannot be reached (probably because the host is down or
42090		a remote router cannot reach it).
42091	[EIO]	An I/O error occurred while reading from or writing to the file system.
42092	[EISCONN]	A destination address was specified and the socket is already connected.
42093	[ENETDOWN]	The local network interface used to reach the destination is down.
42094	[ENETUNREACH]	
42095		No route to the network is present.
42096	[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
42097	[ENOMEM]	Insufficient memory was available to fulfill the request.
42098		If the address family of the socket is AF_UNIX, then <i>sendmsg()</i> may fail if:
42099	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during
42100		resolution of the pathname in the socket address.
42101	[ENAMETOOLONG]	
42102		Pathname resolution of a symbolic link produced an intermediate result
42103		whose length exceeds {PATH_MAX}.
42104	EXAMPLES	
42105		Done.
42106	APPLICATION USAGE	
42107		The <i>select()</i> and <i>poll()</i> functions can be used to determine when it is possible to send more data.
42108	RATIONALE	
42109		None.
42110	FUTURE DIRECTIONS	
42111		None.
42112	SEE ALSO	
42113		<i>getsockopt()</i> , <i>poll()</i> , <i>recv()</i> , <i>recvfrom()</i> , <i>recvmsg()</i> , <i>select()</i> , <i>send()</i> , <i>sendto()</i> , <i>setsockopt()</i> ,
42114		<i>shutdown()</i> , <i>socket()</i> , the Base Definitions volume of IEEE Std 1003.1-200x, <sys/socket.h>
42115	CHANGE HISTORY	
42116		First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
42117		The wording of the mandatory [ELOOP] error condition is updated, and a second optional
42118		[ELOOP] error condition is added.
42119	Issue 7	
42120		Austin Group Interpretation 1003.1-2001 #073 is applied, updating the DESCRIPTION.
42121		The MSG_NOSIGNAL flag is added from The Open Group Technical Standard, 2006, Extended
42122		API Set Part 2.

42123 **NAME**42124 `sendto` — send a message on a socket42125 **SYNOPSIS**

```
42126 #include <sys/socket.h>
42127
42127 ssize_t sendto(int socket, const void *message, size_t length,
42128               int flags, const struct sockaddr *dest_addr,
42129               socklen_t dest_len);
```

42130 **DESCRIPTION**

42131 The `sendto()` function shall send a message through a connection-mode or connectionless-mode
 42132 socket. If the socket is a connectionless-mode socket, the message shall be sent to the address
 42133 specified by `dest_addr` if no pre-specified peer address has been set. If a peer address has been
 42134 pre-specified, either the message shall be sent to the address specified by `dest_addr` (overriding
 42135 the pre-specified peer address), or the function shall return `-1` and set `errno` to `[EISCONN]`. If
 42136 the socket is connection-mode, `dest_addr` shall be ignored.

42137 The `sendto()` function takes the following arguments:

42138	<code>socket</code>	Specifies the socket file descriptor.
42139	<code>message</code>	Points to a buffer containing the message to be sent.
42140	<code>length</code>	Specifies the size of the message in bytes.
42141	<code>flags</code>	Specifies the type of message transmission. Values of this argument are 42142 formed by logically OR'ing zero or more of the following flags:
42143	<code>MSG_EOR</code>	Terminates a record (if supported by the protocol).
42144	<code>MSG_OOB</code>	Sends out-of-band data on sockets that support out-of- 42145 band data. The significance and semantics of out-of- 42146 band data are protocol-specific.
42147	<code>MSG_NOSIGNAL</code>	Requests not to send the SIGPIPE signal if an attempt to 42148 send is made on a stream-oriented socket that is no 42149 longer connected. The <code>[EPIPE]</code> error shall still be 42150 returned.
42151	<code>dest_addr</code>	Points to a <code>sockaddr</code> structure containing the destination address. The length 42152 and format of the address depend on the address family of the socket.
42153	<code>dest_len</code>	Specifies the length of the <code>sockaddr</code> structure pointed to by the <code>dest_addr</code> 42154 argument.

42155 If the socket protocol supports broadcast and the specified address is a broadcast address for the
 42156 socket protocol, `sendto()` shall fail if the `SO_BROADCAST` option is not set for the socket.

42157 The `dest_addr` argument specifies the address of the target. The `length` argument specifies the
 42158 length of the message.

42159 Successful completion of a call to `sendto()` does not guarantee delivery of the message. A return
 42160 value of `-1` indicates only locally-detected errors.

42161 If space is not available at the sending socket to hold the message to be transmitted and the
 42162 socket file descriptor does not have `O_NONBLOCK` set, `sendto()` shall block until space is
 42163 available. If space is not available at the sending socket to hold the message to be transmitted
 42164 and the socket file descriptor does have `O_NONBLOCK` set, `sendto()` shall fail.

42165 The socket in use may require the process to have appropriate privileges to use the `sendto()`

42166 function.

42167 RETURN VALUE

42168 Upon successful completion, *sendto()* shall return the number of bytes sent. Otherwise, `-1` shall
42169 be returned and *errno* set to indicate the error.

42170 ERRORS

42171 The *sendto()* function shall fail if:

42172 [EAFNOSUPPORT]

42173 Addresses in the specified address family cannot be used with this socket.

42174 [EAGAIN] or [EWOULDBLOCK]

42175 The socket's file descriptor is marked `O_NONBLOCK` and the requested
42176 operation would block.

42177 [EBADF] The *socket* argument is not a valid file descriptor.

42178 [ECONNRESET] A connection was forcibly closed by a peer.

42179 [EINTR] A signal interrupted *sendto()* before any data was transmitted.

42180 [EMSGSIZE] The message is too large to be sent all at once, as the socket requires.

42181 [ENOTCONN] The socket is connection-mode but is not connected.

42182 [ENOTSOCK] The *socket* argument does not refer to a socket.

42183 [EOPNOTSUPP] The *socket* argument is associated with a socket that does not support one or
42184 more of the values set in *flags*.

42185 [EPIPE] The socket is shut down for writing, or the socket is connection-mode and is
42186 no longer connected. In the latter case, and if the socket is of type
42187 `SOCK_STREAM`, the `SIGPIPE` signal is generated to the calling thread.

42188 If the address family of the socket is `AF_UNIX`, then *sendto()* shall fail if:

42189 [EIO] An I/O error occurred while reading from or writing to the file system.

42190 [ELOOP] A loop exists in symbolic links encountered during resolution of the pathname
42191 in the socket address.

42192 [ENAMETOOLONG]

42193 A component of a pathname exceeded `{NAME_MAX}` characters, or an entire
42194 pathname exceeded `{PATH_MAX}` characters.

42195 [ENOENT] A component of the pathname does not name an existing file or the pathname
42196 is an empty string.

42197 [ENOTDIR] A component of the path prefix of the pathname in the socket address is not a
42198 directory.

42199 The *sendto()* function may fail if:

42200 [EACCES] Search permission is denied for a component of the path prefix; or write access
42201 to the named socket is denied.

42202 [EDESTADDRREQ]

42203 The socket is not connection-mode and does not have its peer address set, and
42204 no destination address was specified.

42205 [EHOSTUNREACH]

42206 The destination host cannot be reached (probably because the host is down or
42207 a remote router cannot reach it).

42208	[EINVAL]	The <i>dest_len</i> argument is not a valid length for the address family.
42209	[EIO]	An I/O error occurred while reading from or writing to the file system.
42210	[EISCONN]	A destination address was specified and the socket is already connected.
42211	[ENETDOWN]	The local network interface used to reach the destination is down.
42212	[ENETUNREACH]	
42213		No route to the network is present.
42214	[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
42215	[ENOMEM]	Insufficient memory was available to fulfill the request.
42216		If the address family of the socket is AF_UNIX, then <i>sendto()</i> may fail if:
42217	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the pathname in the socket address.
42218		
42219	[ENAMETOOLONG]	
42220		Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.
42221		

EXAMPLES

None.

APPLICATION USAGEThe *select()* and *poll()* functions can be used to determine when it is possible to send more data.**RATIONALE**

None.

FUTURE DIRECTIONS

None.

SEE ALSO*getsockopt()*, *poll()*, *recv()*, *recvfrom()*, *recvmsg()*, *select()*, *send()*, *sendmsg()*, *setsockopt()*, *shutdown()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-200x, <sys/socket.h>**CHANGE HISTORY**

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

Issue 7

Austin Group Interpretations 1003.1-2001 #035 and #073 are applied, updating the [EISCONN] error and the DESCRIPTION.

The MSG_NOSIGNAL flag is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

42242 **NAME**

42243 setbuf — assign buffering to a stream

42244 **SYNOPSIS**

42245 #include <stdio.h>

42246 void setbuf(FILE *restrict stream, char *restrict buf);

42247 **DESCRIPTION**42248 CX The functionality described on this reference page is aligned with the ISO C standard. Any
42249 conflict between the requirements described here and the ISO C standard is unintentional. This
42250 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

42251 Except that it returns no value, the function call:

42252 setbuf(stream, buf)

42253 shall be equivalent to:

42254 setvbuf(stream, buf, _IOFBF, BUFSIZ)

42255 if *buf* is not a null pointer, or to:

42256 setvbuf(stream, buf, _IONBF, BUFSIZ)

42257 if *buf* is a null pointer.42258 **RETURN VALUE**42259 The *setbuf()* function shall not return a value.42260 **ERRORS**

42261 No errors are defined.

42262 **EXAMPLES**

42263 None.

42264 **APPLICATION USAGE**42265 A common source of error is allocating buffer space as an “automatic” variable in a code block,
42266 and then failing to close the stream in the same block.42267 With *setbuf()*, allocating a buffer of BUFSIZ bytes does not necessarily imply that all of BUFSIZ
42268 bytes are used for the buffer area.42269 **RATIONALE**

42270 None.

42271 **FUTURE DIRECTIONS**

42272 None.

42273 **SEE ALSO**42274 *fopen()*, *setvbuf()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>42275 **CHANGE HISTORY**

42276 First released in Issue 1. Derived from Issue 1 of the SVID.

42277 **Issue 6**42278 The prototype for *setbuf()* is updated for alignment with the ISO/IEC 9899:1999 standard.

42279 **NAME**42280 `setegid` — set the effective group ID42281 **SYNOPSIS**42282 `#include <unistd.h>`42283 `int setegid(gid_t gid);`42284 **DESCRIPTION**

42285 If *gid* is equal to the real group ID or the saved set-group-ID, or if the process has appropriate
 42286 privileges, *setegid()* shall set the effective group ID of the calling process to *gid*; the real group
 42287 ID, saved set-group-ID, and any supplementary group IDs shall remain unchanged.

42288 The *setegid()* function shall not affect the supplementary group list in any way.

42289 **RETURN VALUE**

42290 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and *errno* set to
 42291 indicate the error.

42292 **ERRORS**

42293 The *setegid()* function shall fail if:

42294 [EINVAL] The value of the *gid* argument is invalid and is not supported by the
 42295 implementation.

42296 [EPERM] The process does not have appropriate privileges and *gid* does not match the
 42297 real group ID or the saved set-group-ID.

42298 **EXAMPLES**

42299 None.

42300 **APPLICATION USAGE**

42301 None.

42302 **RATIONALE**42303 Refer to the RATIONALE section in *setuid()*.42304 **FUTURE DIRECTIONS**

42305 None.

42306 **SEE ALSO**

42307 *exec*, *getegid()*, *geteuid()*, *getgid()*, *getuid()*, *seteuid()*, *setgid()*, *setregid()*, *setreuid()*, *setuid()*, the
 42308 Base Definitions volume of IEEE Std 1003.1-200x, `<sys/types.h>`, `<unistd.h>`

42309 **CHANGE HISTORY**

42310 First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

42311 **NAME**
 42312 setenv — add or change environment variable

42313 **SYNOPSIS**

```
42314 CX #include <stdlib.h>
42315 int setenv(const char *envname, const char *envval, int overwrite);
```

42316 **DESCRIPTION**

42317 The *setenv()* function shall update or add a variable in the environment of the calling process.
 42318 The *envname* argument points to a string containing the name of an environment variable to be
 42319 added or altered. The environment variable shall be set to the value to which *envval* points. The
 42320 function shall fail if *envname* points to a string which contains an '=' character. If the
 42321 environment variable named by *envname* already exists and the value of *overwrite* is non-zero,
 42322 the function shall return success and the environment shall be updated. If the environment
 42323 variable named by *envname* already exists and the value of *overwrite* is zero, the function shall
 42324 return success and the environment shall remain unchanged.

42325 If the application modifies *environ* or the pointers to which it points, the behavior of *setenv()* is
 42326 undefined. The *setenv()* function shall update the list of pointers to which *environ* points.

42327 The strings described by *envname* and *envval* are copied by this function.

42328 The *setenv()* function need not be thread-safe. A function that is not required to be thread-safe is
 42329 not required to be reentrant.

42330 **RETURN VALUE**

42331 Upon successful completion, zero shall be returned. Otherwise, -1 shall be returned, *errno* set to
 42332 indicate the error, and the environment shall be unchanged.

42333 **ERRORS**

42334 The *setenv()* function shall fail if:

42335 [EINVAL] The *name* argument is a null pointer, points to an empty string, or points to a
 42336 string containing an '=' character.

42337 [ENOMEM] Insufficient memory was available to add a variable or its value to the
 42338 environment.

42339 **EXAMPLES**

42340 None.

42341 **APPLICATION USAGE**

42342 See *exec*, for restrictions on changing the environment in multi-threaded applications.

42343 **RATIONALE**

42344 Unanticipated results may occur if *setenv()* changes the external variable *environ*. In particular, if
 42345 the optional *envp* argument to *main()* is present, it is not changed, and thus may point to an
 42346 obsolete copy of the environment (as may any other copy of *environ*). However, other than the
 42347 aforementioned restriction, the developers of IEEE Std 1003.1-200x intended that the traditional
 42348 method of walking through the environment by way of the *environ* pointer must be supported.

42349 It was decided that *setenv()* should be required by this revision because it addresses a piece of
 42350 missing functionality, and does not impose a significant burden on the implementor.

42351 There was considerable debate as to whether the System V *putenv()* function or the BSD *setenv()*
 42352 function should be required as a mandatory function. The *setenv()* function was chosen because
 42353 it permitted the implementation of the *unsetenv()* function to delete environmental variables,

42354 without specifying an additional interface. The *putenv()* function is available as part of the XSI
42355 option.

42356 The standard developers considered requiring that *setenv()* indicate an error when a call to it
42357 would result in exceeding {ARG_MAX}. The requirement was rejected since the condition might
42358 be temporary, with the application eventually reducing the environment size. The ultimate
42359 success or failure depends on the size at the time of a call to *exec*, which returns an indication of
42360 this error condition.

42361 **FUTURE DIRECTIONS**

42362 None.

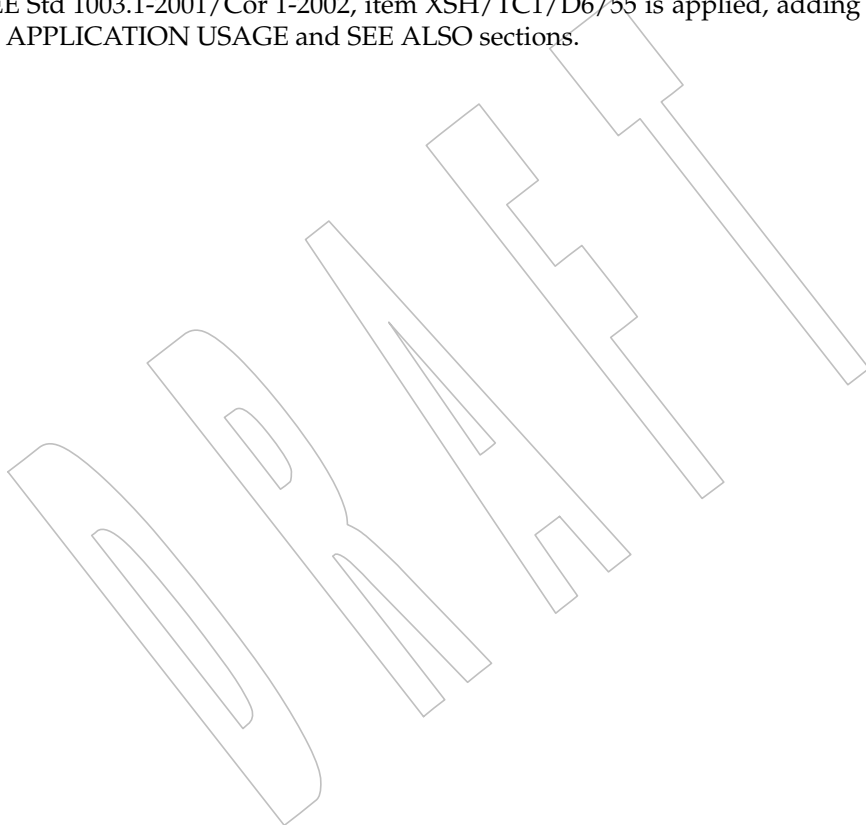
42363 **SEE ALSO**

42364 *exec*, *getenv()*, *unsetenv()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdlib.h>`,
42365 `<sys/types.h>`, `<unistd.h>`

42366 **CHANGE HISTORY**

42367 First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

42368 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/55 is applied, adding references to *exec* in
42369 the APPLICATION USAGE and SEE ALSO sections.



42370 **NAME**

42371 seteuid — set effective user ID

42372 **SYNOPSIS**

42373 #include <unistd.h>

42374 int seteuid(uid_t uid);

42375 **DESCRIPTION**

42376 If *uid* is equal to the real user ID or the saved set-user-ID, or if the process has appropriate
 42377 privileges, *seteuid()* shall set the effective user ID of the calling process to *uid*; the real user ID
 42378 and saved set-user-ID shall remain unchanged.

42379 The *seteuid()* function shall not affect the supplementary group list in any way.

42380 **RETURN VALUE**

42381 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and *errno* set to
 42382 indicate the error.

42383 **ERRORS**

42384 The *seteuid()* function shall fail if:

42385 [EINVAL] The value of the *uid* argument is invalid and is not supported by the
 42386 implementation.

42387 [EPERM] The process does not have appropriate privileges and *uid* does not match the
 42388 real user ID or the saved set-user-ID.

42389 **EXAMPLES**

42390 None.

42391 **APPLICATION USAGE**

42392 None.

42393 **RATIONALE**42394 Refer to the RATIONALE section in *setuid()*.42395 **FUTURE DIRECTIONS**

42396 None.

42397 **SEE ALSO**

42398 *exec*, *getegid()*, *geteuid()*, *getgid()*, *getuid()*, *setegid()*, *setgid()*, *setregid()*, *setreuid()*, *setuid()*, the
 42399 Base Definitions volume of IEEE Std 1003.1-200x, <sys/types.h>, <unistd.h>

42400 **CHANGE HISTORY**

42401 First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

42402 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/123 is applied, making an editorial
 42403 correction to the [EPERM] error in the ERRORS section.

42404 **NAME**42405 `setgid` — set-group-ID42406 **SYNOPSIS**42407 `#include <unistd.h>`42408 `int setgid(gid_t gid);`42409 **DESCRIPTION**42410 If the process has appropriate privileges, `setgid()` shall set the real group ID, effective group ID,
42411 and the saved set-group-ID of the calling process to `gid`.42412 If the process does not have appropriate privileges, but `gid` is equal to the real group ID or the
42413 saved set-group-ID, `setgid()` shall set the effective group ID to `gid`; the real group ID and saved
42414 set-group-ID shall remain unchanged.42415 The `setgid()` function shall not affect the supplementary group list in any way.

42416 Any supplementary group IDs of the calling process shall remain unchanged.

42417 **RETURN VALUE**42418 Upon successful completion, 0 is returned. Otherwise, -1 shall be returned and `errno` set to
42419 indicate the error.42420 **ERRORS**42421 The `setgid()` function shall fail if:42422 [EINVAL] The value of the `gid` argument is invalid and is not supported by the
42423 implementation.42424 [EPERM] The process does not have appropriate privileges and `gid` does not match the
42425 real group ID or the saved set-group-ID.42426 **EXAMPLES**

42427 None.

42428 **APPLICATION USAGE**

42429 None.

42430 **RATIONALE**42431 Refer to the RATIONALE section in `setuid()`.42432 **FUTURE DIRECTIONS**

42433 None.

42434 **SEE ALSO**42435 `exec`, `getegid()`, `geteuid()`, `getgid()`, `getuid()`, `setegid()`, `seteuid()`, `setregid()`, `setreuid()`, `setuid()`, the
42436 Base Definitions volume of IEEE Std 1003.1-200x, `<sys/types.h>`, `<unistd.h>`42437 **CHANGE HISTORY**

42438 First released in Issue 1. Derived from Issue 1 of the SVID.

42439 **Issue 6**42440 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.42441 The following new requirements on POSIX implementations derive from alignment with the
42442 Single UNIX Specification:

- 42443
- The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
42444 required for conforming implementations of previous POSIX specifications, it was not
42445 required for UNIX applications.

42446

42447

42448

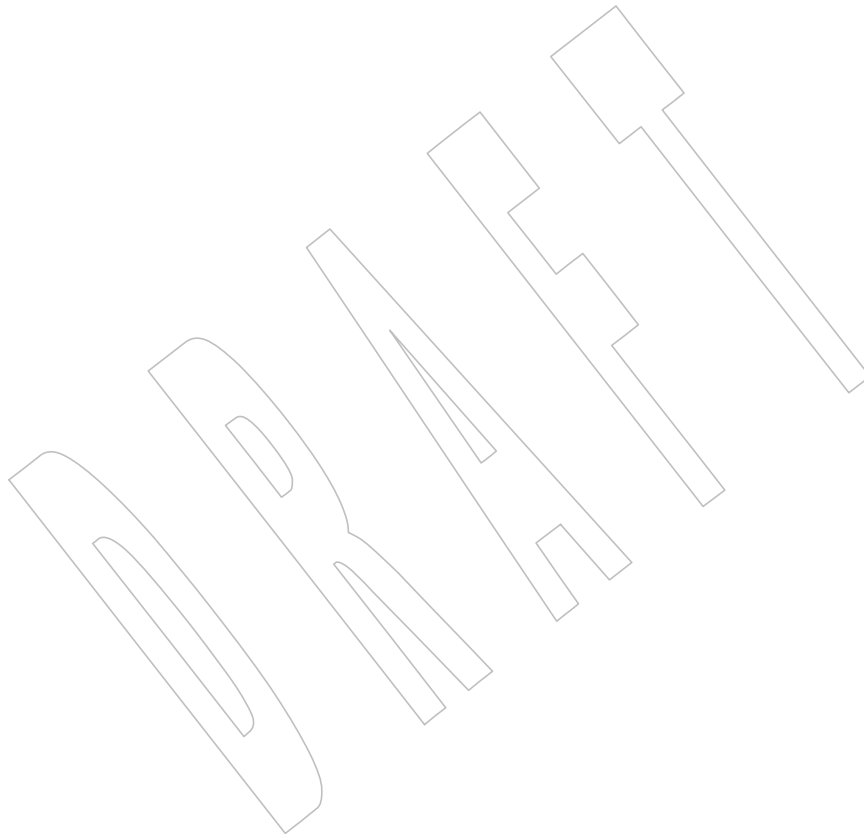
42449

42450

- Functionality associated with `_POSIX_SAVED_IDS` is now mandated. This is a FIPS requirement.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The effects of `setgid()` in processes without appropriate privileges are changed.
- A requirement that the supplementary group list is not affected is added.



setgrent()

42451 **NAME**
42452 setgrent — reset the group database to the first entry

42453 **SYNOPSIS**

42454 XSI #include <grp.h>
42455 void setgrent(void);

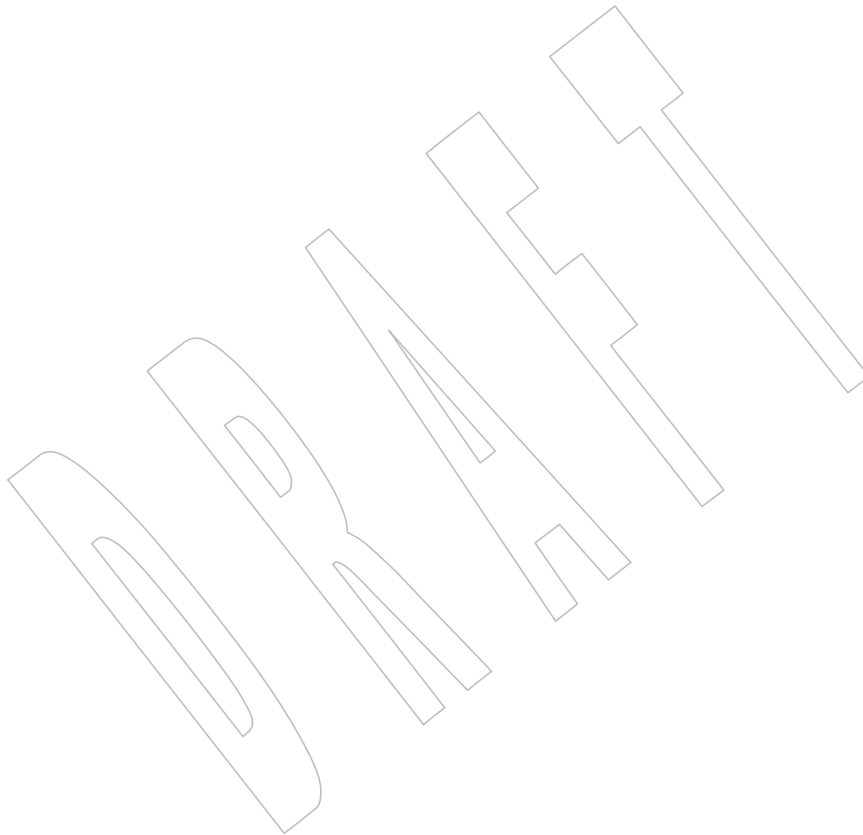
42456 **DESCRIPTION**

42457 Refer to *endgrent()*.

42458 **NAME**
42459 sethostent — network host database functions

42460 **SYNOPSIS**
42461 #include <netdb.h>
42462 void sethostent(int stayopen);

42463 **DESCRIPTION**
42464 Refer to *endhostent()*.



setitimer()

42465

NAME

42466

setitimer — set the value of an interval timer

42467

SYNOPSIS

42468

OB XSI `#include <sys/time.h>`

42469

`int setitimer(int which, const struct itimerval *restrict value,`

42470

`struct itimerval *restrict ovalue);`

42471

DESCRIPTION

42472

Refer to [getitimer\(\)](#).

42473 **NAME**
 42474 setjmp — set jump point for a non-local goto

42475 **SYNOPSIS**
 42476 #include <setjmp.h>
 42477 int setjmp(jmp_buf env);

42478 DESCRIPTION

42479 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 42480 conflict between the requirements described here and the ISO C standard is unintentional. This
 42481 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

42482 A call to *setjmp()* shall save the calling environment in its *env* argument for later use by
 42483 *longjmp()*.

42484 It is unspecified whether *setjmp()* is a macro or a function. If a macro definition is suppressed in
 42485 order to access an actual function, or a program defines an external identifier with the name
 42486 *setjmp*, the behavior is undefined.

42487 An application shall ensure that an invocation of *setjmp()* appears in one of the following
 42488 contexts only:

- 42489 • The entire controlling expression of a selection or iteration statement
- 42490 • One operand of a relational or equality operator with the other operand an integral
 42491 constant expression, with the resulting expression being the entire controlling expression
 42492 of a selection or iteration statement
- 42493 • The operand of a unary '!' operator with the resulting expression being the entire
 42494 controlling expression of a selection or iteration
- 42495 • The entire expression of an expression statement (possibly cast to **void**)

42496 If the invocation appears in any other context, the behavior is undefined.

42497 RETURN VALUE

42498 If the return is from a direct invocation, *setjmp()* shall return 0. If the return is from a call to
 42499 *longjmp()*, *setjmp()* shall return a non-zero value.

42500 ERRORS

42501 No errors are defined.

42502 EXAMPLES

42503 None.

42504 APPLICATION USAGE

42505 In general, *sigsetjmp()* is more useful in dealing with errors and interrupts encountered in a low-
 42506 level subroutine of a program.

42507 RATIONALE

42508 None.

42509 FUTURE DIRECTIONS

42510 None.

42511 SEE ALSO

42512 *longjmp()*, *sigsetjmp()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**setjmp.h**>

setjmp()

42513

CHANGE HISTORY

42514

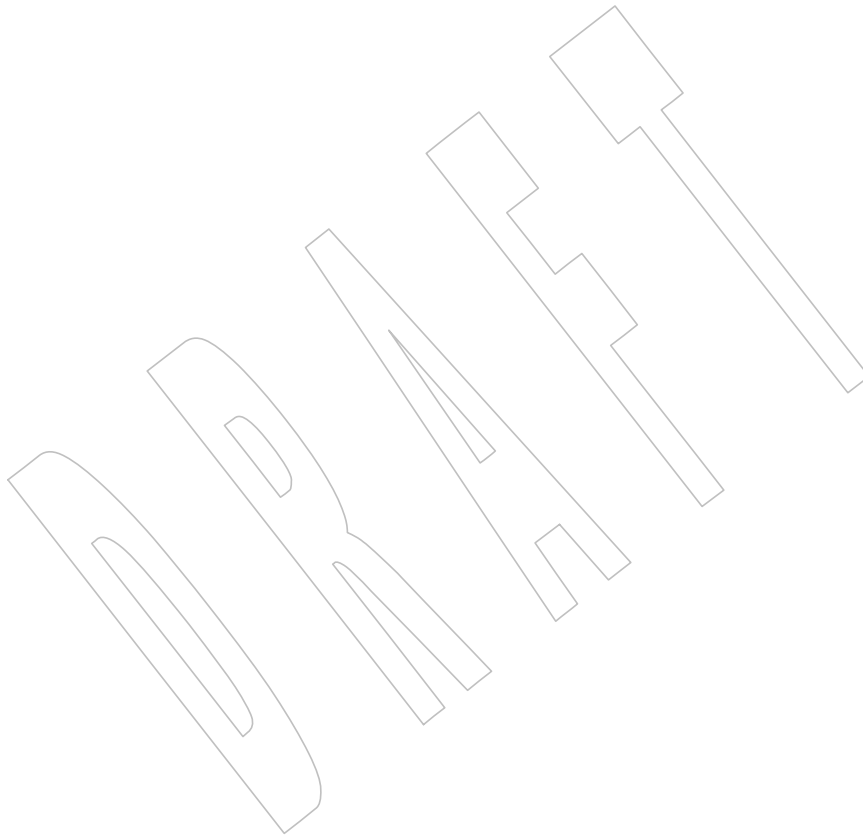
First released in Issue 1. Derived from Issue 1 of the SVID.

42515

Issue 6

42516

The normative text is updated to avoid use of the term “must” for application requirements.



42517 **NAME**42518 setkey — set encoding key (**CRYPT**)42519 **SYNOPSIS**

```
42520 XSI #include <stdlib.h>
42521 void setkey(const char *key);
```

42522 **DESCRIPTION**

42523 The *setkey()* function provides access to an implementation-defined encoding algorithm. The
 42524 argument of *setkey()* is an array of length 64 bytes containing only the bytes with numerical
 42525 value of 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is
 42526 ignored; this gives a 56-bit key which is used by the algorithm. This is the key that shall be used
 42527 with the algorithm to encode a string *block* passed to *encrypt()*.

42528 The *setkey()* function shall not change the setting of *errno* if successful. An application wishing to
 42529 check for error situations should set *errno* to 0 before calling *setkey()*. If *errno* is non-zero on
 42530 return, an error has occurred.

42531 The *setkey()* function need not be thread-safe. A function that is not required to be thread-safe is
 42532 not required to be reentrant.

42533 **RETURN VALUE**

42534 No values are returned.

42535 **ERRORS**42536 The *setkey()* function shall fail if:

42537 [ENOSYS] The functionality is not supported on this implementation.

42538 **EXAMPLES**

42539 None.

42540 **APPLICATION USAGE**

42541 Decoding need not be implemented in all environments. This is related to government
 42542 restrictions in some countries on encryption and decryption routines. Historical practice has
 42543 been to ship a different version of the encryption library without the decryption feature in the
 42544 routines supplied. Thus the exported version of *encrypt()* does encoding but not decoding.

42545 **RATIONALE**

42546 None.

42547 **FUTURE DIRECTIONS**

42548 None.

42549 **SEE ALSO**42550 *crypt()*, *encrypt()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**42551 **CHANGE HISTORY**

42552 First released in Issue 1. Derived from Issue 1 of the SVID.

42553 **Issue 5**42554 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

42555 **NAME**

42556 setlocale — set program locale

42557 **SYNOPSIS**

42558 #include <locale.h>

42559 char *setlocale(int *category*, const char **locale*);42560 **DESCRIPTION**

42561 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 42562 conflict between the requirements described here and the ISO C standard is unintentional. This
 42563 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

42564 The *setlocale()* function selects the appropriate piece of the locale of the process, as specified by
 42565 the *category* and *locale* arguments, and may be used to change or query the entire locale of the
 42566 process or portions thereof. The value *LC_ALL* for *category* names the entire locale of the process;
 42567 other values for *category* name only a part of the locale of the process:

42568 *LC_COLLATE* Affects the behavior of regular expressions and the collation functions.

42569 *LC_CTYPE* Affects the behavior of regular expressions, character classification, character
 42570 conversion functions, and wide-character functions.

42571 CX *LC_MESSAGES* Affects what strings are expected by commands and utilities as affirmative or
 42572 negative responses.

42573 XSI It also affects what strings are given by commands and utilities as affirmative
 42574 or negative responses, and the content of messages.

42575 *LC_MONETARY* Affects the behavior of functions that handle monetary values.

42576 *LC_NUMERIC* Affects the behavior of functions that handle numeric values.

42577 *LC_TIME* Affects the behavior of the time conversion functions.

42578 The *locale* argument is a pointer to a character string containing the required setting of *category*.
 42579 The contents of this string are implementation-defined. In addition, the following preset values
 42580 of *locale* are defined for all settings of *category*:

42581 CX "POSIX" Specifies the minimal environment for C-language translation called the
 42582 POSIX locale. If *setlocale()* is not invoked, the POSIX locale is the default at
 42583 entry to *main()*.

42584 "C" Equivalent to "POSIX".

42585 CX "" Specifies an implementation-defined native environment. The determination
 42586 of the name of the new locale for the specified category depends on the value
 42587 of the associated environment variables, *LC_** and *LANG*; see the Base
 42588 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale and the Base
 42589 Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment
 42590 Variables.

42591 A null pointer Used to direct *setlocale()* to query the current internationalized environment
 42592 and return the name of the locale.

42593 CX Setting all of the categories of the locale of the process is similar to successively setting each
 42594 individual category of the locale of the process, except that all error checking is done before any
 42595 actions are performed. To set all the categories of the locale of the process, *setlocale()* is invoked
 42596 as:

42597 `setlocale(LC_ALL, "");`

42598 In this case, *setlocale()* shall first verify that the values of all the environment variables it needs
 42599 according to the precedence rules (described in the Base Definitions volume of
 42600 IEEE Std 1003.1-200x, Chapter 8, Environment Variables) indicate supported locales. If the value
 42601 of any of these environment variable searches yields a locale that is not supported (and non-
 42602 null), *setlocale()* shall return a null pointer and the locale of the process shall not be changed. If
 42603 all environment variables name supported locales, *setlocale()* shall proceed as if it had been
 42604 called for each category, using the appropriate value from the associated environment variable
 42605 or from the implementation-defined default if there is no such value.

42606 The locale state is common to all threads within a process.

42607 RETURN VALUE

42608 Upon successful completion, *setlocale()* shall return the string associated with the specified
 42609 category for the new locale. Otherwise, *setlocale()* shall return a null pointer and the locale of the
 42610 process is not changed.

42611 A null pointer for *locale* causes *setlocale()* to return a pointer to the string associated with the
 42612 *category* for the current locale of the process. The locale of the process shall not be changed.

42613 The string returned by *setlocale()* is such that a subsequent call with that string and its associated
 42614 *category* shall restore that part of the locale of the process. The application shall not modify the
 42615 string returned which may be overwritten by a subsequent call to *setlocale()*.

42616 ERRORS

42617 No errors are defined.

42618 EXAMPLES

42619 None.

42620 APPLICATION USAGE

42621 The following code illustrates how a program can initialize the international environment for
 42622 one language, while selectively modifying the locale of the process such that regular expressions
 42623 and string operations can be applied to text recorded in a different language:

```
42624 setlocale(LC_ALL, "De");
42625 setlocale(LC_COLLATE, "Fr@dict");
```

42626 Internationalized programs must call *setlocale()* to initiate a specific language operation. This can
 42627 be done by calling *setlocale()* as follows:

```
42628 setlocale(LC_ALL, "");
```

42629 Changing the setting of *LC_MESSAGES* has no effect on catalogs that have already been opened
 42630 by calls to *catopen()*.

42631 RATIONALE

42632 The ISO C standard defines a collection of functions to support internationalization. One of the
 42633 most significant aspects of these functions is a facility to set and query the *international*
 42634 *environment*. The international environment is a repository of information that affects the
 42635 behavior of certain functionality, namely:

- 42636 1. Character handling
- 42637 2. Collating
- 42638 3. Date/time formatting
- 42639 4. Numeric editing
- 42640 5. Monetary formatting

6. Messaging

The *setlocale()* function provides the application developer with the ability to set all or portions, called *categories*, of the international environment. These categories correspond to the areas of functionality mentioned above. The syntax for *setlocale()* is as follows:

```
char *setlocale(int category, const char *locale);
```

where *category* is the name of one of following categories, namely:

```
LC_COLLATE
LC_CTYPE
LC_MESSAGES
LC_MONETARY
LC_NUMERIC
LC_TIME
```

In addition, a special value called *LC_ALL* directs *setlocale()* to set all categories.

There are two primary uses of *setlocale()*:

1. Querying the international environment to find out what it is set to
2. Setting the international environment, or *locale*, to a specific value

The behavior of *setlocale()* in these two areas is described below. Since it is difficult to describe the behavior in words, examples are used to illustrate the behavior of specific uses.

To query the international environment, *setlocale()* is invoked with a specific category and the NULL pointer as the locale. The NULL pointer is a special directive to *setlocale()* that tells it to query rather than set the international environment. The following syntax is used to query the name of the international environment:

```
setlocale({LC_ALL, LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, \
          LC_NUMERIC, LC_TIME}, (char *) NULL);
```

The *setlocale()* function shall return the string corresponding to the current international environment. This value may be used by a subsequent call to *setlocale()* to reset the international environment to this value. However, it should be noted that the return value from *setlocale()* may be a pointer to a static area within the function and is not guaranteed to remain unchanged (that is, it may be modified by a subsequent call to *setlocale()*). Therefore, if the purpose of calling *setlocale()* is to save the value of the current international environment so it can be changed and reset later, the return value should be copied to an array of **char** in the calling program.

There are three ways to set the international environment with *setlocale()*:

setlocale(category, string)

This usage sets a specific *category* in the international environment to a specific value corresponding to the value of the *string*. A specific example is provided below:

```
setlocale(LC_ALL, "fr_FR.ISO-8859-1");
```

In this example, all categories of the international environment are set to the locale corresponding to the string "fr_FR.ISO-8859-1", or to the French language as spoken in France using the ISO/IEC 8859-1:1998 standard codeset.

If the string does not correspond to a valid locale, *setlocale()* shall return a NULL pointer and the international environment is not changed. Otherwise, *setlocale()* shall return the name of the locale just set.

42684
42685
42686
42687*setlocale(category, "C")*

The ISO C standard states that one locale must exist on all conforming implementations. The name of the locale is C and corresponds to a minimal international environment needed to support the C programming language.

42688
42689
42690*setlocale(category, "")*

This sets a specific category to an implementation-defined default. This corresponds to the value of the environment variables.

42691
42692**FUTURE DIRECTIONS**

None.

42693

SEE ALSO42694
42695
42696
42697
42698
42699
42700

exec, *isalnum()*, *isalpha()*, *isblank()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *iswalnum()*, *iswalpha()*, *iswblank()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*, *iswspace()*, *iswupper()*, *iswxdigit()*, *isxdigit()*, *localeconv()*, *mblen()*, *mbstowcs()*, *mbtowc()*, *nl_langinfo()*, *printf()*, *scanf()*, *setlocale()*, *strcoll()*, *strerror()*, *strfmon()*, *strsignal()*, *strtod()*, *strxfrm()*, *tolower()*, *toupper()*, *towlower()*, *towupper()*, *uselocale()*, *wscoll()*, *wctod()*, *wcstombs()*, *wcsxfrm()*, *wctomb()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<langinfo.h>**, **<locale.h>**

42701
42702**CHANGE HISTORY**

First released in Issue 3.

42703
42704**Issue 5**

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

42705
42706**Issue 6**

Extensions beyond the ISO C standard are marked.

42707
42708
42709

The normative text is updated to avoid use of the term “must” for application requirements.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/124 is applied, updating the DESCRIPTION to clarify the behavior of:

42710

*setlocale(LC_ALL, "");*42711
42712**Issue 7**

Functionality relating to the Threads option is moved to the Base.

setlogmask()

42713 **NAME**
42714 setlogmask — set the log priority mask

SYNOPSIS

42716 XSI #include <syslog.h>
42717 int setlogmask(int *maskpri*);

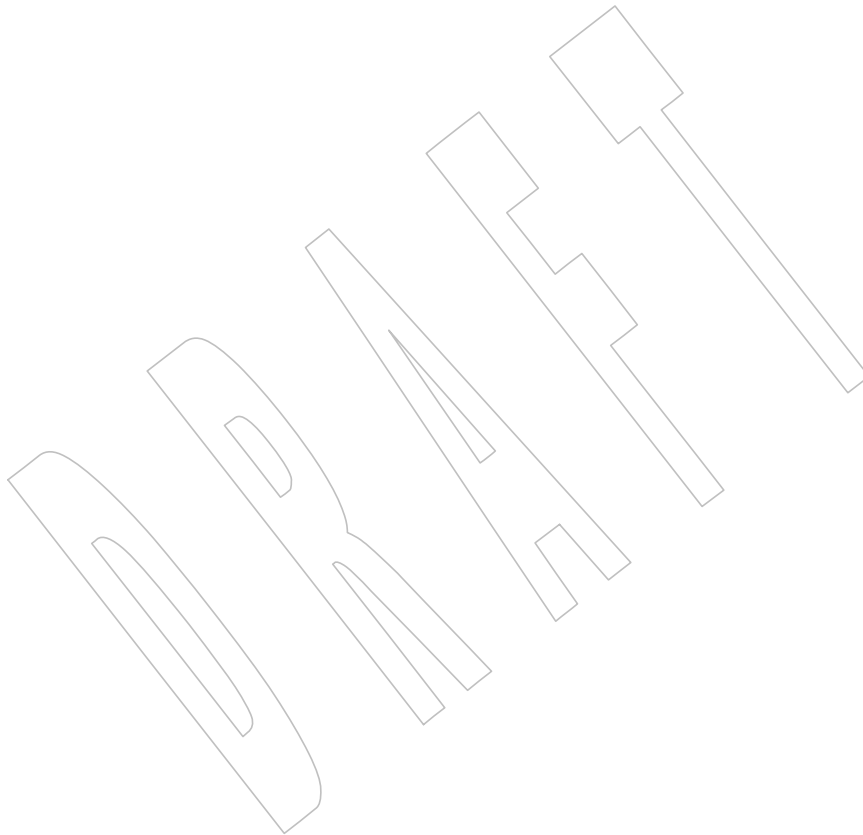
DESCRIPTION

42718 Refer to *closelog()*.
42719

42720 **NAME**
42721 setnetent — network database function

42722 **SYNOPSIS**
42723 #include <netdb.h>
42724 void setnetent(int stayopen);

42725 **DESCRIPTION**
42726 Refer to *endnetent()*.



42727 **NAME**

42728 setpgid — set process group ID for job control

42729 **SYNOPSIS**

42730 #include <unistd.h>

42731 int setpgid(pid_t pid, pid_t pgid);

42732 **DESCRIPTION**

42733 The *setpgid()* function shall either join an existing process group or create a new process group
 42734 within the session of the calling process. The process group ID of a session leader shall not
 42735 change. Upon successful completion, the process group ID of the process with a process ID that
 42736 matches *pid* shall be set to *pgid*. As a special case, if *pid* is 0, the process ID of the calling process
 42737 shall be used. Also, if *pgid* is 0, the process ID of the indicated process shall be used.

42738 **RETURN VALUE**

42739 Upon successful completion, *setpgid()* shall return 0; otherwise, -1 shall be returned and *errno*
 42740 shall be set to indicate the error.

42741 **ERRORS**42742 The *setpgid()* function shall fail if:

- 42743 [EACCES] The value of the *pid* argument matches the process ID of a child process of the
 42744 calling process and the child process has successfully executed one of the *exec*
 42745 functions.
- 42746 [EINVAL] The value of the *pgid* argument is less than 0, or is not a value supported by
 42747 the implementation.
- 42748 [EPERM] The process indicated by the *pid* argument is a session leader.
- 42749 [EPERM] The value of the *pid* argument matches the process ID of a child process of the
 42750 calling process and the child process is not in the same session as the calling
 42751 process.
- 42752 [EPERM] The value of the *pgid* argument is valid but does not match the process ID of
 42753 the process indicated by the *pid* argument and there is no process with a
 42754 process group ID that matches the value of the *pgid* argument in the same
 42755 session as the calling process.
- 42756 [ESRCH] The value of the *pid* argument does not match the process ID of the calling
 42757 process or of a child process of the calling process.

42758 **EXAMPLES**

42759 None.

42760 **APPLICATION USAGE**

42761 None.

42762 **RATIONALE**

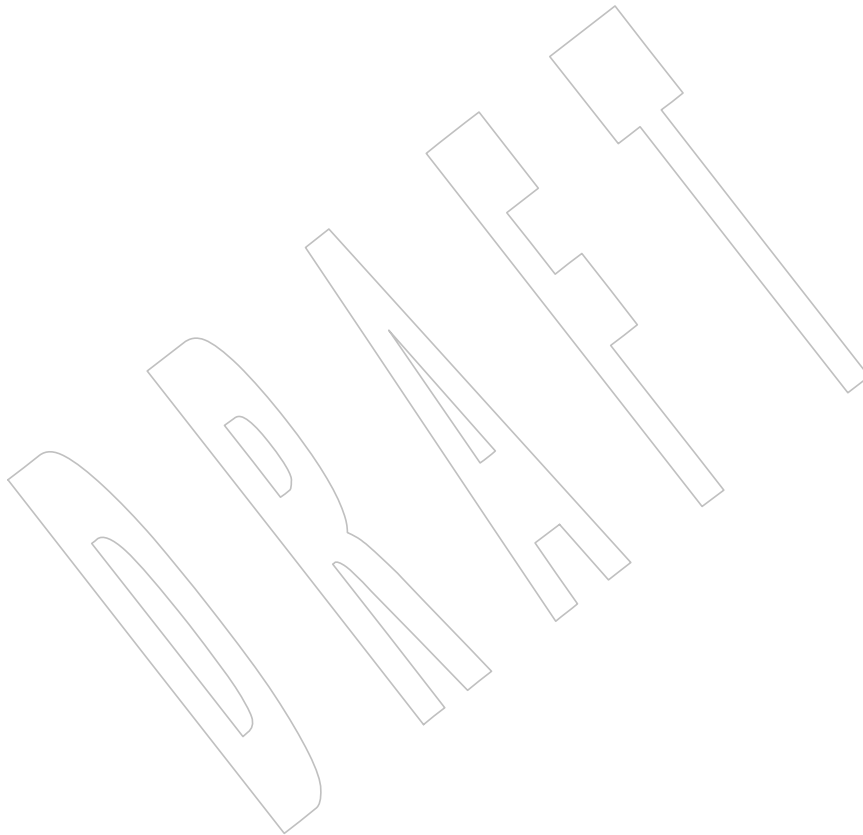
42763 The *setpgid()* function shall group processes together for the purpose of signaling, placement in
 42764 foreground or background, and other job control actions.

42765 The *setpgid()* function is similar to the *setpgrp()* function of 4.2 BSD, except that 4.2 BSD allowed
 42766 the specified new process group to assume any value. This presents certain security problems
 42767 and is more flexible than necessary to support job control.

42768 To provide tighter security, *setpgid()* only allows the calling process to join a process group
 42769 already in use inside its session or create a new process group whose process group ID was
 42770 equal to its process ID.

When a job control shell spawns a new job, the processes in the job must be placed into a new process group via *setpgid()*. There are two timing constraints involved in this action:

1. The new process must be placed in the new process group before the appropriate



42815 **NAME**42816 `setpgrp` — set the process group ID42817 **SYNOPSIS**42818 OB XSI

```
#include <unistd.h>
42819 pid_t setpgrp(void);
```

42820 **DESCRIPTION**42821 If the calling process is not already a session leader, *setpgrp()* sets the process group ID of the
42822 calling process to the process ID of the calling process. If *setpgrp()* creates a new session, then the
42823 new session has no controlling terminal.42824 The *setpgrp()* function has no effect when the calling process is a session leader.42825 **RETURN VALUE**42826 Upon completion, *setpgrp()* shall return the process group ID.42827 **ERRORS**

42828 No errors are defined.

42829 **EXAMPLES**

42830 None.

42831 **APPLICATION USAGE**42832 It is unspecified whether this function behaves as *setpgid(0,0)* or *setsid()* unless the process is
42833 already a session leader. Therefore, applications are encouraged to use *setpgid()* or *setsid()* as
42834 appropriate.42835 **RATIONALE**

42836 None.

42837 **FUTURE DIRECTIONS**42838 The *setpgrp()* function may be removed in a future version.42839 **SEE ALSO**42840 *exec*, *fork()*, *getpid()*, *getsid()*, *kill()*, *setpgid()*, *setsid()*, the Base Definitions volume of
42841 IEEE Std 1003.1-200x, **<unistd.h>**42842 **CHANGE HISTORY**

42843 First released in Issue 4, Version 2.

42844 **Issue 5**

42845 Moved from X/OPEN UNIX extension to BASE.

42846 **Issue 7**42847 The *setpgrp()* function is marked obsolescent.

42848 **NAME**
42849 setpriority — set the nice value

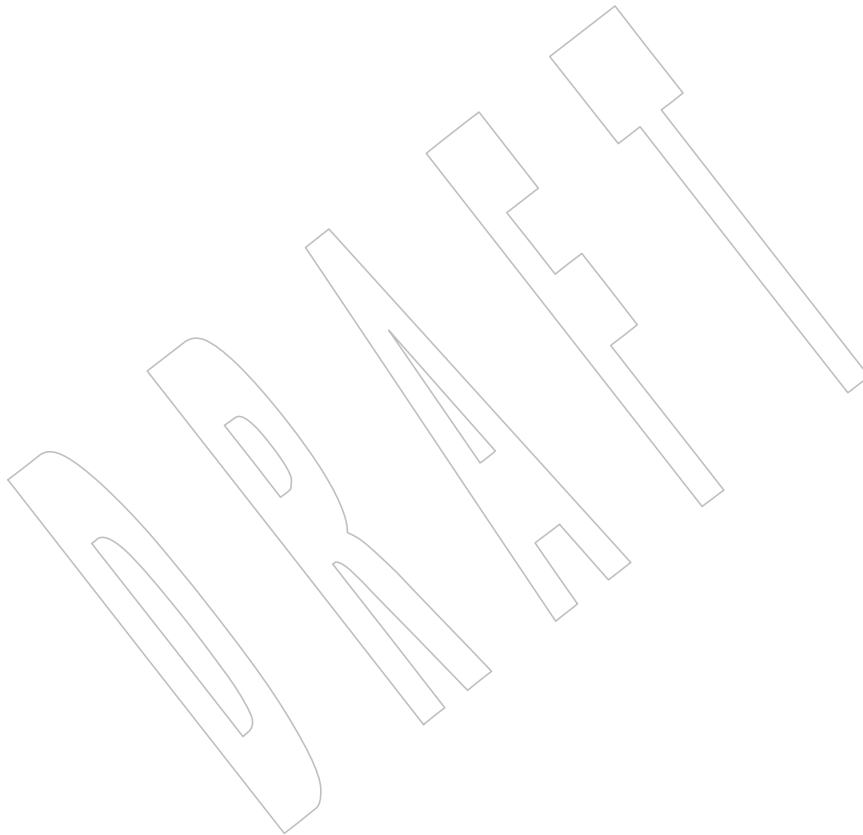
42850 **SYNOPSIS**
42851 XSI #include <sys/resource.h>
42852 int setpriority(int *which*, id_t *who*, int *nice*);

42853 **DESCRIPTION**
42854 Refer to *getpriority()*.

42855 **NAME**
42856 setprotoent — network protocol database functions

42857 **SYNOPSIS**
42858 #include <netdb.h>
42859 void setprotoent(int *stayopen*);

42860 **DESCRIPTION**
42861 Refer to *endprotoent()*.



42862 **NAME**
42863 setpwent — user database function

42864 **SYNOPSIS**

```
42865 XSI #include <pwd.h>  
42866 void setpwent(void);
```

42867 **DESCRIPTION**

42868 Refer to *endpwent()*.

42869 **NAME**

42870 setregid — set real and effective group IDs

42871 **SYNOPSIS**

```
42872 XSI #include <unistd.h>
42873 int setregid(gid_t rgid, gid_t egid);
```

42874 **DESCRIPTION**42875 The *setregid()* function shall set the real and effective group IDs of the calling process.42876 If *rgid* is -1 , the real group ID shall not be changed; if *egid* is -1 , the effective group ID shall not
42877 be changed.

42878 The real and effective group IDs may be set to different values in the same call.

42879 Only a process with appropriate privileges can set the real group ID and the effective group ID
42880 to any valid value.42881 A non-privileged process can set either the real group ID to the saved set-group-ID from one of
42882 the *exec* family of functions, or the effective group ID to the saved set-group-ID or the real group
42883 ID.

42884 Any supplementary group IDs of the calling process remain unchanged.

42885 **RETURN VALUE**42886 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
42887 indicate the error, and neither of the group IDs are changed.42888 **ERRORS**42889 The *setregid()* function shall fail if:42890 [EINVAL] The value of the *rgid* or *egid* argument is invalid or out-of-range.42891 [EPERM] The process does not have appropriate privileges and a change other than
42892 changing the real group ID to the saved set-group-ID, or changing the
42893 effective group ID to the real group ID or the saved set-group-ID, was
42894 requested.42895 **EXAMPLES**

42896 None.

42897 **APPLICATION USAGE**42898 If a set-group-ID process sets its effective group ID to its real group ID, it can still set its effective
42899 group ID back to the saved set-group-ID.42900 **RATIONALE**

42901 None.

42902 **FUTURE DIRECTIONS**

42903 None.

42904 **SEE ALSO**42905 *exec*, *getegid()*, *geteuid()*, *getgid()*, *getuid()*, *setegid()*, *seteuid()*, *setgid()*, *setreuid()*, *setuid()*, the
42906 Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

42907
42908
42909
42910
42911
42912

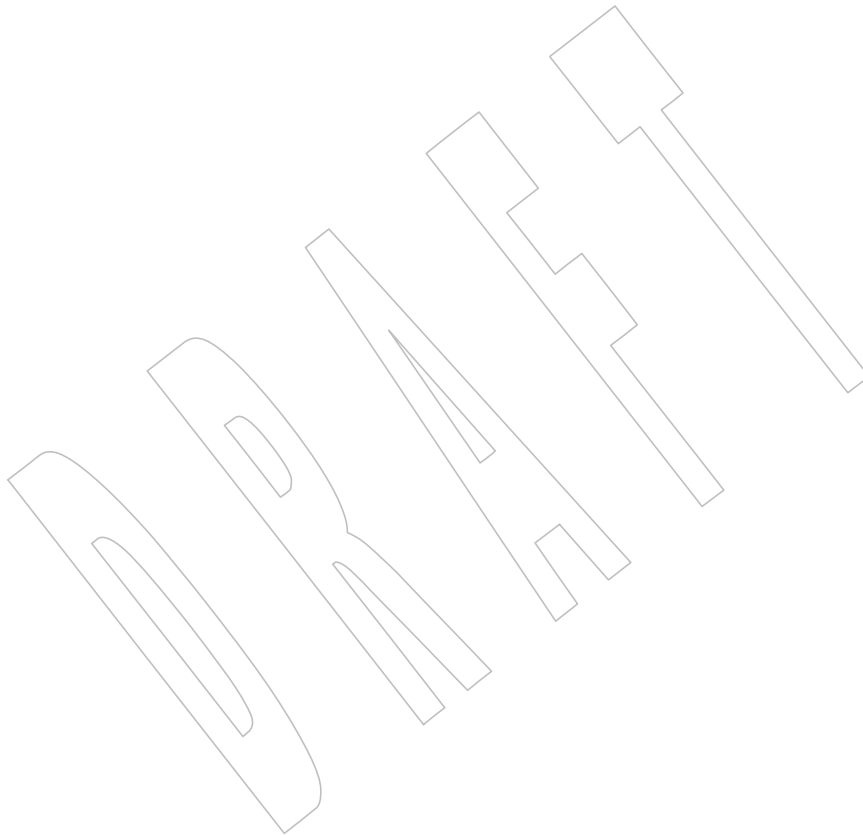
CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is updated to indicate that the saved set-group-ID can be set by any of the *exec* family of functions, not just *execve*().



42913 **NAME**

42914 setreuid — set real and effective user IDs

42915 **SYNOPSIS**

```
42916 XSI #include <unistd.h>
42917 int setreuid(uid_t ruid, uid_t euid);
```

42918 **DESCRIPTION**

42919 The *setreuid()* function shall set the real and effective user IDs of the current process to the
 42920 values specified by the *ruid* and *euid* arguments. If *ruid* or *euid* is -1 , the corresponding effective
 42921 or real user ID of the current process shall be left unchanged.

42922 A process with appropriate privileges can set either ID to any value. An unprivileged process
 42923 can only set the effective user ID if the *euid* argument is equal to either the real, effective, or
 42924 saved user ID of the process.

42925 It is unspecified whether a process without appropriate privileges is permitted to change the real
 42926 user ID to match the current real, effective, or saved set-user-ID of the process.

42927 **RETURN VALUE**

42928 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
 42929 indicate the error.

42930 **ERRORS**

42931 The *setreuid()* function shall fail if:

- | | | |
|-------|----------|--|
| 42932 | [EINVAL] | The value of the <i>ruid</i> or <i>euid</i> argument is invalid or out-of-range. |
| 42933 | [EPERM] | The current process does not have appropriate privileges, and either an attempt was made to change the effective user ID to a value other than the real user ID or the saved set-user-ID or an attempt was made to change the real user ID to a value not permitted by the implementation. |

42937 **EXAMPLES**42938 **Setting the Effective User ID to the Real User ID**

42939 The following example sets the effective user ID of the calling process to the real user ID, so that
 42940 files created later will be owned by the current user.

```
42941 #include <unistd.h>
42942 #include <sys/types.h>
42943 ...
42944 setreuid(getuid(), getuid());
42945 ...
```

42946 **APPLICATION USAGE**

42947 None.

42948 **RATIONALE**

42949 None.

42950 **FUTURE DIRECTIONS**

42951 None.

42952

SEE ALSO

42953

getegid(), *geteuid()*, *getgid()*, *getuid()*, *setegid()*, *seteuid()*, *setgid()*, *setregid()*, *setuid()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**unistd.h**>

42954

42955

CHANGE HISTORY

42956

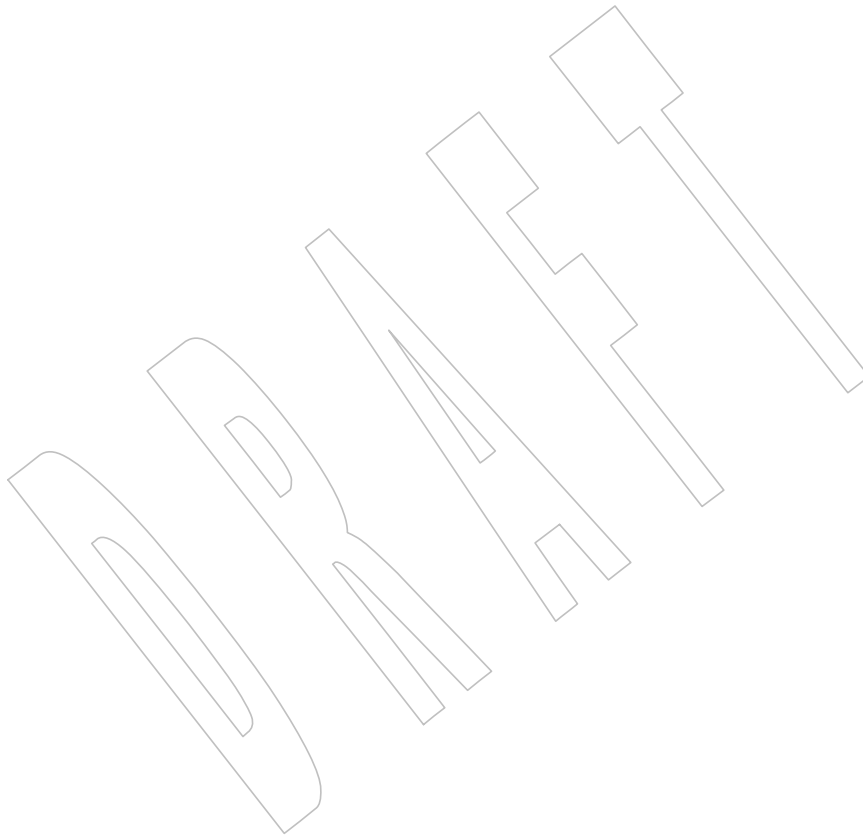
First released in Issue 4, Version 2.

42957

Issue 5

42958

Moved from X/OPEN UNIX extension to BASE.



setrlimit()

42959 **NAME**
42960 setrlimit — control maximum resource consumption

SYNOPSIS

42961 XSI `#include <sys/resource.h>`
42962 `int setrlimit(int resource, const struct rlimit *rlp);`

DESCRIPTION

42964 Refer to [getrlimit\(\)](#).
42965

42966 **NAME**
42967 setservent — network services database functions

42968 **SYNOPSIS**
42969 #include <netdb.h>
42970 void setservent(int stayopen);

42971 **DESCRIPTION**
42972 Refer to *endservent()*.



42973 **NAME**
 42974 `setsid` — create session and set process group ID

42975 **SYNOPSIS**
 42976 `#include <unistd.h>`
 42977 `pid_t setsid(void);`

42978 **DESCRIPTION**
 42979 The *setsid()* function shall create a new session, if the calling process is not a process group
 42980 leader. Upon return the calling process shall be the session leader of this new session, shall be
 42981 the process group leader of a new process group, and shall have no controlling terminal. The
 42982 process group ID of the calling process shall be set equal to the process ID of the calling process.
 42983 The calling process shall be the only process in the new process group and the only process in
 42984 the new session.

42985 **RETURN VALUE**
 42986 Upon successful completion, *setsid()* shall return the value of the new process group ID of the
 42987 calling process. Otherwise, it shall return **(pid_t)-1** and set *errno* to indicate the error.

42988 **ERRORS**
 42989 The *setsid()* function shall fail if:
 42990 [EPERM] The calling process is already a process group leader, or the process group ID
 42991 of a process other than the calling process matches the process ID of the
 42992 calling process.

42993 **EXAMPLES**
 42994 None.

42995 **APPLICATION USAGE**
 42996 None.

42997 **RATIONALE**
 42998 The *setsid()* function is similar to the *setpgid()* function of System V. System V, without job
 42999 control, groups processes into process groups and creates new process groups via *setpgid()*; only
 43000 one process group may be part of a login session.

43001 Job control allows multiple process groups within a login session. In order to limit job control
 43002 actions so that they can only affect processes in the same login session, this volume of
 43003 IEEE Std 1003.1-200x adds the concept of a session that is created via *setsid()*. The *setsid()*
 43004 function also creates the initial process group contained in the session. Additional process
 43005 groups can be created via the *setpgid()* function. A System V process group would correspond to
 43006 a POSIX System Interfaces session containing a single POSIX process group. Note that this
 43007 function requires that the calling process not be a process group leader. The usual way to ensure
 43008 this is true is to create a new process with *fork()* and have it call *setsid()*. The *fork()* function
 43009 guarantees that the process ID of the new process does not match any existing process group ID.

43010 **FUTURE DIRECTIONS**
 43011 None.

43012 **SEE ALSO**
 43013 *getsid()*, *setpgid()*, *setpgid()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<sys/types.h>`,
 43014 `<unistd.h>`

43015
43016
43017
43018
43019
43020
43021
43022
43023

CHANGE HISTORY

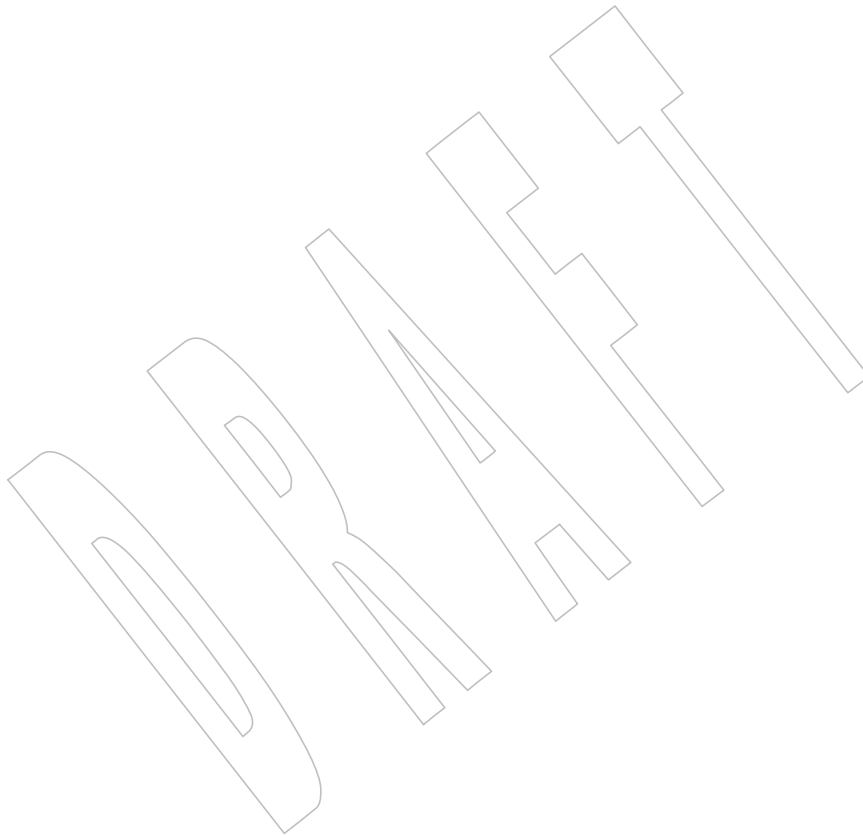
First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 6

In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.



43024 **NAME**
 43025 setsockopt — set the socket options

43026 **SYNOPSIS**
 43027 #include <sys/socket.h>
 43028 int setsockopt(int socket, int level, int option_name,
 43029 const void *option_value, socklen_t option_len);

43030 **DESCRIPTION**
 43031 The *setsockopt()* function shall set the option specified by the *option_name* argument, at the
 43032 protocol level specified by the *level* argument, to the value pointed to by the *option_value*
 43033 argument for the socket associated with the file descriptor specified by the *socket* argument.

43034 The *level* argument specifies the protocol level at which the option resides. To set options at the
 43035 socket level, specify the *level* argument as SOL_SOCKET. To set options at other levels, supply
 43036 the appropriate *level* identifier for the protocol controlling the option. For example, to indicate
 43037 that an option is interpreted by the TCP (Transport Control Protocol), set *level* to IPPROTO_TCP
 43038 as defined in the **<netinet/in.h>** header.

43039 The *option_name* argument specifies a single option to set. The *option_name* argument and any
 43040 specified options are passed uninterpreted to the appropriate protocol module for
 43041 interpretations. The **<sys/socket.h>** header defines the socket-level options. The options are as
 43042 follows:

43043 SO_DEBUG Turns on recording of debugging information. This option enables or
 43044 disables debugging in the underlying protocol modules. This option takes
 43045 an **int** value. This is a Boolean option.

43046 SO_BROADCAST Permits sending of broadcast messages, if this is supported by the
 43047 protocol. This option takes an **int** value. This is a Boolean option.

43048 SO_REUSEADDR Specifies that the rules used in validating addresses supplied to *bind()*
 43049 should allow reuse of local addresses, if this is supported by the protocol.
 43050 This option takes an **int** value. This is a Boolean option.

43051 SO_KEEPALIVE Keeps connections active by enabling the periodic transmission of
 43052 messages, if this is supported by the protocol. This option takes an **int**
 43053 value.

43054 If the connected socket fails to respond to these messages, the connection
 43055 is broken and threads writing to that socket are notified with a SIGPIPE
 43056 signal. This is a Boolean option.

43057 SO_LINGER Lingers on a *close()* if data is present. This option controls the action taken
 43058 when unsent messages queue on a socket and *close()* is performed. If
 43059 SO_LINGER is set, the system shall block the calling thread during *close()*
 43060 until it can transmit the data or until the time expires. If SO_LINGER is
 43061 not specified, and *close()* is issued, the system handles the call in a way
 43062 that allows the calling thread to continue as quickly as possible. This
 43063 option takes a **linger** structure, as defined in the **<sys/socket.h>** header, to
 43064 specify the state of the option and linger interval.

43065 SO_OOBINLINE Leaves received out-of-band data (data marked urgent) inline. This option
 43066 takes an **int** value. This is a Boolean option.

43067	SO_SNDBUF	Sets send buffer size. This option takes an int value.
43068	SO_RCVBUF	Sets receive buffer size. This option takes an int value.
43069	SO_DONTROUTE	Requests that outgoing messages bypass the standard routing facilities. The destination shall be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. The effect, if any, of this option depends on what protocol is in use. This option takes an int value. This is a Boolean option.
43070		
43071		
43072		
43073		
43074	SO_RCVLOWAT	Sets the minimum number of bytes to process for socket input operations. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. (They may return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that returned; for example, out-of-band data.) This option takes an int value. Note that not all implementations allow this option to be set.
43075		
43076		
43077		
43078		
43079		
43080		
43081		
43082	SO_RCVTIMEO	Sets the timeout value that specifies the maximum amount of time an input function waits until it completes. It accepts a timeval structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. If a receive operation has blocked for this much time without receiving additional data, it shall return with a partial count or <i>errno</i> set to [EAGAIN] or [EWOULDBLOCK] if no data is received. The default for this option is zero, which indicates that a receive operation shall not time out. This option takes a timeval structure. Note that not all implementations allow this option to be set.
43083		
43084		
43085		
43086		
43087		
43088		
43089		
43090		
43091		
43092	SO_SNDLOWAT	Sets the minimum number of bytes to process for socket output operations. Non-blocking output operations shall process no data if flow control does not allow the smaller of the send low water mark value or the entire request to be processed. This option takes an int value. Note that not all implementations allow this option to be set.
43093		
43094		
43095		
43096		
43097	SO_SNDTIMEO	Sets the timeout value specifying the amount of time that an output function blocks because flow control prevents data from being sent. If a send operation has blocked for this time, it shall return with a partial count or with <i>errno</i> set to [EAGAIN] or [EWOULDBLOCK] if no data is sent. The default for this option is zero, which indicates that a send operation shall not time out. This option stores a timeval structure. Note that not all implementations allow this option to be set.
43098		
43099		
43100		
43101		
43102		
43103		
43104		
43105		
43106		
43107		
43108		
43109		
43110		
43111		
43112		

For Boolean options, 0 indicates that the option is disabled and 1 indicates that the option is enabled.

Options at other protocol levels vary in format and name.

RETURN VALUE

Upon successful completion, *setsockopt()* shall return 0. Otherwise, -1 shall be returned and *errno* set to indicate the error.

ERRORS

The *setsockopt()* function shall fail if:

[EBADF] The *socket* argument is not a valid file descriptor.

setsockopt()

43113	[EDOM]	The send and receive timeout values are too big to fit into the timeout fields in the socket structure.
43114		
43115	[EINVAL]	The specified option is invalid at the specified socket level or the socket has been shut down.
43116		
43117	[EISCONN]	The socket is already connected, and a specified option cannot be set while the socket is connected.
43118		
43119	[ENOPROTOOPT]	
43120		The option is not supported by the protocol.
43121	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
43122		The <i>setsockopt()</i> function may fail if:
43123	[ENOMEM]	There was insufficient memory available for the operation to complete.
43124	[ENOBUFS]	Insufficient resources are available in the system to complete the call.

EXAMPLES

None.

APPLICATION USAGE

The *setsockopt()* function provides an application program with the means to control socket behavior. An application program can use *setsockopt()* to allocate buffer space, control timeouts, or permit socket data broadcasts. The `<sys/socket.h>` header defines the socket-level options available to *setsockopt()*.

Options may exist at multiple protocol levels. The `SO_` options are always present at the uppermost socket level.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.10 (on page 60), *bind()*, *endprotoent()*, *getsockopt()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<netinet/in.h>`, `<sys/socket.h>`

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/125 is applied, updating the `SO_LINGER` option in the DESCRIPTION to refer to the calling thread rather than the process.

43145 **NAME**
43146 setstate — switch pseudo-random number generator state arrays

43147 **SYNOPSIS**

43148 XSI #include <stdlib.h>
43149 char *setstate(const char *state);

43150 **DESCRIPTION**

43151 Refer to *initstate()*.

43152 **NAME**

43153 setuid — set user ID

43154 **SYNOPSIS**43155 #include <unistd.h>
43156 int setuid(uid_t uid);43157 **DESCRIPTION**43158 If the process has appropriate privileges, *setuid()* shall set the real user ID, effective user ID, and
43159 the saved set-user-ID of the calling process to *uid*.43160 If the process does not have appropriate privileges, but *uid* is equal to the real user ID or the
43161 saved set-user-ID, *setuid()* shall set the effective user ID to *uid*; the real user ID and saved set-
43162 user-ID shall remain unchanged.43163 The *setuid()* function shall not affect the supplementary group list in any way.43164 **RETURN VALUE**43165 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
43166 indicate the error.43167 **ERRORS**43168 The *setuid()* function shall fail, return -1, and set *errno* to the corresponding value if one or more
43169 of the following are true:43170 [EINVAL] The value of the *uid* argument is invalid and not supported by the
43171 implementation.43172 [EPERM] The process does not have appropriate privileges and *uid* does not match the
43173 real user ID or the saved set-user-ID.43174 **EXAMPLES**

43175 None.

43176 **APPLICATION USAGE**

43177 None.

43178 **RATIONALE**43179 The various behaviors of the *setuid()* and *setgid()* functions when called by non-privileged
43180 processes reflect the behavior of different historical implementations. For portability, it is
43181 recommended that new non-privileged applications use the *seteuid()* and *setegid()* functions
43182 instead.43183 The saved set-user-ID capability allows a program to regain the effective user ID established at
43184 the last *exec* call. Similarly, the saved set-group-ID capability allows a program to regain the
43185 effective group ID established at the last *exec* call. These capabilities are derived from System V.
43186 Without them, a program might have to run as superuser in order to perform the same
43187 functions, because superuser can write on the user's files. This is a problem because such a
43188 program can write on any user's files, and so must be carefully written to emulate the
43189 permissions of the calling process properly. In System V, these capabilities have traditionally
43190 been implemented only via the *setuid()* and *setgid()* functions for non-privileged processes. The
43191 fact that the behavior of those functions was different for privileged processes made them
43192 difficult to use. The POSIX.1-1990 standard defined the *setuid()* function to behave differently
43193 for privileged and unprivileged users. When the caller had the appropriate privilege, the
43194 function set the real user ID, effective user ID, and saved set-user ID of the calling process on
43195 implementations that supported it. When the caller did not have the appropriate privilege, the
43196 function set only the effective user ID, subject to permission checks. The former use is generally

43197 needed for utilities like *login* and *su*, which are not conforming applications and thus outside the
 43198 scope of IEEE Std 1003.1-200x. These utilities wish to change the user ID irrevocably to a new
 43199 value, generally that of an unprivileged user. The latter use is needed for conforming
 43200 applications that are installed with the set-user-ID bit and need to perform operations using the
 43201 real user ID.

43202 IEEE Std 1003.1-200x augments the latter functionality with a mandatory feature named
 43203 `_POSIX_SAVED_IDS`. This feature permits a set-user-ID application to switch its effective user
 43204 ID back and forth between the values of its *exec*-time real user ID and effective user ID.
 43205 Unfortunately, the POSIX.1-1990 standard did not permit a conforming application using this
 43206 feature to work properly when it happened to be executed with the (implementation-defined)
 43207 appropriate privilege. Furthermore, the application did not even have a means to tell whether it
 43208 had this privilege. Since the saved set-user-ID feature is quite desirable for applications, as
 43209 evidenced by the fact that NIST required it in FIPS 151-2, it has been mandated by
 43210 IEEE Std 1003.1-200x. However, there are implementors who have been reluctant to support it
 43211 given the limitation described above.

43212 The 4.3BSD system handles the problem by supporting separate functions: *setuid()* (which
 43213 always sets both the real and effective user IDs, like *setuid()* in IEEE Std 1003.1-200x for
 43214 privileged users), and *seteuid()* (which always sets just the effective user ID, like *setuid()* in
 43215 IEEE Std 1003.1-200x for non-privileged users). This separation of functionality into distinct
 43216 functions seems desirable. 4.3BSD does not support the saved set-user-ID feature. It supports
 43217 similar functionality of switching the effective user ID back and forth via *setreuid()*, which
 43218 permits reversing the real and effective user IDs. This model seems less desirable than the saved
 43219 set-user-ID because the real user ID changes as a side effect. The current 4.4BSD includes saved
 43220 effective IDs and uses them for *seteuid()* and *setegid()* as described above. The *setreuid()* and
 43221 *setregid()* functions will be deprecated or removed.

43222 The solution here is:

- 43223 • Require that all implementations support the functionality of the saved set-user-ID, which
 43224 is set by the *exec* functions and by privileged calls to *setuid()*.
- 43225 • Add the *seteuid()* and *setegid()* functions as portable alternatives to *setuid()* and *setgid()* for
 43226 non-privileged and privileged processes.

43227 Historical systems have provided two mechanisms for a set-user-ID process to change its
 43228 effective user ID to be the same as its real user ID in such a way that it could return to the
 43229 original effective user ID: the use of the *setuid()* function in the presence of a saved set-user-ID,
 43230 or the use of the BSD *setreuid()* function, which was able to swap the real and effective user IDs.
 43231 The changes included in IEEE Std 1003.1-200x provide a new mechanism using *seteuid()* in
 43232 conjunction with a saved set-user-ID. Thus, all implementations with the new *seteuid()*
 43233 mechanism will have a saved set-user-ID for each process, and most of the behavior controlled
 43234 by `_POSIX_SAVED_IDS` has been changed to agree with the case where the option was defined.
 43235 The *kill()* function is an exception. Implementors of the new *seteuid()* mechanism will generally
 43236 be required to maintain compatibility with the older mechanisms previously supported by their
 43237 systems. However, compatibility with this use of *setreuid()* and with the `_POSIX_SAVED_IDS`
 43238 behavior of *kill()* is unfortunately complicated. If an implementation with a saved set-user-ID
 43239 allows a process to use *setreuid()* to swap its real and effective user IDs, but were to leave the
 43240 saved set-user-ID unmodified, the process would then have an effective user ID equal to the
 43241 original real user ID, and both real and saved set-user-ID would be equal to the original effective
 43242 user ID. In that state, the real user would be unable to kill the process, even though the effective
 43243 user ID of the process matches that of the real user, if the *kill()* behavior of `_POSIX_SAVED_IDS`
 43244 was used. This is obviously not acceptable. The alternative choice, which is used in at least one
 43245 implementation, is to change the saved set-user-ID to the effective user ID during most calls to
 43246 *setreuid()*. The standard developers considered that alternative to be less correct than the
 43247 retention of the old behavior of *kill()* in such systems. Current conforming applications shall

43248 accommodate either behavior from *kill()*, and there appears to be no strong reason for *kill()* to
 43249 check the saved set-user-ID rather than the effective user ID.

FUTURE DIRECTIONS

43250 None.

SEE ALSO

43253 *exec*, *getegid()*, *geteuid()*, *getgid()*, *getuid()*, *setegid()*, *seteuid()*, *setgid()*, *setregid()*, *setreuid()*, the
 43254 Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

43255 First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

43257 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

43259 The following new requirements on POSIX implementations derive from alignment with the
 43260 Single UNIX Specification:

- 43261 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
 43262 required for conforming implementations of previous POSIX specifications, it was not
 43263 required for UNIX applications.
- 43264 • The functionality associated with `_POSIX_SAVED_IDS` is now mandatory. This is a FIPS
 43265 requirement.

43266 The following changes were made to align with the IEEE P1003.1a draft standard:

- 43267 • The effects of *setuid()* in processes without appropriate privileges are changed.
- 43268 • A requirement that the supplementary group list is not affected is added.

43269 **NAME**
43270 setutxent — reset the user accounting database to the first entry

43271 **SYNOPSIS**

43272 XSI #include <utmpx.h>
43273 void setutxent(void);

43274 **DESCRIPTION**

43275 Refer to *endutxent()*.

NAME

setvbuf — assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>
```

```
int setvbuf(FILE *restrict stream, char *restrict buf, int type,  
            size_t size);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is u

43316
43317
43318
43319
43320
43321
43322
43323
43324
43325**FUTURE DIRECTIONS**

None.

SEE ALSO

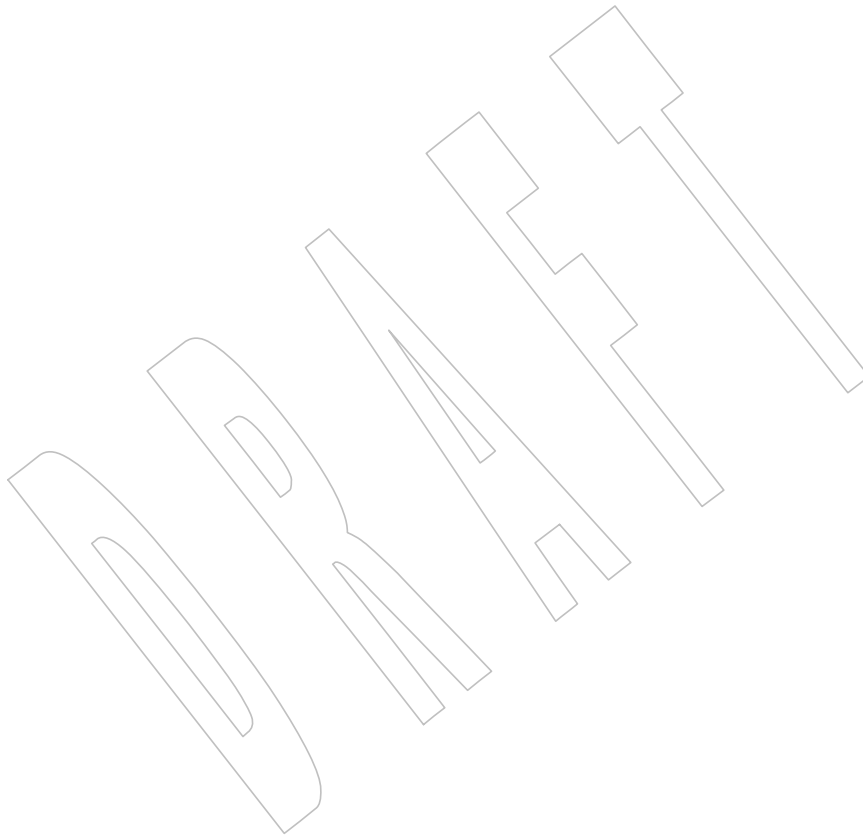
Section 2.5 (on page 34), *fopen()*, *setbuf()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdio.h>`

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are marked.

The *setvbuf()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

43326 **NAME**
 43327 shm_open — open a shared memory object (**REALTIME**)

43328 **SYNOPSIS**

```
43329 SHM #include <sys/mman.h>
43330 int shm_open(const char *name, int oflag, mode_t mode);
```

43331 **DESCRIPTION**

43332 The *shm_open()* function shall establish a connection between a shared memory object and a file
 43333 descriptor. It shall create an open file description that refers to the shared memory object and a
 43334 file descriptor that refers to that open file description. The file descriptor is used by other
 43335 functions to refer to that shared memory object. The *name* argument points to a string naming a
 43336 shared memory object. It is unspecified whether the name appears in the file system and is
 43337 visible to other functions that take pathnames as arguments. The *name* argument conforms to the
 43338 construction rules for a pathname, except that the interpretation of slash characters other than
 43339 the leading slash character in *name* is implementation-defined, and that the length limits for the
 43340 *name* argument are implementation-defined and need not be the same as the pathname limits
 43341 {PATH_MAX} and {NAME_MAX}. If *name* begins with the slash character, then processes
 43342 calling *shm_open()* with the same value of *name* refer to the same shared memory object, as long
 43343 as that name has not been removed. If *name* does not begin with the slash character, the effect is
 43344 implementation-defined.

43345 If successful, *shm_open()* shall return a file descriptor for the shared memory object that is the
 43346 lowest numbered file descriptor not currently open for that process. The open file description is
 43347 new, and therefore the file descriptor does not share it with any other processes. It is unspecified
 43348 whether the file offset is set. The FD_CLOEXEC file descriptor flag associated with the new file
 43349 descriptor is set.

43350 The file status flags and file access modes of the open file description are according to the value
 43351 of *oflag*. The *oflag* argument is the bitwise-inclusive OR of the following flags defined in the
 43352 **<fcntl.h>** header. Applications specify exactly one of the first two values (access modes) below
 43353 in the value of *oflag*:

43354 O_RDONLY Open for read access only.

43355 O_RDWR Open for read or write access.

43356 Any combination of the remaining flags may be specified in the value of *oflag*:

43357 O_CREAT If the shared memory object exists, this flag has no effect, except as noted
 43358 under O_EXCL below. Otherwise, the shared memory object is created; the user ID of the shared memory object shall be set to the effective user ID of the
 43359 process; the group ID of the shared memory object is set to a system default group ID or to the effective group ID of the process. The permission bits of the
 43360 shared memory object shall be set to the value of the *mode* argument except
 43361 those set in the file mode creation mask of the process. When bits in *mode*
 43362 other than the file permission bits are set, the effect is unspecified. The *mode*
 43363 argument does not affect whether the shared memory object is opened for
 43364 reading, for writing, or for both. The shared memory object has a size of zero.

43367 O_EXCL If O_EXCL and O_CREAT are set, *shm_open()* fails if the shared memory object
 43368 exists. The check for the existence of the shared memory object and the
 43369 creation of the object if it does not exist is atomic with respect to other
 43370 processes executing *shm_open()* naming the same shared memory object with
 43371 O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the

43372 result is undefined.

43373 O_TRUNC If the shared memory object exists, and it is successfully opened O_RDWR, the

43374 object shall be truncated to zero length and the mode and owner shall be

43375 unchanged by this function call. The result of using O_TRUNC with

43376 O_RDONLY is undefined.

43377 When a shared memory object is created, the state of the shared memory object, including all

43378 data associated with the shared memory object, persists until the shared memory object is

43379 unlinked and all other references are gone. It is unspecified whether the name and shared

43380 memory object state remain valid after a system reboot.

43381 RETURN VALUE

43382 Upon successful completion, the *shm_open()* function shall return a non-negative integer

43383 representing the lowest numbered unused file descriptor. Otherwise, it shall return `-1` and set

43384 *errno* to indicate the error.

43385 ERRORS

43386 The *shm_open()* function shall fail if:

43387 [EACCES] The shared memory object exists and the permissions specified by *oflag* are

43388 denied, or the shared memory object does not exist and permission to create

43389 the shared memory object is denied, or O_TRUNC is specified and write

43390 permission is denied.

43391 [EEXIST] O_CREAT and O_EXCL are set and the named shared memory object already

43392 exists.

43393 [EINTR] The *shm_open()* operation was interrupted by a signal.

43394 [EINVAL] The *shm_open()* operation is not supported for the given name.

43395 [EMFILE] All file descriptors available to the process are currently open.

43396 [ENFILE] Too many shared memory objects are currently open in the system.

43397 [ENOENT] O_CREAT is not set and the named shared memory object does not exist.

43398 [ENOSPC] There is insufficient space for the creation of the new shared memory object.

43399 The *shm_open()* function may fail if:

43400 [ENAMETOOLONG]

43401 The length of the *name* argument exceeds `{_POSIX_PATH_MAX}` on systems

43402 XSI that do not support the XSI option or exceeds `{_XOPEN_PATH_MAX}` on XSI

43403 systems, or has a pathname component that is longer than

43404 XSI `{_POSIX_NAME_MAX}` on systems that do not support the XSI option or

43405 longer than `{_XOPEN_NAME_MAX}` on XSI systems.

43406 EXAMPLES

43407 Creating and Mapping a Shared Memory Object

43408 The following code segment demonstrates the use of *shm_open()* to create a shared memory

43409 object which is then sized using *ftruncate()* before being mapped into the process address space

43410 using *mmap()*:

```
43411 #include <unistd.h>
43412 #include <sys/mman.h>
43413 ...
43414 #define MAX_LEN 10000
43415 struct region {          /* Defines "structure" of shared memory */
```

```

43416         int len;
43417         char buf[MAX_LEN];
43418     };
43419     struct region *rptr;
43420     int fd;
43421
43422     /* Create shared memory object and set its size */
43423     fd = shm_open("/myregion", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
43424     if (fd == -1)
43425         /* Handle error */;
43426
43427     if (ftruncate(fd, sizeof(struct region)) == -1)
43428         /* Handle error */;
43429
43430     /* Map shared memory object */
43431     rptr = mmap(NULL, sizeof(struct region),
43432                PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
43433     if (rptr == MAP_FAILED)
43434         /* Handle error */;
43435
43436     /* Now we can refer to mapped region using fields of rptr;
43437        for example, rptr->len */
43438     ...

```

APPLICATION USAGE

None.

RATIONALE

When the Memory Mapped Files option is supported, the normal *open()* call is used to obtain a descriptor to a file to be mapped according to existing practice with *mmap()*. When the Shared Memory Objects option is supported, the *shm_open()* function shall obtain a descriptor to the shared memory object to be mapped.

There is ample precedent for having a file descriptor represent several types of objects. In the POSIX.1-1990 standard, a file descriptor can represent a file, a pipe, a FIFO, a tty, or a directory. Many implementations simply have an operations vector, which is indexed by the file descriptor type and does very different operations. Note that in some cases the file descriptor passed to generic operations on file descriptors is returned by *open()* or *creat()* and in some cases returned by alternate functions, such as *pipe()*. The latter technique is used by *shm_open()*.

Note that such shared memory objects can actually be implemented as mapped files. In both cases, the size can be set after the open using *ftruncate()*. The *shm_open()* function itself does not create a shared object of a specified size because this would duplicate an extant function that set the size of an object referenced by a file descriptor.

On implementations where memory objects are implemented using the existing file system, the *shm_open()* function may be implemented using a macro that invokes *open()*, and the *shm_unlink()* function may be implemented using a macro that invokes *unlink()*.

For implementations without a permanent file system, the definition of the name of the memory objects is allowed not to survive a system reboot. Note that this allows systems with a permanent file system to implement memory objects as data structures internal to the implementation as well.

On implementations that choose to implement memory objects using memory directly, a *shm_open()* followed by an *ftruncate()* and *close()* can be used to preallocate a shared memory area and to set the size of that preallocation. This may be necessary for systems without virtual memory hardware support in order to ensure that the memory is contiguous.

43463 The set of valid open flags to *shm_open()* was restricted to *O_RDONLY*, *O_RDWR*, *O_CREAT*,
 43464 and *O_TRUNC* because these could be easily implemented on most memory mapping systems.
 43465 This volume of IEEE Std 1003.1-200x is silent on the results if the implementation cannot supply
 43466 the requested file access because of implementation-defined reasons, including hardware ones.

43467 The error conditions [EACCES] and [ENOTSUP] are provided to inform the application that the
 43468 implementation cannot complete a request.

43469 [EACCES] indicates for implementation-defined reasons, probably hardware-related, that the
 43470 implementation cannot comply with a requested mode because it conflicts with another
 43471 requested mode. An example might be that an application desires to open a memory object two
 43472 times, mapping different areas with different access modes. If the implementation cannot map a
 43473 single area into a process space in two places, which would be required if different access modes
 43474 were required for the two areas, then the implementation may inform the application at the time
 43475 of the second open.

43476 [ENOTSUP] indicates for implementation-defined reasons, probably hardware-related, that the
 43477 implementation cannot comply with a requested mode at all. An example would be that the
 43478 hardware of the implementation cannot support write-only shared memory areas.

43479 On all implementations, it may be desirable to restrict the location of the memory objects to
 43480 specific file systems for performance (such as a RAM disk) or implementation-defined reasons
 43481 (shared memory supported directly only on certain file systems). The *shm_open()* function may
 43482 be used to enforce these restrictions. There are a number of methods available to the application
 43483 to determine an appropriate name of the file or the location of an appropriate directory. One way
 43484 is from the environment via *getenv()*. Another would be from a configuration file.

43485 This volume of IEEE Std 1003.1-200x specifies that memory objects have initial contents of zero
 43486 when created. This is consistent with current behavior for both files and newly allocated
 43487 memory. For those implementations that use physical memory, it would be possible that such
 43488 implementations could simply use available memory and give it to the process uninitialized.
 43489 This, however, is not consistent with standard behavior for the uninitialized data area, the stack,
 43490 and of course, files. Finally, it is highly desirable to set the allocated memory to zero for security
 43491 reasons. Thus, initializing memory objects to zero is required.

43492 FUTURE DIRECTIONS

43493 None.

43494 SEE ALSO

43495 *close()*, *dup()*, *exec*, *fcntl()*, *mmap()*, *shmat()*, *shmctl()*, *shmdt()*, *shm_unlink()*, *umask()*, the Base
 43496 Definitions volume of IEEE Std 1003.1-200x, <fcntl.h>, <sys/mman.h>

43497 CHANGE HISTORY

43498 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

43499 Issue 6

43500 The *shm_open()* function is marked as part of the Shared Memory Objects option.

43501 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 43502 implementation does not support the Shared Memory Objects option.

43503 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/126 is applied, adding the example to the
 43504 EXAMPLES section.

43505 Issue 7

43506 Austin Group Interpretation 1003.1-2001 #077 is applied, clarifying the *name* argument and
 43507 changing [ENAMETOOLONG] from a “shall fail” to a “may fail” error.

43508 **NAME**
 43509 shm_unlink — remove a shared memory object (**REALTIME**)

43510 **SYNOPSIS**

```
43511 SHM #include <sys/mman.h>
43512 int shm_unlink(const char *name);
```

43513 **DESCRIPTION**

43514 The *shm_unlink()* function shall remove the name of the shared memory object named by the
 43515 string pointed to by *name*.

43516 If one or more references to the shared memory object exist when the object is unlinked, the
 43517 name shall be removed before *shm_unlink()* returns, but the removal of the memory object
 43518 contents shall be postponed until all open and map references to the shared memory object have
 43519 been removed.

43520 Even if the object continues to exist after the last *shm_unlink()*, reuse of the name shall
 43521 subsequently cause *shm_open()* to behave as if no shared memory object of this name exists (that
 43522 is, *shm_open()* will fail if *O_CREAT* is not set, or will create a new shared memory object if
 43523 *O_CREAT* is set).

43524 **RETURN VALUE**

43525 Upon successful completion, a value of zero shall be returned. Otherwise, a value of -1 shall be
 43526 returned and *errno* set to indicate the error. If -1 is returned, the named shared memory object
 43527 shall not be changed by this function call.

43528 **ERRORS**

43529 The *shm_unlink()* function shall fail if:

- 43530 [EACCES] Permission is denied to unlink the named shared memory object.
- 43531 [ENOENT] The named shared memory object does not exist.

43532 The *shm_unlink()* function may fail if:

43533 [ENAMETOOLONG]

43534 The length of the *name* argument exceeds $\{_POSIX_PATH_MAX\}$ on systems
 43535 XSI that do not support the XSI option or exceeds $\{_XOPEN_PATH_MAX\}$ on XSI
 43536 systems, or has a pathname component that is longer than
 43537 XSI $\{_POSIX_NAME_MAX\}$ on systems that do not support the XSI option or
 43538 longer than $\{_XOPEN_NAME_MAX\}$ on XSI systems. A call to *shm_unlink()*
 43539 with a *name* argument that contains the same shared memory object name as
 43540 was previously used in a successful *shm_open()* call shall not give an
 43541 [ENAMETOOLONG] error.

43542 **EXAMPLES**

43543 None.

43544 **APPLICATION USAGE**

43545 Names of memory objects that were allocated with *open()* are deleted with *unlink()* in the usual
 43546 fashion. Names of memory objects that were allocated with *shm_open()* are deleted with
 43547 *shm_unlink()*. Note that the actual memory object is not destroyed until the last close and
 43548 unmap on it have occurred if it was already in use.

43549
43550
43551
43552
43553
43554
43555
43556
43557
43558
43559
43560
43561
43562
43563
43564
43565
43566

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

close(), *mmap()*, *munmap()*, *shmat()*, *shmctl()*, *shmdt()*, *shm_open()*, the Base Definitions volume of IEEE Std 1003.1-200x, <sys/mman.h>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

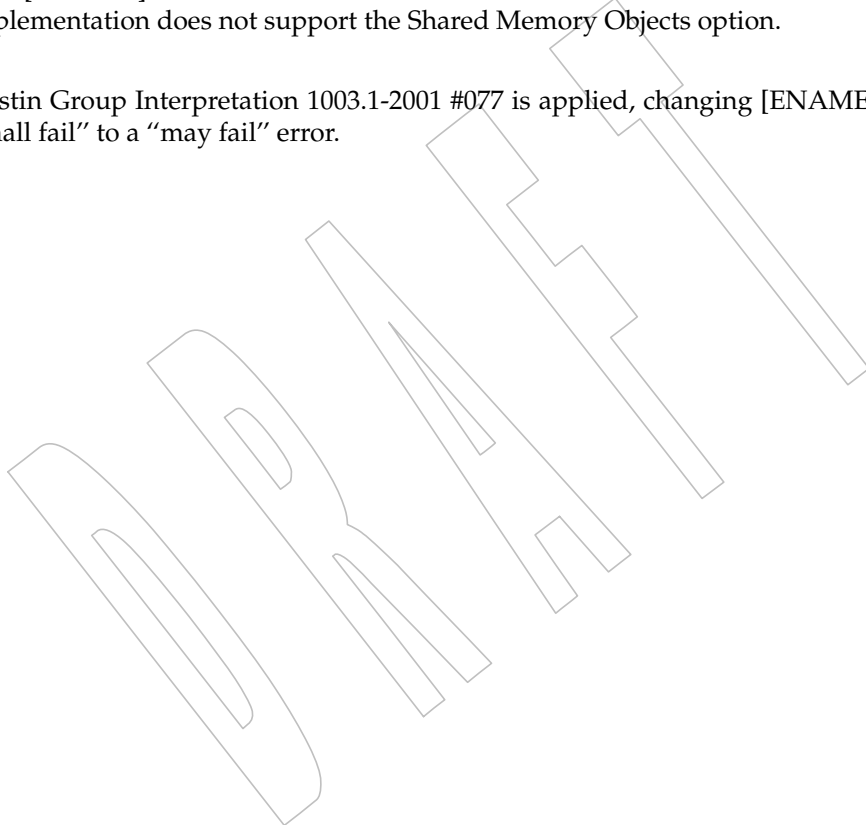
The *shm_unlink()* function is marked as part of the Shared Memory Objects option.

In the DESCRIPTION, text is added to clarify that reusing the same name after a *shm_unlink()* will not attach to the old shared memory object.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Shared Memory Objects option.

Issue 7

Austin Group Interpretation 1003.1-2001 #077 is applied, changing [ENAMETOOLONG] from a “shall fail” to a “may fail” error.



43567 **NAME**
 43568 shmat — XSI shared memory attach operation

43569 **SYNOPSIS**

```
43570 XSI #include <sys/shm.h>
43571 void *shmat(int shmid, const void *shmaddr, int shmflg);
```

43572 **DESCRIPTION**

43573 The *shmat()* function operates on XSI shared memory (see the Base Definitions volume of
 43574 IEEE Std 1003.1-200x, Section 3.340, Shared Memory Object). It is unspecified whether this
 43575 function interoperates with the realtime interprocess communication facilities defined in [Section](#)
 43576 [2.8](#) (on page 40).

43577 The *shmat()* function attaches the shared memory segment associated with the shared memory
 43578 identifier specified by *shmid* to the address space of the calling process. The segment is attached
 43579 at the address specified by one of the following criteria:

- 43580 • If *shmaddr* is a null pointer, the segment is attached at the first available address as selected
 43581 by the system.
- 43582 • If *shmaddr* is not a null pointer and (*shmflg* &SHM_RND) is non-zero, the segment is
 43583 attached at the address given by (*shmaddr* - ((*uintptr_t*)*shmaddr* %SHMLBA)). The character
 43584 ' % ' is the C-language remainder operator.
- 43585 • If *shmaddr* is not a null pointer and (*shmflg* &SHM_RND) is 0, the segment is attached at
 43586 the address given by *shmaddr*.
- 43587 • The segment is attached for reading if (*shmflg* &SHM_RDONLY) is non-zero and the
 43588 calling process has read permission; otherwise, if it is 0 and the calling process has read
 43589 and write permission, the segment is attached for reading and writing.

43590 **RETURN VALUE**

43591 Upon successful completion, *shmat()* shall increment the value of *shm_nattch* in the data
 43592 structure associated with the shared memory ID of the attached shared memory segment and
 43593 return the segment's start address.

43594 Otherwise, the shared memory segment shall not be attached, *shmat()* shall return -1, and *errno*
 43595 shall be set to indicate the error.

43596 **ERRORS**

43597 The *shmat()* function shall fail if:

- | | | |
|-------|----------|--|
| 43598 | [EACCES] | Operation permission is denied to the calling process; see Section 2.7 (on page 39). |
| 43600 | [EINVAL] | The value of <i>shmid</i> is not a valid shared memory identifier, the <i>shmaddr</i> is not a null pointer, and the value of (<i>shmaddr</i> - ((<i>uintptr_t</i>) <i>shmaddr</i> %SHMLBA)) is an illegal address for attaching shared memory; or the <i>shmaddr</i> is not a null pointer, (<i>shmflg</i> &SHM_RND) is 0, and the value of <i>shmaddr</i> is an illegal address for attaching shared memory. |
| 43605 | [EMFILE] | The number of shared memory segments attached to the calling process would exceed the system-imposed limit. |
| 43607 | [ENOMEM] | The available data space is not large enough to accommodate the shared memory segment. |

43609

EXAMPLES

43610

None.

43611

APPLICATION USAGE

43612

The POSIX Realtime Extension defines alternative interfaces for interprocess communication. Application developers who need to use IPC should design their applications so that modules using the IPC routines described in [Section 2.7](#) can be easily modified to use the alternative interfaces.

43613

43614

43615

43616

RATIONALE

43617

None.

43618

FUTURE DIRECTIONS

43619

None.

43620

SEE ALSO

43621

[Section 2.7](#) (on page 39), [Section 2.8](#) (on page 40), *exec*, *exit()*, *fork()*, *shmctl()*, *shmdt()*, *shmget()*, *shm_open()*, *shm_unlink()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<sys/shm.h>`

43622

43623

CHANGE HISTORY

43624

First released in Issue 2. Derived from Issue 2 of the SVID.

43625

Issue 5

43626

Moved from SHARED MEMORY to BASE.

43627

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

43628

43629

Issue 6

43630

The Open Group Corrigendum U021/13 is applied.

43631 **NAME**
 43632 shmctl — XSI shared memory control operations

43633 **SYNOPSIS**

```
43634 XSI #include <sys/shm.h>
43635 int shmctl(int shmid, int cmd, struct shmids *buf);
```

43636 **DESCRIPTION**

43637 The *shmctl()* function operates on XSI shared memory (see the Base Definitions volume of
 43638 IEEE Std 1003.1-200x, Section 3.340, Shared Memory Object). It is unspecified whether this
 43639 function interoperates with the realtime interprocess communication facilities defined in [Section](#)
 43640 [2.8](#) (on page 40).

43641 The *shmctl()* function provides a variety of shared memory control operations as specified by
 43642 *cmd*. The following values for *cmd* are available:

43643 **IPC_STAT** Place the current value of each member of the **shmids** data structure
 43644 associated with *shmid* into the structure pointed to by *buf*. The contents of the
 43645 structure are defined in **<sys/shm.h>**.

43646 **IPC_SET** Set the value of the following members of the **shmids** data structure
 43647 associated with *shmid* to the corresponding value found in the structure
 43648 pointed to by *buf*:

43649 shm_perm.uid
 43650 shm_perm.gid
 43651 shm_perm.mode Low-order nine bits.

43652 **IPC_SET** can only be executed by a process that has an effective user ID equal
 43653 to either that of a process with appropriate privileges or to the value of
 43654 *shm_perm.cuid* or *shm_perm.uid* in the **shmids** data structure associated with
 43655 *shmid*.

43656 **IPC_RMID** Remove the shared memory identifier specified by *shmid* from the system and
 43657 destroy the shared memory segment and **shmids** data structure associated
 43658 with it. **IPC_RMID** can only be executed by a process that has an effective user
 43659 ID equal to either that of a process with appropriate privileges or to the value
 43660 of *shm_perm.cuid* or *shm_perm.uid* in the **shmids** data structure associated
 43661 with *shmid*.

43662 **RETURN VALUE**

43663 Upon successful completion, *shmctl()* shall return 0; otherwise, it shall return -1 and set *errno* to
 43664 indicate the error.

43665 **ERRORS**

43666 The *shmctl()* function shall fail if:

43667 [EACCES] The argument *cmd* is equal to **IPC_STAT** and the calling process does not have
 43668 read permission; see [Section 2.7](#) (on page 39).

43669 [EINVAL] The value of *shmid* is not a valid shared memory identifier, or the value of *cmd*
 43670 is not a valid command.

43671 [EPERM] The argument *cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user ID
 43672 of the calling process is not equal to that of a process with appropriate
 43673 privileges and it is not equal to the value of *shm_perm.cuid* or *shm_perm.uid* in
 43674 the data structure associated with *shmid*.

43675 The *shmctl()* function may fail if:

43676 [EOVERFLOW] The *cmd* argument is `IPC_STAT` and the *gid* or *uid* value is too large to be
43677 stored in the structure pointed to by the *buf* argument.

43678 EXAMPLES

43679 None.

43680 APPLICATION USAGE

43681 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
43682 Application developers who need to use IPC should design their applications so that modules
43683 using the IPC routines described in [Section 2.7](#) can be easily modified to use the alternative
43684 interfaces.

43685 RATIONALE

43686 None.

43687 FUTURE DIRECTIONS

43688 None.

43689 SEE ALSO

43690 [Section 2.7](#) (on page 39), [Section 2.8](#) (on page 40), *shmat()*, *shmdt()*, *shmget()*, *shm_open()*,
43691 *shm_unlink()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<sys/shm.h>`

43692 CHANGE HISTORY

43693 First released in Issue 2. Derived from Issue 2 of the SVID.

43694 Issue 5

43695 Moved from SHARED MEMORY to BASE.

43696 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
43697 DIRECTIONS to a new APPLICATION USAGE section.

43698 **NAME**
 43699 `shmdt` — XSI shared memory detach operation

43700 **SYNOPSIS**

```
43701 XSI #include <sys/shm.h>
43702 int shmdt(const void *shmaddr);
```

43703 **DESCRIPTION**

43704 The `shmdt()` function operates on XSI shared memory (see the Base Definitions volume of
 43705 IEEE Std 1003.1-200x, Section 3.340, Shared Memory Object). It is unspecified whether this
 43706 function interoperates with the realtime interprocess communication facilities defined in [Section](#)
 43707 [2.8](#) (on page 40).

43708 The `shmdt()` function detaches the shared memory segment located at the address specified by
 43709 `shmaddr` from the address space of the calling process.

43710 **RETURN VALUE**

43711 Upon successful completion, `shmdt()` shall decrement the value of `shm_nattch` in the data
 43712 structure associated with the shared memory ID of the attached shared memory segment and
 43713 return 0.

43714 Otherwise, the shared memory segment shall not be detached, `shmdt()` shall return `-1`, and `errno`
 43715 shall be set to indicate the error.

43716 **ERRORS**

43717 The `shmdt()` function shall fail if:

43718 [EINVAL] The value of `shmaddr` is not the data segment start address of a shared memory
 43719 segment.

43720 **EXAMPLES**

43721 None.

43722 **APPLICATION USAGE**

43723 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
 43724 Application developers who need to use IPC should design their applications so that modules
 43725 using the IPC routines described in [Section 2.7](#) can be easily modified to use the alternative
 43726 interfaces.

43727 **RATIONALE**

43728 None.

43729 **FUTURE DIRECTIONS**

43730 None.

43731 **SEE ALSO**

43732 [Section 2.7](#) (on page 39), [Section 2.8](#) (on page 40), `exec`, `exit()`, `fork()`, `shmat()`, `shmctl()`, `shmget()`,
 43733 `shm_open()`, `shm_unlink()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<sys/shm.h>`

43734 **CHANGE HISTORY**

43735 First released in Issue 2. Derived from Issue 2 of the SVID.

43736 **Issue 5**

43737 Moved from SHARED MEMORY to BASE.

43738 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
 43739 DIRECTIONS to a new APPLICATION USAGE section.

NAME

shmget — get an XSI shared memory segment

SYNOPSIS

```
XSI #include <sys/shm.h>
    int shmget(key_t key, size_t size, int shmflg);
```

DESCRIPTION

The *shmget()* function operates on XSI shared memory (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.340, Shared Memory Object). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in [Section 2.8](#) (on page 40).

The *shmget()* function shall return the shared memory identifier associated with *key*.

A shared memory identifier, associated data structure, and shared memory segment of at least *size* bytes (see [<sys/shm.h>](#)) are created for *key* if one of the following is true:

- The argument *key* is equal to `IPC_PRIVATE`.
- The argument *key* does not already have a shared memory identifier associated with it and (*shmflg* & `IPC_CREAT`) is non-zero.

Upon creation, the data structure associated with the new shared memory identifier shall be initialized as follows:

- The values of *shm_perm.cuid*, *shm_perm.uid*, *shm_perm.cgid*, and *shm_perm.gid* are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order nine bits of *shm_perm.mode* are set equal to the low-order nine bits of *shmflg*.
- The value of *shm_segsz* is set equal to the value of *size*.
- The values of *shm_lpid*, *shm_nattch*, *shm_atime*, and *shm_dtime* are set equal to 0.
- The value of *shm_ctime* is set equal to the current time.

When the shared memory segment is created, it shall be initialized with all zero values.

RETURN VALUE

Upon successful completion, *shmget()* shall return a non-negative integer, namely a shared

shmget()

43780	[ENOENT]	A shared memory identifier does not exist for the argument <i>key</i> and (<i>shmflg</i> &IPC_CREAT) is 0.
43781		
43782	[ENOMEM]	A shared memory identifier and associated shared memory segment shall be created, but the amount of available physical memory is not sufficient to fill the request.
43783		
43784		
43785	[ENOSPC]	A shared memory identifier is to be created, but the system-imposed limit on the maximum number of allowed shared memory identifiers system-wide would be exceeded.
43786		
43787		

EXAMPLES

None.

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication. Application developers who need to use IPC should design their applications so that modules using the IPC routines described in [Section 2.7](#) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 2.7](#) (on page 39), [Section 2.8](#) (on page 40), [shmat\(\)](#), [shmctl\(\)](#), [shmdt\(\)](#), [shm_open\(\)](#), [shm_unlink\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, [<sys/shm.h>](#)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

Moved from SHARED MEMORY to BASE.

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

NAME

shutdown — shut down socket send and receive operations

SYNOPSIS

```
#include <sys/socket.h>

int shutdown(int socket, int how);
```

DESCRIPTION

The *shutdown()* function shall cause all or part of a full-duplex connection on the socket associated with the file descriptor *socket* to be shut down.

The *shutdown()* function takes the following arguments:

<i>socket</i>	Specifies the file descriptor of the socket.
<i>how</i>	Specifies the type of shutdown. The values are as follows:
SHUT_RD	Disables further receive operations.
SHUT_WR	Disables further send operations.
SHUT_RDWR	Disables further send and receive operations.

The *shutdown()* function disables subsequent send and/or receive operations on a socket, depending on the value of the *how* argument.

RETURN VALUE

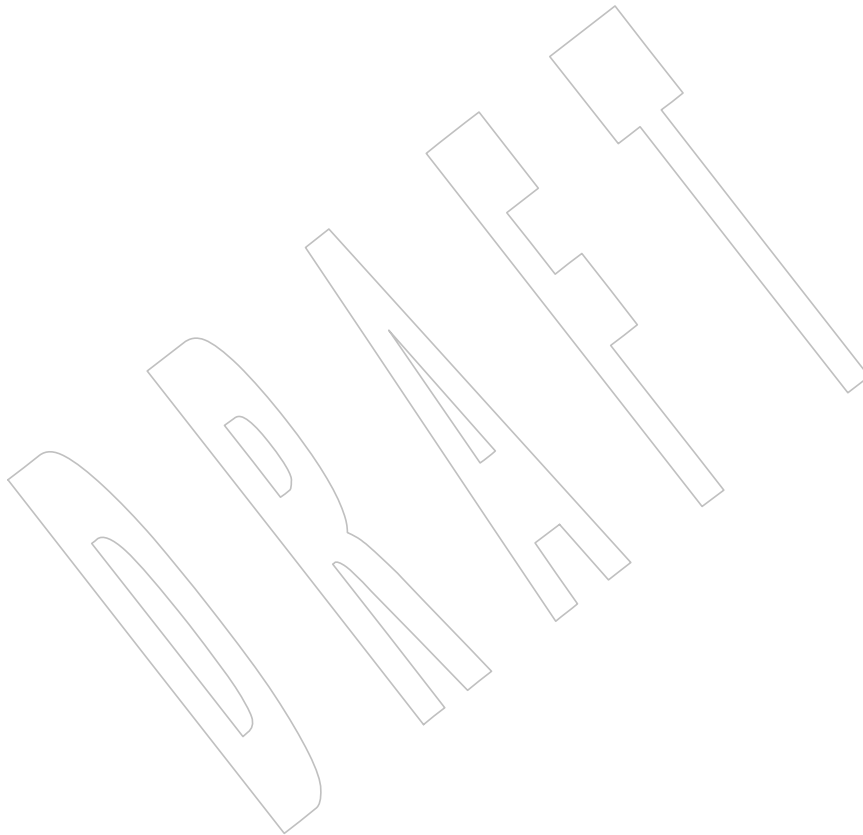
Upon successful completion, *shutdown()* shall return 0; otherwise, -1 shall be returned and *errno*

shutdown()

43846
43847

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.



43848 **NAME**

43849 sigaction — examine and change a signal action

43850 **SYNOPSIS**

```
43851 CX #include <signal.h>
43852 int sigaction(int sig, const struct sigaction *restrict act,
43853 struct sigaction *restrict oact);
```

43854 **DESCRIPTION**

43855 The *sigaction()* function allows the calling process to examine and/or specify the action to be
 43856 associated with a specific signal. The argument *sig* specifies the signal; acceptable values are
 43857 defined in **<signal.h>**.

43858 The structure **sigaction**, used to describe an action to be taken, is defined in the **<signal.h>**
 43859 header to include at least the following members:

Member Type	Member Name	Description
void(*) (int)	<i>sa_handler</i>	Pointer to a signal-catching function or one of the macros SIG_IGN or SIG_DFL.
sigset_t	<i>sa_mask</i>	Additional set of signals to be blocked during execution of signal-catching function.
int	<i>sa_flags</i>	Special flags to affect behavior of signal.
void(*) (int, siginfo_t *, void *)	<i>sa_sigaction</i>	Pointer to a signal-catching function.

43870 The storage occupied by *sa_handler* and *sa_sigaction* may overlap, and a conforming application
 43871 shall not use both simultaneously.

43872 If the argument *act* is not a null pointer, it points to a structure specifying the action to be
 43873 associated with the specified signal. If the argument *oact* is not a null pointer, the action
 43874 previously associated with the signal is stored in the location pointed to by the argument *oact*. If
 43875 the argument *act* is a null pointer, signal handling is unchanged; thus, the call can be used to
 43876 enquire about the current handling of a given signal. The SIGKILL and SIGSTOP signals shall
 43877 not be added to the signal mask using this mechanism; this restriction shall be enforced by the
 43878 system without causing an error to be indicated.

43879 If the SA_SIGINFO flag (see below) is cleared in the *sa_flags* field of the **sigaction** structure, the
 43880 *sa_handler* field identifies the action to be associated with the specified signal. If the
 43881 SA_SIGINFO flag is set in the *sa_flags* field, the *sa_sigaction* field specifies a signal-catching
 43882 function.

43883 The *sa_flags* field can be used to modify the behavior of the specified signal.

43884 The following flags, defined in the **<signal.h>** header, can be set in *sa_flags*:

43885 XSI	SA_NOCLDSTOP	Do not generate SIGCHLD when children stop or stopped children continue.
43886		
43887		If <i>sig</i> is SIGCHLD and the SA_NOCLDSTOP flag is not set in <i>sa_flags</i> , and the implementation supports the SIGCHLD signal, then a SIGCHLD signal shall be generated for the calling process whenever any of its child processes stop and a SIGCHLD signal may be generated for the calling process whenever any of its stopped child processes are continued. If <i>sig</i>
43888		
43889		
43890 XSI		
43891		

43892			is SIGCHLD and the SA_NOCLDSTOP flag is set in <i>sa_flags</i> , then the implementation shall not generate a SIGCHLD signal in this way.
43893			
43894	XSI	SA_ONSTACK	If set and an alternate signal stack has been declared with <i>sigaltstack()</i> , the signal shall be delivered to the calling process on that stack. Otherwise, the signal shall be delivered on the current stack.
43895			
43896			
43897		SA_RESETHAND	If set, the disposition of the signal shall be reset to SIG_DFL and the SA_SIGINFO flag shall be cleared on entry to the signal handler.
43898			
43899			Note: SIGILL and SIGTRAP cannot be automatically reset when delivered;
43900			the system silently enforces this restriction.
43901			Otherwise, the disposition of the signal shall not be modified on entry to the signal handler.
43902			
43903			In addition, if this flag is set, <i>sigaction()</i> behaves as if the SA_NODEFER flag were also set.
43904			
43905		SA_RESTART	This flag affects the behavior of interruptible functions; that is, those specified to fail with <i>errno</i> set to [EINTR]. If set, and a function specified as interruptible is interrupted by this signal, the function shall restart and shall not fail with [EINTR] unless otherwise specified. If an interruptible function which uses a timeout is restarted, the duration of the timeout following the restart is set to an unspecified value that does not exceed the original timeout value. If the flag is not set, interruptible functions interrupted by this signal shall fail with <i>errno</i> set to [EINTR].
43906			
43907			
43908			
43909			
43910			
43911			
43912			
43913		SA_SIGINFO	If cleared and the signal is caught, the signal-catching function shall be entered as:
43914			
43915			<pre>void func(int signo);</pre>
43916			where <i>signo</i> is the only argument to the signal-catching function. In this case, the application shall use the <i>sa_handler</i> member to describe the signal-catching function and the application shall not modify the <i>sa_sigaction</i> member.
43917			
43918			
43919			
43920			If SA_SIGINFO is set and the signal is caught, the signal-catching function shall be entered as:
43921			
43922			<pre>void func(int signo, siginfo_t *info, void *context);</pre>
43923			where two additional arguments are passed to the signal-catching function. The second argument shall point to an object of type siginfo_t explaining the reason why the signal was generated; the third argument can be cast to a pointer to an object of type ucontext_t to refer to the receiving thread's context that was interrupted when the signal was delivered. In this case, the application shall use the <i>sa_sigaction</i> member to describe the signal-catching function and the application shall not modify the <i>sa_handler</i> member.
43924			
43925			
43926			
43927			
43928			
43929			
43930			
43931			The <i>si_signo</i> member contains the system-generated signal number.
43932	XSI		The <i>si_errno</i> member may contain implementation-defined additional error information; if non-zero, it contains an error number identifying the condition that caused the signal to be generated.
43933			
43934			
43935			The <i>si_code</i> member contains a code identifying the cause of the signal.
43936			If the value of <i>si_code</i> is less than or equal to 0, then the signal was generated by a process and <i>si_pid</i> and <i>si_uid</i> , respectively, indicate the process ID and the real user ID of the sender. The <signal.h> header
43937			
43938			

- 43939 description contains information about the signal-specific contents of the
43940 elements of the **siginfo_t** type.
- 43941 **SA_NOCLDWAIT** If set, and *sig* equals SIGCHLD, child processes of the calling processes
43942 shall not be transformed into zombie processes when they terminate. If
43943 the calling process subsequently waits for its children, and the process has
43944 no unwaited-for children that were transformed into zombie processes, it
43945 shall block until all of its children terminate, and *wait()*, *waitid()*, and
43946 *waitpid()* shall fail and set *errno* to [ECHILD]. Otherwise, terminating
43947 child processes shall be transformed into zombie processes, unless
43948 SIGCHLD is set to SIG_IGN.
- 43949 **SA_NODEFER** If set and *sig* is caught, *sig* shall not be added to the thread's signal mask
43950 on entry to the signal handler unless it is included in *sa_mask*. Otherwise,
43951 *sig* shall always be added to the thread's signal mask on entry to the
43952 signal handler.
- 43953 When a signal is caught by a signal-catching function installed by *sigaction()*, a new signal mask
43954 is calculated and installed for the duration of the signal-catching function (or until a call to either
43955 *sigprocmask()* or *sigsuspend()* is made). This mask is formed by taking the union of the current
43956 signal mask and the value of the *sa_mask* for the signal being delivered, and unless
43957 SA_NODEFER or SA_RESETHAND is set, then including the signal being delivered. If and
43958 when the user's signal handler returns normally, the original signal mask is restored.
- 43959 Once an action is installed for a specific signal, it shall remain installed until another action is
43960 explicitly requested (by another call to *sigaction()*), until the SA_RESETHAND flag causes
43961 resetting of the handler, or until one of the *exec* functions is called.
- 43962 If the previous action for *sig* had been established by *signal()*, the values of the fields returned in
43963 the structure pointed to by *oact* are unspecified, and in particular *oact->sa_handler* is not
43964 necessarily the same value passed to *signal()*. However, if a pointer to the same structure or a
43965 copy thereof is passed to a subsequent call to *sigaction()* via the *act* argument, handling of the
43966 signal shall be as if the original call to *signal()* were repeated.
- 43967 If *sigaction()* fails, no new signal handler is installed.
- 43968 It is unspecified whether an attempt to set the action for a signal that cannot be caught or
43969 ignored to SIG_DFL is ignored or causes an error to be returned with *errno* set to [EINVAL].
- 43970 If SA_SIGINFO is not set in *sa_flags*, then the disposition of subsequent occurrences of *sig* when
43971 it is already pending is implementation-defined; the signal-catching function shall be invoked
43972 with a single argument. If SA_SIGINFO is set in *sa_flags*, then subsequent occurrences of *sig*
43973 generated by *sigqueue()* or as a result of any signal-generating function that supports the
43974 specification of an application-defined value (when *sig* is already pending) shall be queued in
43975 FIFO order until delivered or accepted; the signal-catching function shall be invoked with three
43976 arguments. The application specified value is passed to the signal-catching function as the
43977 *si_value* member of the **siginfo_t** structure.
- 43978 The result of the use of *sigaction()* and a *sigwait()* function concurrently within a process on the
43979 same signal is unspecified.
- 43980 **RETURN VALUE**
- 43981 Upon successful completion, *sigaction()* shall return 0; otherwise, -1 shall be returned, *errno* shall
43982 be set to indicate the error, and no new signal-catching function shall be installed.
- 43983 **ERRORS**
- 43984 The *sigaction()* function shall fail if:

- 43985 [EINVAL] The *sig* argument is not a valid signal number or an attempt is made to catch a
43986 signal that cannot be caught or ignore a signal that cannot be ignored.
- 43987 [ENOTSUP] The SA_SIGINFO bit flag is set in the *sa_flags* field of the **sigaction** structure.
- 43988 The *sigaction()* function may fail if:
- 43989 [EINVAL] An attempt was made to set the action to SIG_DFL for a signal that cannot be
43990 caught or ignored (or both).
- 43991 In addition, the *sigaction()* function may fail if the SA_SIGINFO flag is set in the *sa_flags* field of
43992 the **sigaction** structure for a signal not in the range SIGRTMIN to SIGRTMAX.

EXAMPLES**Establishing a Signal Handler**

The following example demonstrates the use of *sigaction()* to establish a handler for the SIGINT signal.

```
43997 #include <signal.h>
43998 static void handler(int signum)
43999 {
44000     /* Take appropriate actions for signal delivery */
44001 }
44002 int main()
44003 {
44004     struct sigaction sa;
44005     sa.sa_handler = handler;
44006     sigemptyset(&sa.sa_mask);
44007     sa.sa_flags = SA_RESTART; /* Restart functions if
44008                             interrupted by handler */
44009     if (sigaction(SIGINT, &sa, NULL) == -1)
44010         /* Handle error */;
44011     /* Further code */
44012 }
```

APPLICATION USAGE

The *sigaction()* function supersedes the *signal()* function, and should be used in preference. In particular, *sigaction()* and *signal()* should not be used in the same process to control the same signal. The behavior of reentrant functions, as defined in the DESCRIPTION, is as specified by this volume of IEEE Std 1003.1-200x, regardless of invocation from a signal-catching function. This is the only intended meaning of the statement that reentrant functions may be used in signal-catching functions without restrictions. Applications must still consider all effects of such functions on such things as data structures, files, and process state. In particular, application writers need to consider the restrictions on interactions when interrupting *sleep()* and interactions among multiple handles for a file description. The fact that any specific function is listed as reentrant does not necessarily mean that invocation of that function from a signal-catching function is recommended.

In order to prevent errors arising from interrupting non-reentrant function calls, applications should protect calls to these functions either by blocking the appropriate signals or through the use of some programmatic semaphore (see *semget()*, *sem_init()*, *sem_open()*, and so on). Note in particular that even the "safe" functions may modify *errno*; the signal-catching function, if not executing as an independent thread, may want to save and restore its value. Naturally, the same principles apply to the reentrancy of application routines and asynchronous data access. Note that *longjmp()* and *siglongjmp()* are not in the list of reentrant functions. This is because the code

executing after *longjmp()* and *siglongjmp()* can call any unsafe functions with the same danger as calling those unsafe functions directly from the signal handler. Applications that use *longjmp()* and *siglongjmp()* from within signal handlers require rigorous protection in order to be portable. Many of the other functions that are excluded from the list are traditionally implemented using either *malloc()* or *free()* functions or the standard I/O library, both of which traditionally use data structures in a non-reentrant manner. Since any combination of different functions using a common data structure can cause reentrancy problems, this volume of IEEE Std 1003.1-200x does not define the behavior when any unsafe function is called in a signal handler that interrupts an unsafe function.

If the signal occurs other than as the result of calling *abort()*, *kill()*, or *raise()*, the behavior is undefined if the signal handler calls any function in the standard library other than one of the functions listed in the table above or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type **volatile sig_atomic_t**. Furthermore, if such a call fails, the value of *errno* is unspecified.

Usually, the signal is executed on the stack that was in effect before the signal was delivered. An alternate stack may be specified to receive a subset of the signals being caught.

When the signal handler returns, the receiving thread resumes execution at the point it was interrupted unless the signal handler makes other arrangements. If *longjmp()* or *_longjmp()* is used to leave the signal handler, then the signal mask must be explicitly restored.

This volume of IEEE Std 1003.1-200x defines the third argument of a signal handling function when SA_SIGINFO is set as a **void *** instead of a **ucontext_t ***, but without requiring type checking. New applications should explicitly cast the third argument of the signal handling function to **ucontext_t ***.

The BSD optional four argument signal handling function is not supported by this volume of IEEE Std 1003.1-200x. The BSD declaration would be:

```
void handler(int sig, int code, struct sigcontext *scp,
             char *addr);
```

where *sig* is the signal number, *code* is additional information on certain signals, *scp* is a pointer to the **sigcontext** structure, and *addr* is additional address information. Much the same information is available in the objects pointed to by the second argument of the signal handler specified when SA_SIGINFO is set.

RATIONALE

Although this volume of IEEE Std 1003.1-200x requires that signals that cannot be ignored shall not be added to the signal mask when a signal-catching function is entered, there is no explicit requirement that subsequent calls to *sigaction()* reflect this in the information returned in the *oact* argument. In other words, if SIGKILL is included in the *sa_mask* field of *act*, it is unspecified whether or not a subsequent call to *sigaction()* returns with SIGKILL included in the *sa_mask* field of *oact*.

The SA_NOCLDSTOP flag, when supplied in the *act->sa_flags* parameter, allows overloading SIGCHLD with the System V semantics that each SIGCLD signal indicates a single terminated child. Most conforming applications that catch SIGCHLD are

44081 This volume of IEEE Std 1003.1-200x requires that calls to *sigaction()* that supply a NULL *act*
 44082 argument succeed, even in the case of signals that cannot be caught or ignored (that is, SIGKILL
 44083 or SIGSTOP). The System V *signal()* and BSD *sigvec()* functions return [EINVAL] in these cases
 44084 and, in this respect, their behavior varies from *sigaction()*.

44085 This volume of IEEE Std 1003.1-200x requires that *sigaction()* properly save and restore a signal
 44086 action set up by the ISO C standard *signal()* function. However, there is no guarantee that the
 44087 reverse is true, nor could there be given the greater amount of information conveyed by the
 44088 **sigaction** structure. Because of this, applications should avoid using both functions for the same
 44089 signal in the same process. Since this cannot always be avoided in case of general-purpose
 44090 library routines, they should always be implemented with *sigaction()*.

44091 It was intended that the *signal()* function should be implementable as a library routine using
 44092 *sigaction()*.

44093 The POSIX Realtime Extension extends the *sigaction()* function as specified by the POSIX.1-1990
 44094 standard to allow the application to request on a per-signal basis via an additional signal action
 44095 flag that the extra parameters, including the application-defined signal value, if any, be passed to
 44096 the signal-catching function.

44097 FUTURE DIRECTIONS

44098 None.

44099 SEE ALSO

44100 Section 2.4 (on page 28), *exec*, *kill()*, *_longjmp()*, *longjmp()*, *raise()*, *semget()*, *sem_init()*,
 44101 *sem_open()*, *sigaddset()*, *sigaltstack()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*, *signal()*,
 44102 *sigprocmask()*, *sigsuspend()*, *wait()*, *waitid()*, *waitpid()*, the Base Definitions volume of
 44103 IEEE Std 1003.1-200x, <**signal.h**>

44104 CHANGE HISTORY

44105 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

44106 Issue 5

44107 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and POSIX
 44108 Threads Extension.

44109 In the DESCRIPTION, the second argument to *func* when SA_SIGINFO is set is no longer
 44110 permitted to be NULL, and the description of permitted **siginfo_t** contents is expanded by
 44111 reference to <**signal.h**>.

44112 Since the X/OPEN UNIX Extension functionality is now folded into the BASE, the [ENOTSUP]
 44113 error is deleted.

44114 Issue 6

44115 The Open Group Corrigendum U028/7 is applied. In the paragraph entitled “Signal Effects on
 44116 Other Functions”, a reference to *sigpending()* is added.

44117 In the DESCRIPTION, the text “Signal Generation and Delivery”, “Signal Actions”, and “Signal
 44118 Effects on Other Functions” are moved to a separate section of this volume of
 44119 IEEE Std 1003.1-200x.

44120 Text describing functionality from the Realtime Signals Extension option is marked.

44121 The following changes are made for alignment with the ISO POSIX-1:1996 standard:

- 44122 • The [ENOTSUP] error condition is added.

44123 The normative text is updated to avoid use of the term “must” for application requirements.

44124 The **restrict** keyword is added to the *sigaction()* prototype for alignment with the
 44125 ISO/IEC 9899:1999 standard.

44126 References to the *wait3()* function are removed.

- 44127 The SYNOPSIS is marked CX since the presence of this function in the <signal.h> header is an
44128 extension over the ISO C standard.
- 44129 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/57 is applied, changing text in the table
44130 describing the **sigaction** structure.
- 44131 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/127 is applied, removing text from the
44132 DESCRIPTION duplicated later in the same section.
- 44133 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/128 is applied, updating the
44134 DESCRIPTION and APPLICATION USAGE sections. Changes are made to refer to the thread
44135 rather than the process.
- 44136 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/129 is applied, adding the example to the
44137 EXAMPLES section.
- 44138 **Issue 7**
- 44139 Austin Group Interpretations 1003.1-2001 #065 and #084 are applied, clarifying the role of the
44140 SA_NODEFER flag with respect to the signal mask, and clarifying the SA_RESTART flag for
44141 interrupted functions which use timeouts.
- 44142 Austin Group Interpretation 1003.1-2001 #004 is applied.
- 44143 Functionality relating to the Realtime Signals Extension option is moved to the Base.

DRAFT

44144 **NAME**

44145 sigaddset — add a signal to a signal set

44146 **SYNOPSIS**

```
44147 CX #include <signal.h>
44148 int sigaddset(sigset_t *set, int signo);
```

44149 **DESCRIPTION**

44150 The *sigaddset()* function adds the individual signal specified by the *signo* to the signal set pointed
44151 to by *set*.

44152 Applications shall call either *sigemptyset()* or *sigfillset()* at least once for each object of type
44153 **sigset_t** prior to any other use of that object. If such an object is not initialized in this way, but is
44154 nonetheless supplied as an argument to any of *pthread_sigmask()*, *sigaction()*, *sigaddset()*,
44155 *sigdelset()*, *sigismember()*, *sigpending()*, *sigprocmask()*, *sigsuspend()*, *sigtimedwait()*, *sigwait()*, or
44156 *sigwaitinfo()*, the results are undefined.

44157 **RETURN VALUE**

44158 Upon successful completion, *sigaddset()* shall return 0; otherwise, it shall return -1 and set *errno*
44159 to indicate the error.

44160 **ERRORS**

44161 The *sigaddset()* function may fail if:

44162 [EINVAL] The value of the *signo* argument is an invalid or unsupported signal number.

44163 **EXAMPLES**

44164 None.

44165 **APPLICATION USAGE**

44166 None.

44167 **RATIONALE**

44168 None.

44169 **FUTURE DIRECTIONS**

44170 None.

44171 **SEE ALSO**

44172 Section 2.4 (on page 28), *sigaction()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*,
44173 *sigpending()*, *sigprocmask()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-200x,
44174 **<signal.h>**

44175 **CHANGE HISTORY**

44176 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

44177 **Issue 5**

44178 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
44179 previous issues.

44180 **Issue 6**

44181 The normative text is updated to avoid use of the term “must” for application requirements.

44182 The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an
44183 extension over the ISO C standard.

44184 **NAME**
 44185 sigaltstack — set and get signal alternate stack context

44186 **SYNOPSIS**

```
44187 XSI #include <signal.h>
44188 int sigaltstack(const stack_t *restrict ss, stack_t *restrict oss);
```

44189 **DESCRIPTION**

44190 The *sigaltstack()* function allows a process to define and examine the state of an alternate stack
 44191 for signal handlers for the current thread. Signals that have been explicitly declared to execute
 44192 on the alternate stack shall be delivered on the alternate stack.

44193 If *ss* is not a null pointer, it points to a **stack_t** structure that specifies the alternate signal stack
 44194 that shall take effect upon return from *sigaltstack()*. The *ss_flags* member specifies the new stack
 44195 state. If it is set to *SS_DISABLE*, the stack is disabled and *ss_sp* and *ss_size* are ignored.
 44196 Otherwise, the stack shall be enabled, and the *ss_sp* and *ss_size* members specify the new address
 44197 and size of the stack.

44198 The range of addresses starting at *ss_sp* up to but not including *ss_sp+ss_size* is available to the
 44199 implementation for use as the stack. This function makes no assumptions regarding which end
 44200 is the stack base and in which direction the stack grows as items are pushed.

44201 If *oss* is not a null pointer, on successful completion it shall point to a **stack_t** structure that
 44202 specifies the alternate signal stack that was in effect prior to the call to *sigaltstack()*. The *ss_sp*
 44203 and *ss_size* members specify the address and size of that stack. The *ss_flags* member specifies the
 44204 stack's state, and may contain one of the following values:

44205 **SS_ONSTACK** The process is currently executing on the alternate signal stack. Attempts to
 44206 modify the alternate signal stack while the process is executing on it fail. This
 44207 flag shall not be modified by processes.

44208 **SS_DISABLE** The alternate signal stack is currently disabled.

44209 The value *SIGSTKSZ* is a system default specifying the number of bytes that would be used to
 44210 cover the usual case when manually allocating an alternate stack area. The value *MINSIGSTKSZ*
 44211 is defined to be the minimum stack size for a signal handler. In computing an alternate stack
 44212 size, a program should add that amount to its stack requirements to allow for the system
 44213 implementation overhead. The constants *SS_ONSTACK*, *SS_DISABLE*, *SIGSTKSZ*, and
 44214 *MINSIGSTKSZ* are defined in **<signal.h>**.

44215 After a successful call to one of the *exec* functions, there are no alternate signal stacks in the new
 44216 process image.

44217 In some implementations, a signal (whether or not indicated to execute on the alternate stack)
 44218 shall always execute on the alternate stack if it is delivered while another signal is being caught
 44219 using the alternate stack.

44220 Use of this function by library threads that are not bound to kernel-scheduled entities results in
 44221 undefined behavior.

44222 **RETURN VALUE**

44223 Upon successful completion, *sigaltstack()* shall return 0; otherwise, it shall return -1 and set *errno*
 44224 to indicate the error.

44225 ERRORS

- 44226 The *sigaltstack()* function shall fail if:
- 44227 [EINVAL] The *ss* argument is not a null pointer, and the *ss_flags* member pointed to by *ss*
44228 contains flags other than SS_DISABLE.
- 44229 [ENOMEM] The size of the alternate stack area is less than MINSIGSTKSZ.
- 44230 [EPERM] An attempt was made to modify an active stack.

44231 EXAMPLES**44232 Allocating Memory for an Alternate Stack**

44233 The following example illustrates a method for allocating memory for an alternate stack.

```
44234 #include <signal.h>
44235 ...
44236 if ((sigstk.ss_sp = malloc(SIGSTKSZ)) == NULL)
44237     /* Error return. */
44238     sigstk.ss_size = SIGSTKSZ;
44239     sigstk.ss_flags = 0;
44240     if (sigaltstack(&sigstk, (stack_t *)0) < 0)
44241         perror("sigaltstack");
```

44242 APPLICATION USAGE

44243 On some implementations, stack space is automatically extended as needed. On those
44244 implementations, automatic extension is typically not available for an alternate stack. If the stack
44245 overflows, the behavior is undefined.

44246 RATIONALE

44247 None.

44248 FUTURE DIRECTIONS

44249 None.

44250 SEE ALSO

44251 [Section 2.4](#) (on page 28), *exec*, *sigaction()*, *sigsetjmp()*, the Base Definitions volume of
44252 IEEE Std 1003.1-200x, **<signal.h>**

44253 CHANGE HISTORY

44254 First released in Issue 4, Version 2.

44255 Issue 5

44256 Moved from X/OPEN UNIX extension to BASE.

44257 The last sentence of the DESCRIPTION was included as an APPLICATION USAGE note in
44258 previous issues.

44259 Issue 6

44260 The normative text is updated to avoid use of the term “must” for application requirements.

44261 The **restrict** keyword is added to the *sigaltstack()* prototype for alignment with the
44262 ISO/IEC 9899:1999 standard.

44263 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/58 is applied, updating the first sentence
44264 to include “for the current thread”.

44265 **NAME**
 44266 sigdelset — delete a signal from a signal set

44267 **SYNOPSIS**

44268 CX

```
#include <signal.h>
```


 44269

```
int sigdelset(sigset_t *set, int signo);
```

44270 **DESCRIPTION**

44271 The *sigdelset()* function deletes the individual signal specified by *signo* from the signal set
 44272 pointed to by *set*.

44273 Applications should call either *sigemptyset()* or *sigfillset()* at least once for each object of type
 44274 **sigset_t** prior to any other use of that object. If such an object is not initialized in this way, but is
 44275 nonetheless supplied as an argument to any of *pthread_sigmask()*, *sigaction()*, *sigaddset()*,
 44276 *sigdelset()*, *sigismember()*, *sigpending()*, *sigprocmask()*, *sigsuspend()*, *sigtimedwait()*, *sigwait()*, or
 44277 *sigwaitinfo()*, the results are undefined.

44278 **RETURN VALUE**

44279 Upon successful completion, *sigdelset()* shall return 0; otherwise, it shall return -1 and set *errno*
 44280 to indicate the error.

44281 **ERRORS**

44282 The *sigdelset()* function may fail if:

44283 [EINVAL] The *signo* argument is not a valid signal number, or is an unsupported signal
 44284 number.

44285 **EXAMPLES**

44286 None.

44287 **APPLICATION USAGE**

44288 None.

44289 **RATIONALE**

44290 None.

44291 **FUTURE DIRECTIONS**

44292 None.

44293 **SEE ALSO**

44294 Section 2.4 (on page 28), *sigaction()*, *sigaddset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*,
 44295 *sigpending()*, *sigprocmask()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-200x,
 44296 <**signal.h**>

44297 **CHANGE HISTORY**

44298 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

44299 **Issue 5**

44300 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
 44301 previous issues.

44302 **Issue 6**

44303 The SYNOPSIS is marked CX since the presence of this function in the <**signal.h**> header is an
 44304 extension over the ISO C standard.

44305 **NAME**
 44306 sigemptyset — initialize and empty a signal set

44307 **SYNOPSIS**

44308 CX `#include <signal.h>`
 44309 `int sigemptyset(sigset_t *set);`

44310 **DESCRIPTION**

44311 The *sigemptyset()* function initializes the signal set pointed to by *set*, such that all signals defined
 44312 in IEEE Std 1003.1-200x are excluded.

44313 **RETURN VALUE**

44314 Upon successful completion, *sigemptyset()* shall return 0; otherwise, it shall return -1 and set
 44315 *errno* to indicate the error.

44316 **ERRORS**

44317 No errors are defined.

44318 **EXAMPLES**

44319 None.

44320 **APPLICATION USAGE**

44321 None.

44322 **RATIONALE**

44323 The implementation of the *sigemptyset()* (or *sigfillset()*) function could quite trivially clear (or set)
 44324 all the bits in the signal set. Alternatively, it would be reasonable to initialize part of the
 44325 structure, such as a version field, to permit binary-compatibility between releases where the size
 44326 of the set varies. For such reasons, either *sigemptyset()* or *sigfillset()* must be called prior to any
 44327 other use of the signal set, even if such use is read-only (for example, as an argument to
 44328 *sigpending()*). This function is not intended for dynamic allocation.

44329 The *sigfillset()* and *sigemptyset()* functions require that the resulting signal set include (or
 44330 exclude) all the signals defined in this volume of IEEE Std 1003.1-200x. Although it is outside the
 44331 scope of this volume of IEEE Std 1003.1-200x to place this requirement on signals that are
 44332 implemented as extensions, it is recommended that implementation-defined signals also be
 44333 affected by these functions. However, there may be a good reason for a particular signal not to
 44334 be affected. For example, blocking or ignoring an implementation-defined signal may have
 44335 undesirable side effects, whereas the default action for that signal is harmless. In such a case, it
 44336 would be preferable for such a signal to be excluded from the signal set returned by *sigfillset()*.

44337 In early proposals there was no distinction between invalid and unsupported signals (the names
 44338 of optional signals that were not supported by an implementation were not defined by that
 44339 implementation). The [EINVAL] error was thus specified as a required error for invalid signals.
 44340 With that distinction, it is not necessary to require implementations of these functions to
 44341 determine whether an optional signal is actually supported, as that could have a significant
 44342 performance impact for little value. The error could have been required for invalid signals and
 44343 optional for unsupported signals, but this seemed unnecessarily complex. Thus, the error is
 44344 optional in both cases.

44345 **FUTURE DIRECTIONS**

44346 None.

44347

SEE ALSO

44348

Section 2.4 (on page 28), *sigaction()*, *sigaddset()*, *sigdelset()*, *sigfillset()*, *sigismember()*, *sigpending()*, *sigprocmask()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

44349

44350

CHANGE HISTORY

44351

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

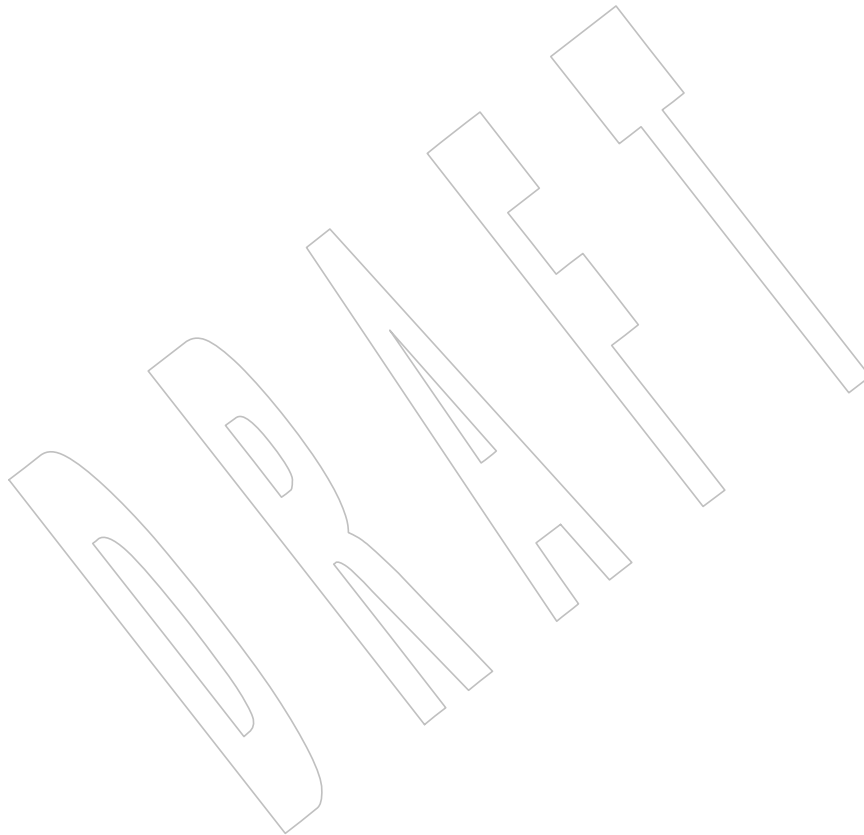
44352

Issue 6

44353

The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an extension over the ISO C standard.

44354



44355 **NAME**
 44356 sigfillset — initialize and fill a signal set

44357 **SYNOPSIS**

44358 CX #include <signal.h>
 44359 int sigfillset(sigset_t *set);

44360 **DESCRIPTION**

44361 The *sigfillset()* function shall initialize the signal set pointed to by *set*, such that all signals
 44362 defined in this volume of IEEE Std 1003.1-200x are included.

44363 **RETURN VALUE**

44364 Upon successful completion, *sigfillset()* shall return 0; otherwise, it shall return -1 and set *errno*
 44365 to indicate the error.

44366 **ERRORS**

44367 No errors are defined.

44368 **EXAMPLES**

44369 None.

44370 **APPLICATION USAGE**

44371 None.

44372 **RATIONALE**

44373 Refer to *sigemptyset()* (on page 1428).

44374 **FUTURE DIRECTIONS**

44375 None.

44376 **SEE ALSO**

44377 Section 2.4 (on page 28), *sigaction()*, *sigaddset()*, *sigdelset()*, *sigemptyset()*, *sigismember()*,
 44378 *sigpending()*, *sigprocmask()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-200x,
 44379 <signal.h>

44380 **CHANGE HISTORY**

44381 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

44382 **Issue 6**

44383 The SYNOPSIS is marked CX since the presence of this function in the <signal.h> header is an
 44384 extension over the ISO C standard.

44385 **NAME**
 44386 sighold, sigignore, sigpause, sigrelse, sigset — signal management

44387 **SYNOPSIS**

```
44388 OB XSI #include <signal.h>
44389
44389 int sighold(int sig);
44390 int sigignore(int sig);
44391 int sigpause(int sig);
44392 int sigrelse(int sig);
44393 void (*sigset(int sig, void (*disp)(int)))(int);
```

44394 **DESCRIPTION**

44395 Use of any of these functions is unspecified in a multi-threaded process.

44396 The *sighold()*, *sigignore()*, *sigpause()*, *sigrelse()*, and *sigset()* functions provide simplified signal
 44397 management.

44398 The *sigset()* function shall modify signal dispositions. The *sig* argument specifies the signal,
 44399 which may be any signal except SIGKILL and SIGSTOP. The *disp* argument specifies the signal's
 44400 disposition, which may be SIG_DFL, SIG_IGN, or the address of a signal handler. If *sigset()* is
 44401 used, and *disp* is the address of a signal handler, the system shall add *sig* to the signal mask of
 44402 the calling process before executing the signal handler; when the signal handler returns, the
 44403 system shall restore the signal mask of the calling process to its state prior to the delivery of the
 44404 signal. In addition, if *sigset()* is used, and *disp* is equal to SIG_HOLD, *sig* shall be added to the
 44405 signal mask of the calling process and *sig*'s disposition shall remain unchanged. If *sigset()* is
 44406 used, and *disp* is not equal to SIG_HOLD, *sig* shall be removed from the signal mask of the
 44407 calling process.

44408 The *sighold()* function shall add *sig* to the signal mask of the calling process.

44409 The *sigrelse()* function shall remove *sig* from the signal mask of the calling process.

44410 The *sigignore()* function shall set the disposition of *sig* to SIG_IGN.

44411 The *sigpause()* function shall remove *sig* from the signal mask of the calling process and suspend
 44412 the calling process until a signal is received. The *sigpause()* function shall restore the signal mask
 44413 of the process to its original state before returning.

44414 If the action for the SIGCHLD signal is set to SIG_IGN, child processes of the calling processes
 44415 shall not be transformed into zombie processes when they terminate. If the calling process
 44416 subsequently waits for its children, and the process has no unwaited-for children that were
 44417 transformed into zombie processes, it shall block until all of its children terminate, and *wait()*,
 44418 *waitid()*, and *waitpid()* shall fail and set *errno* to [ECHILD].

44419 **RETURN VALUE**

44420 Upon successful completion, *sigset()* shall return SIG_HOLD if the signal had been blocked and
 44421 the signal's previous disposition if it had not been blocked. Otherwise, SIG_ERR shall be
 44422 returned and *errno* set to indicate the error.

44423 The *sigpause()* function shall suspend execution of the thread until a signal is received,
 44424 whereupon it shall return -1 and set *errno* to [EINTR].

44425 For all other functions, upon successful completion, 0 shall be returned. Otherwise, -1 shall be
 44426 returned and *errno* set to indicate the error.

44427 ERRORS

44428 These functions shall fail if:

44429 [EINVAL] The *sig* argument is an illegal signal number.

44430 The *sigset()* and *sigignore()* functions shall fail if:

44431 [EINVAL] An attempt is made to catch a signal that cannot be caught, or to ignore a
44432 signal that cannot be ignored.

44433 EXAMPLES

44434 None.

44435 APPLICATION USAGE

44436 The *sigaction()* function provides a more comprehensive and reliable mechanism for controlling
44437 signals; new applications should use the *sigaction()* function instead of the obsolescent *sigset()*
44438 function.

44439 The *sighold()* function, in conjunction with *sigelse()* or *sigpause()*, may be used to establish
44440 critical regions of code that require the delivery of a signal to be temporarily deferred. For
44441 broader portability, the *pthread_sigmask()* or *sigprocmask()* functions should be used instead of
44442 the obsolescent *sighold()* and *sigelse()* functions.

44443 For broader portability, the *sigsuspend()* function should be used instead of the obsolescent
44444 *sigpause()* function.

44445 RATIONALE

44446 Each of these historic functions has a direct analog in the other functions which are required to
44447 be per-thread and thread-safe (aside from *sigprocmask()*, which is replaced by *pthread_sigmask()*).
44448 The *sigset()* function can be implemented as a simple wrapper for *sigaction()*. The *sighold()*
44449 function is equivalent to *sigprocmask()* or *pthread_sigmask()* with SIG_BLOCK set. The *sigignore()*
44450 function is equivalent to *sigaction()* with SIG_IGN set. The *sigpause()* function is equivalent to
44451 *sigsuspend()*. The *sigelse()* function is equivalent to *sigprocmask()* or *pthread_sigmask()* with
44452 SIG_UNBLOCK set.

44453 FUTURE DIRECTIONS

44454 These functions may be removed in a future version.

44455 SEE ALSO

44456 Section 2.4 (on page 28), *exec*, *pause()*, *pthread_sigmask()*, *sigaction()*, *signal()*, *sigsuspend()*,
44457 *waitid()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**signal.h**>

44458 CHANGE HISTORY

44459 First released in Issue 4, Version 2.

44460 Issue 5

44461 Moved from X/OPEN UNIX extension to BASE.

44462 The DESCRIPTION is updated to indicate that the *sigpause()* function restores the signal mask of
44463 the process to its original state before returning.

44464 The RETURN VALUE section is updated to indicate that the *sigpause()* function suspends
44465 execution of the process until a signal is received, whereupon it returns -1 and sets *errno* to
44466 [EINTR].

44467 Issue 6

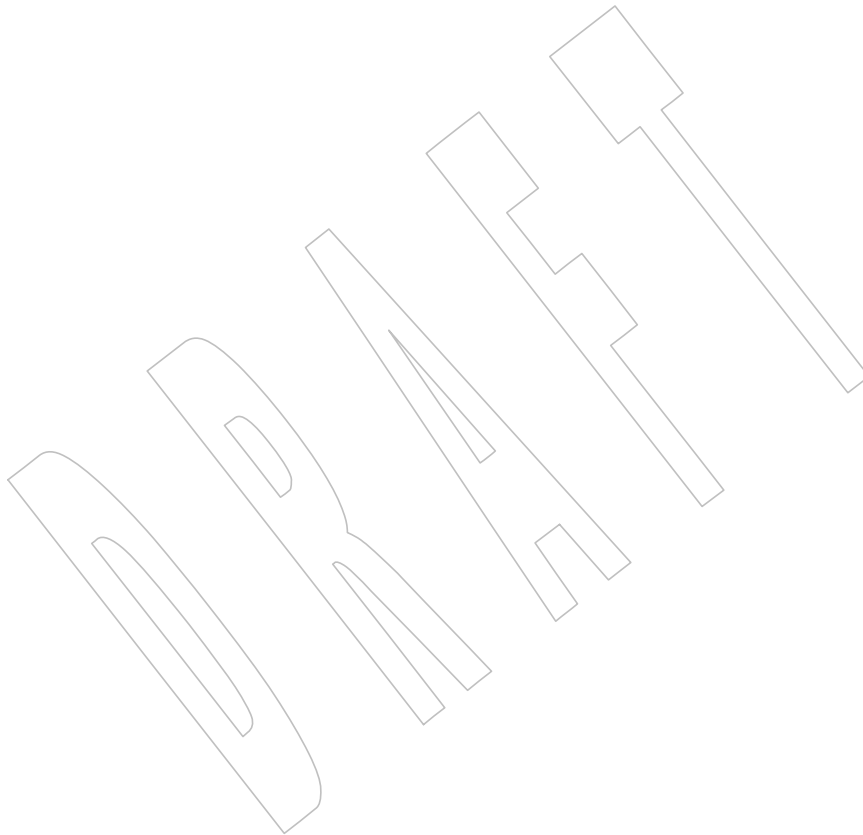
44468 The normative text is updated to avoid use of the term “must” for application requirements.

44469 References to the *wait3()* function are removed.

44470 The XSI functions are split out into their own reference page.

44471
44472
44473**Issue 7**

SD5-XSH-ERN-113 and SD5-XSH-ERN-42 are applied, marking these functions obsolescent and updating the APPLICATION USAGE and RATIONALE sections.



44474 **NAME**
 44475 siginterrupt — allow signals to interrupt functions

44476 **SYNOPSIS**

44477 OB XSI #include <signal.h>
 44478 int siginterrupt(int sig, int flag);

44479 **DESCRIPTION**

44480 The *siginterrupt()* function shall change the restart behavior when a function is interrupted by
 44481 the specified signal. The function *siginterrupt(sig, flag)* has an effect as if implemented as:

```
44482 int siginterrupt(int sig, int flag) {
44483     int ret;
44484     struct sigaction act;
44485
44486     (void) sigaction(sig, NULL, &act);
44487     if (flag)
44488         act.sa_flags &= ~SA_RESTART;
44489     else
44490         act.sa_flags |= SA_RESTART;
44491     ret = sigaction(sig, &act, NULL);
44492     return ret;
44493 }
```

44493 **RETURN VALUE**

44494 Upon successful completion, *siginterrupt()* shall return 0; otherwise, -1 shall be returned and
 44495 *errno* set to indicate the error.

44496 **ERRORS**

44497 The *siginterrupt()* function shall fail if:

44498 [EINVAL] The *sig* argument is not a valid signal number.

44499 **EXAMPLES**

44500 None.

44501 **APPLICATION USAGE**

44502 The *siginterrupt()* function supports programs written to historical system interfaces.
 44503 Applications should use the *sigaction()* with the SA_RESTART flag instead of the obsolescent
 44504 *siginterrupt()* function.

44505 **RATIONALE**

44506 None.

44507 **FUTURE DIRECTIONS**

44508 None.

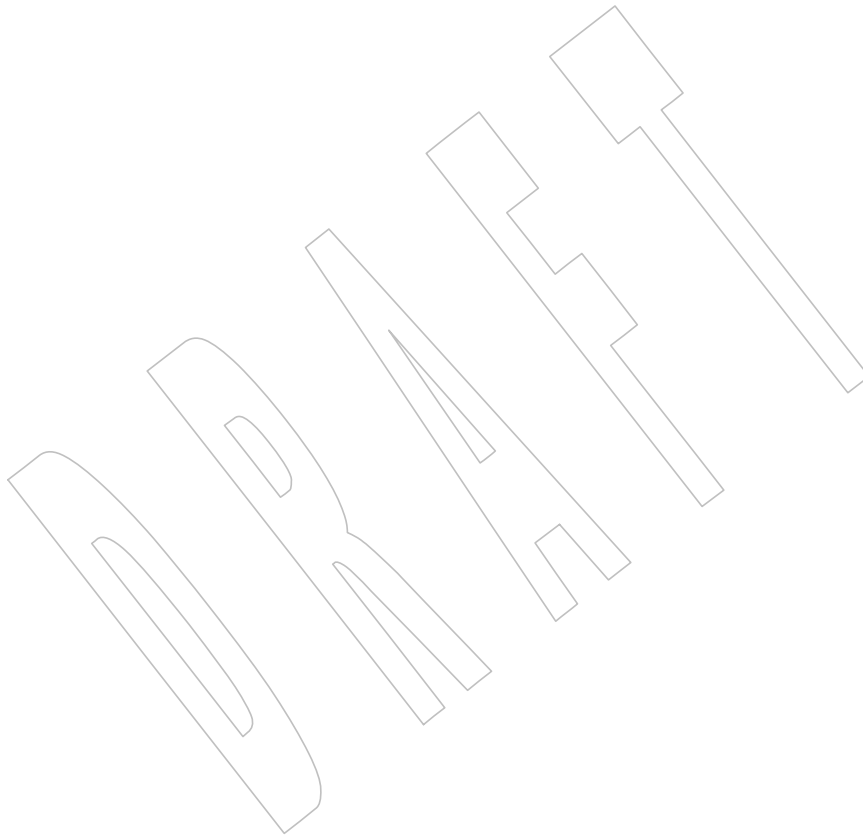
44509 **SEE ALSO**

44510 Section 2.4 (on page 28), *sigaction()*, the Base Definitions volume of IEEE Std 1003.1-200x,
 44511 <signal.h>

44512 **CHANGE HISTORY**

44513 First released in Issue 4, Version 2.

- 44514 **Issue 5**
44515 Moved from X/OPEN UNIX extension to BASE.
- 44516 **Issue 6**
44517 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/59 is applied, correcting the declaration in
44518 the sample implementation given in the DESCRIPTION.
- 44519 **Issue 7**
44520 The *siginterrupt()* function is marked obsolescent.



44521 **NAME**
 44522 sigismember — test for a signal in a signal set

44523 **SYNOPSIS**

44524 CX `#include <signal.h>`
 44525 `int sigismember(const sigset_t *set, int signo);`

44526 **DESCRIPTION**

44527 The *sigismember()* function shall test whether the signal specified by *signo* is a member of the set
 44528 pointed to by *set*.

44529 Applications should call either *sigemptyset()* or *sigfillset()* at least once for each object of type
 44530 **sigset_t** prior to any other use of that object. If such an object is not initialized in this way, but is
 44531 nonetheless supplied as an argument to any of *pthread_sigmask()*, *sigaction()*, *sigaddset()*,
 44532 *sigdelset()*, *sigismember()*, *sigpending()*, *sigprocmask()*, *sigsuspend()*, *sigtimedwait()*, *sigwait()*, or
 44533 *sigwaitinfo()*, the results are undefined.

44534 **RETURN VALUE**

44535 Upon successful completion, *sigismember()* shall return 1 if the specified signal is a member of
 44536 the specified set, or 0 if it is not. Otherwise, it shall return -1 and set *errno* to indicate the error.

44537 **ERRORS**

44538 The *sigismember()* function may fail if:

44539 [EINVAL] The *signo* argument is not a valid signal number, or is an unsupported signal
 44540 number.

44541 **EXAMPLES**

44542 None.

44543 **APPLICATION USAGE**

44544 None.

44545 **RATIONALE**

44546 None.

44547 **FUTURE DIRECTIONS**

44548 None.

44549 **SEE ALSO**

44550 [Section 2.4](#) (on page 28), *sigaction()*, *sigaddset()*, *sigdelset()*, *sigfillset()*, *sigemptyset()*, *sigpending()*,
 44551 *sigprocmask()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

44552 **CHANGE HISTORY**

44553 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

44554 **Issue 5**

44555 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
 44556 previous issues.

44557 **Issue 6**

44558 The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an
 44559 extension over the ISO C standard.

44560 **NAME**
 44561 siglongjmp — non-local goto with signal handling

44562 **SYNOPSIS**

44563 CX `#include <setjmp.h>`
 44564 `void siglongjmp(sigjmp_buf env, int val);`

44565 **DESCRIPTION**

44566 The *siglongjmp()* function shall be equivalent to the *longjmp()* function, except as follows:

- 44567 • References to *setjmp()* shall be equivalent to *sigsetjmp()*.
- 44568 • The *siglongjmp()* function shall restore the saved signal mask if and only if the *env*
 44569 argument was initialized by a call to *sigsetjmp()* with a non-zero *savemask* argument.

44570 **RETURN VALUE**

44571 After *siglongjmp()* is completed, program execution shall continue as if the corresponding
 44572 invocation of *sigsetjmp()* had just returned the value specified by *val*. The *siglongjmp()* function
 44573 shall not cause *sigsetjmp()* to return 0; if *val* is 0, *sigsetjmp()* shall return the value 1.

44574 **ERRORS**

44575 No errors are defined.

44576 **EXAMPLES**

44577 None.

44578 **APPLICATION USAGE**

44579 The distinction between *setjmp()* or *longjmp()* and *sigsetjmp()* or *siglongjmp()* is only significant
 44580 for programs which use *sigaction()*, *sigprocmask()*, or *sigsuspend()*.

44581 **RATIONALE**

44582 None.

44583 **FUTURE DIRECTIONS**

44584 None.

44585 **SEE ALSO**

44586 *longjmp()*, *setjmp()*, *sigprocmask()*, *sigsetjmp()*, *sigsuspend()*, the Base Definitions volume of
 44587 IEEE Std 1003.1-200x, **<setjmp.h>**

44588 **CHANGE HISTORY**

44589 First released in Issue 3. Included for alignment with the ISO POSIX-1 standard.

44590 **Issue 5**

44591 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

44592 **Issue 6**

44593 The DESCRIPTION is rewritten in terms of *longjmp()*.

44594 The SYNOPSIS is marked CX since the presence of this function in the **<setjmp.h>** header is an
 44595 extension over the ISO C standard.

44596 **NAME**44597 `signal` — signal management44598 **SYNOPSIS**44599 `#include <signal.h>`44600 `void (*signal(int sig, void (*func)(int)))(int);`44601 **DESCRIPTION**44602 CX The functionality described on this reference page is aligned with the ISO C standard. Any
44603 conflict between the requirements described here and the ISO C standard is unintentional. This
44604 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

44605 Use of this function is unspecified in a multi-threaded process.

44606 The `signal()` function chooses one of three ways in which receipt of the signal number `sig` is to be
44607 subsequently handled. If the value of `func` is `SIG_DFL`, default handling for that signal shall
44608 occur. If the value of `func` is `SIG_IGN`, the signal shall be ignored. Otherwise, the application
44609 shall ensure that `func` points to a function to be called when that signal occurs. An invocation of
44610 such a function because of a signal, or (recursively) of any further functions called by that
44611 invocation (other than functions in the standard library), is called a “signal handler”.44612 When a signal occurs, and `func` points to a function, it is implementation-defined whether the
44613 equivalent of a:44614 `signal(sig, SIG_DFL);`44615 is executed or the implementation prevents some implementation-defined set of signals (at least
44616 including `sig`) from occurring until the current signal handling has completed. (If the value of `sig`
44617 is `SIGILL`, the implementation may alternatively define that no action is taken.) Next the
44618 equivalent of:44619 `(*func)(sig);`44620 is executed. If and when the function returns, if the value of `sig` was `SIGFPE`, `SIGILL`, or
44621 `SIGSEGV` or any other implementation-defined value corresponding to a computational
44622 exception, the behavior is undefined. Otherwise, the program shall resume execution at the
44623 point it was interrupted. If the signal occurs as the result of calling the `abort()`, `raise()`, `kill()`,
44624 `pthread_kill()`, or `sigqueue()` function, the signal handler shall not call the `raise()` function.44625 CX If the signal occurs other than as the result of calling `abort()`, `raise()`, `kill()`, `pthread_kill()`, or
44626 `sigqueue()`, the behavior is undefined if the signal handler refers to any object with static storage
44627 duration other than by assigning a value to an object declared as volatile `sig_atomic_t`, or if the
44628 signal handler calls any function in the standard library other than one of the functions listed in
44629 [Section 2.4](#) (on page 28). Furthermore, if such a call fails, the value of `errno` is unspecified.

44630 At program start-up, the equivalent of:

44631 `signal(sig, SIG_IGN);`

44632 is executed for some signals, and the equivalent of:

44633 `signal(sig, SIG_DFL);`44634 CX is executed for all other signals (see [exec](#)).

44635 **RETURN VALUE**

44636 If the request can be honored, *signal()* shall return the value of *func* for the most recent call to
 44637 *signal()* for the specified signal *sig*. Otherwise, SIG_ERR shall be returned and a positive value
 44638 shall be stored in *errno*.

44639 **ERRORS**

44640 The *signal()* function shall fail if:

44641 CX [EINVAL] The *sig* argument is not a valid signal number or an attempt is made to catch a
 44642 signal that cannot be caught or ignore a signal that cannot be ignored.

44643 The *signal()* function may fail if:

44644 CX [EINVAL] An attempt was made to set the action to SIG_DFL for a signal that cannot be
 44645 caught or ignored (or both).

44646 **EXAMPLES**

44647 None.

44648 **APPLICATION USAGE**

44649 The *sigaction()* function provides a more comprehensive and reliable mechanism for controlling
 44650 signals; new applications should use *sigaction()* rather than *signal()*.

44651 **RATIONALE**

44652 None.

44653 **FUTURE DIRECTIONS**

44654 None.

44655 **SEE ALSO**

44656 Section 2.4 (on page 28), *exec*, *pause()*, *sigaction()*, *sigsuspend()*, *waitid()*, the Base Definitions
 44657 volume of IEEE Std 1003.1-200x, <signal.h>

44658 **CHANGE HISTORY**

44659 First released in Issue 1. Derived from Issue 1 of the SVID.

44660 **Issue 5**

44661 Moved from X/OPEN UNIX extension to BASE.

44662 The DESCRIPTION is updated to indicate that the *sigpause()* function restores the signal mask of
 44663 the process to its original state before returning.

44664 The RETURN VALUE section is updated to indicate that the *sigpause()* function suspends
 44665 execution of the process until a signal is received, whereupon it returns -1 and sets *errno* to
 44666 [EINTR].

44667 **Issue 6**

44668 Extensions beyond the ISO C standard are marked.

44669 The normative text is updated to avoid use of the term “must” for application requirements.

44670 The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.

44671 References to the *wait3()* function are removed.

44672 The *sighold()*, *sigignore()*, *sigrelse()*, and *sigset()* functions are split out onto their own reference
 44673 page.

44674 **NAME**44675 `signbit` — test sign44676 **SYNOPSIS**44677 `#include <math.h>`44678 `int signbit(real-floating x);`44679 **DESCRIPTION**44680 CX The functionality described on this reference page is aligned with the ISO C standard. Any
44681 conflict between the requirements described here and the ISO C standard is unintentional. This
44682 volume of IEEE Std 1003.1-200x defers to the ISO C standard.44683 The *signbit()* macro shall determine whether the sign of its argument value is negative. NaNs,
44684 zeros, and infinities have a sign bit.44685 **RETURN VALUE**44686 The *signbit()* macro shall return a non-zero value if and only if the sign of its argument value is
44687 negative.44688 **ERRORS**

44689 No errors are defined.

44690 **EXAMPLES**

44691 None.

44692 **APPLICATION USAGE**

44693 None.

44694 **RATIONALE**

44695 None.

44696 **FUTURE DIRECTIONS**

44697 None.

44698 **SEE ALSO**44699 *fpclassify()*, *isfinite()*, *isinf()*, *isnan()*, *isnormal()*, the Base Definitions volume of
44700 IEEE Std 1003.1-200x, `<math.h>`44701 **CHANGE HISTORY**

44702 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

44703 **NAME**
44704 sigpause — remove a signal from the signal mask and suspend the thread

44705 **SYNOPSIS**

44706 OB XSI #include <signal.h>
44707 int sigpause(int sig);

44708 **DESCRIPTION**

44709 Refer to *sighold()*.

44710 **NAME**
 44711 sigpending — examine pending signals

44712 **SYNOPSIS**

44713 CX #include <signal.h>
 44714 int sigpending(sigset_t *set);

44715 **DESCRIPTION**

44716 The *sigpending()* function shall store, in the location referenced by the *set* argument, the set of
 44717 signals that are blocked from delivery to the calling thread and that are pending on the process
 44718 or the calling thread.

44719 **RETURN VALUE**

44720 Upon successful completion, *sigpending()* shall return 0; otherwise, -1 shall be returned and
 44721 *errno* set to indicate the error.

44722 **ERRORS**

44723 No errors are defined.

44724 **EXAMPLES**

44725 None.

44726 **APPLICATION USAGE**

44727 None.

44728 **RATIONALE**

44729 None.

44730 **FUTURE DIRECTIONS**

44731 None.

44732 **SEE ALSO**

44733 *exec*, *sigaddset()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*, *sigprocmask()*, the Base
 44734 Definitions volume of IEEE Std 1003.1-200x, <signal.h>

44735 **CHANGE HISTORY**

44736 First released in Issue 3.

44737 **Issue 5**

44738 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

44739 **Issue 6**

44740 The SYNOPSIS is marked CX since the presence of this function in the <signal.h> header is an
 44741 extension over the ISO C standard.

44742 **NAME**
44743 sigprocmask — examine and change blocked signals

44744 **SYNOPSIS**

```
44745 CX #include <signal.h>  
44746 int sigprocmask(int how, const sigset_t *restrict set,  
44747 sigset_t *restrict oset);
```

44748 **DESCRIPTION**

44749 Refer to [pthread_sigmask\(\)](#).

44750 **NAME**
 44751 sigqueue — queue a signal to a process

44752 **SYNOPSIS**

44753 CX `#include <signal.h>`
 44754 `int sigqueue(pid_t pid, int signo, const union signal value);`

44755 **DESCRIPTION**

44756 The *sigqueue()* function shall cause the signal specified by *signo* to be sent with the value
 44757 specified by *value* to the process specified by *pid*. If *signo* is zero (the null signal), error checking
 44758 is performed but no signal is actually sent. The null signal can be used to check the validity of
 44759 *pid*.

44760 The conditions required for a process to have permission to queue a signal to another process are
 44761 the same as for the *kill()* function.

44762 The *sigqueue()* function shall return immediately. If SA_SIGINFO is set for *signo* and if the
 44763 resources were available to queue the signal, the signal shall be queued and sent to the receiving
 44764 process. If SA_SIGINFO is not set for *signo*, then *signo* shall be sent at least once to the receiving
 44765 process; it is unspecified whether *value* shall be sent to the receiving process as a result of this
 44766 call.

44767 If the value of *pid* causes *signo* to be generated for the sending process, and if *signo* is not blocked
 44768 for the calling thread and if no other thread has *signo* unblocked or is waiting in a *sigwait()*
 44769 function for *signo*, either *signo* or at least the pending, unblocked signal shall be delivered to the
 44770 calling thread before the *sigqueue()* function returns. Should any multiple pending signals in the
 44771 range SIGRTMIN to SIGRTMAX be selected for delivery, it shall be the lowest numbered one.
 44772 The selection order between realtime and non-realtime signals, or between multiple pending
 44773 non-realtime signals, is unspecified.

44774 **RETURN VALUE**

44775 Upon successful completion, the specified signal shall have been queued, and the *sigqueue()*
 44776 function shall return a value of zero. Otherwise, the function shall return a value of -1 and set
 44777 *errno* to indicate the error.

44778 **ERRORS**

44779 The *sigqueue()* function shall fail if:

- | | | |
|-------|----------|---|
| 44780 | [EAGAIN] | No resources are available to queue the signal. The process has already queued {SIGQUEUE_MAX} signals that are still pending at the receiver(s), or a system-wide resource limit has been exceeded. |
| 44781 | | |
| 44782 | | |
| 44783 | [EINVAL] | The value of the <i>signo</i> argument is an invalid or unsupported signal number. |
| 44784 | [EPERM] | The process does not have the appropriate privilege to send the signal to the receiving process. |
| 44785 | | |
| 44786 | [ESRCH] | The process <i>pid</i> does not exist. |

44787

EXAMPLES

44788

None.

44789

APPLICATION USAGE

44790

None.

44791

RATIONALE

44792

The *sigqueue()* function allows an application to queue a realtime signal to itself or to another process, specifying the application-defined value. This is common practice in realtime applications on existing realtime systems. It was felt that specifying another function in the *sig...* name space already carved out for signals was preferable to extending the interface to *kill()*.

44793

44794

44795

44796

44797

Such a function became necessary when the *put/get* event function of the message queues was removed. It should be noted that the *sigqueue()* function implies reduced performance in a security-conscious implementation as the access permissions between the sender and receiver have to be checked on each send when the *pid* is resolved into a target process. Such access checks were necessary only at message queue open in the previous interface.

44798

44799

44800

44801

44802

The standard developers required that *sigqueue()* have the same semantics with respect to the null signal as *kill()*, and that the same permission checking be used. But because of the difficulty of implementing the “broadcast” semantic of *kill()* (for example, to process groups) and the interaction with resource allocation, this semantic was not adopted. The *sigqueue()* function queues a signal to a single process specified by the *pid* argument.

44803

44804

44805

44806

44807

The *sigqueue()* function can fail if the system has insufficient resources to queue the signal. An explicit limit on the number of queued signals that a process could send was introduced. While the limit is “per-sender”, this volume of IEEE Std 1003.1-200x does not specify that the resources be part of the state of the sender. This would require either that the sender be maintained after exit until all signals that it had sent to other processes were handled or that all such signals that had not yet been acted upon be removed from the queue(s) of the receivers. This volume of IEEE Std 1003.1-200x does not preclude this behavior, but an implementation that allocated queuing resources from a system-wide pool (with per-sender limits) and that leaves queued signals pending after the sender exits is also permitted.

44808

44809

44810

44811

44812

44813

44814

44815

44816

FUTURE DIRECTIONS

44817

None.

44818

SEE ALSO

44819

[Section 2.8.1](#) (on page 41), the Base Definitions volume of IEEE Std 1003.1-200x, `<signal.h>`

44820

CHANGE HISTORY

44821

First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

44822

44823

Issue 6

44824

The *sigqueue()* function is marked as part of the Realtime Signals Extension option.

44825

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Realtime Signals Extension option.

44826

44827

Issue 7

44828

The *sigqueue()* function is moved from the Realtime Signals Extension option to the Base.

44829 **NAME**
44830 sigrelse, sigset — signal management

44831 **SYNOPSIS**

44832 OB XSI #include <signal.h>
44833 int sigrelse(int *sig*);
44834 void (*sigset(int *sig*, void (**disp*)(int)))(int);

44835 **DESCRIPTION**

44836 Refer to *sighold()*.

44837 **NAME**

44838 sigsetjmp — set jump point for a non-local goto

44839 **SYNOPSIS**

```
44840 CX #include <setjmp.h>
44841 int sigsetjmp(sigjmp_buf env, int savemask);
```

44842 **DESCRIPTION**44843 The *sigsetjmp()* function shall be equivalent to the *setjmp()* function, except as follows:

- 44844 • References to *setjmp()* are equivalent to *sigsetjmp()*.
- 44845 • References to *longjmp()* are equivalent to *siglongjmp()*.
- 44846 • If the value of the *savemask* argument is not 0, *sigsetjmp()* shall also save the current signal mask of the calling thread as part of the calling environment.

44848 **RETURN VALUE**

44849 If the return is from a successful direct invocation, *sigsetjmp()* shall return 0. If the return is from
 44850 a call to *siglongjmp()*, *sigsetjmp()* shall return a non-zero value.

44851 **ERRORS**

44852 No errors are defined.

44853 **EXAMPLES**

44854 None.

44855 **APPLICATION USAGE**

44856 The distinction between *setjmp()/longjmp()* and *sigsetjmp()/siglongjmp()* is only significant for
 44857 programs which use *sigaction()*, *sigprocmask()*, or *sigsuspend()*.

44858 Note that since this function is defined in terms of *setjmp()*, if *savemask* is zero, it is unspecified
 44859 whether the signal mask is saved.

44860 **RATIONALE**

44861 The ISO C standard specifies various restrictions on the usage of the *setjmp()* macro in order to
 44862 permit implementors to recognize the name in the compiler and not implement an actual
 44863 function. These same restrictions apply to the *sigsetjmp()* macro.

44864 There are processors that cannot easily support these calls, but this was not considered a
 44865 sufficient reason to exclude them.

44866 4.2 BSD, 4.3 BSD, and XSI-conformant systems provide functions named *_setjmp()* and
 44867 *_longjmp()* that, together with *setjmp()* and *longjmp()*, provide the same functionality as
 44868 *sigsetjmp()* and *siglongjmp()*. On those systems, *setjmp()* and *longjmp()* save and restore signal
 44869 masks, while *_setjmp()* and *_longjmp()* do not. On System V Release 3 and in corresponding
 44870 issues of the SVID, *setjmp()* and *longjmp()* are explicitly defined not to save and restore signal
 44871 masks. In order to permit existing practice in both cases, the relation of *setjmp()* and *longjmp()* to
 44872 signal masks is not specified, and a new set of functions is defined instead.

44873 The *longjmp()* and *siglongjmp()* functions operate as in the previous issue provided the matching
 44874 *setjmp()* or *sigsetjmp()* has been performed in the same thread. Non-local jumps into contexts
 44875 saved by other threads would be at best a questionable practice and were not considered worthy
 44876 of standardization.

44877

FUTURE DIRECTIONS

44878

None.

44879

SEE ALSO

44880

siglongjmp(), *signal()*, *sigprocmask()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<setjmp.h>**

44881

44882

CHANGE HISTORY

44883

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

44884

Issue 5

44885

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

44886

Issue 6

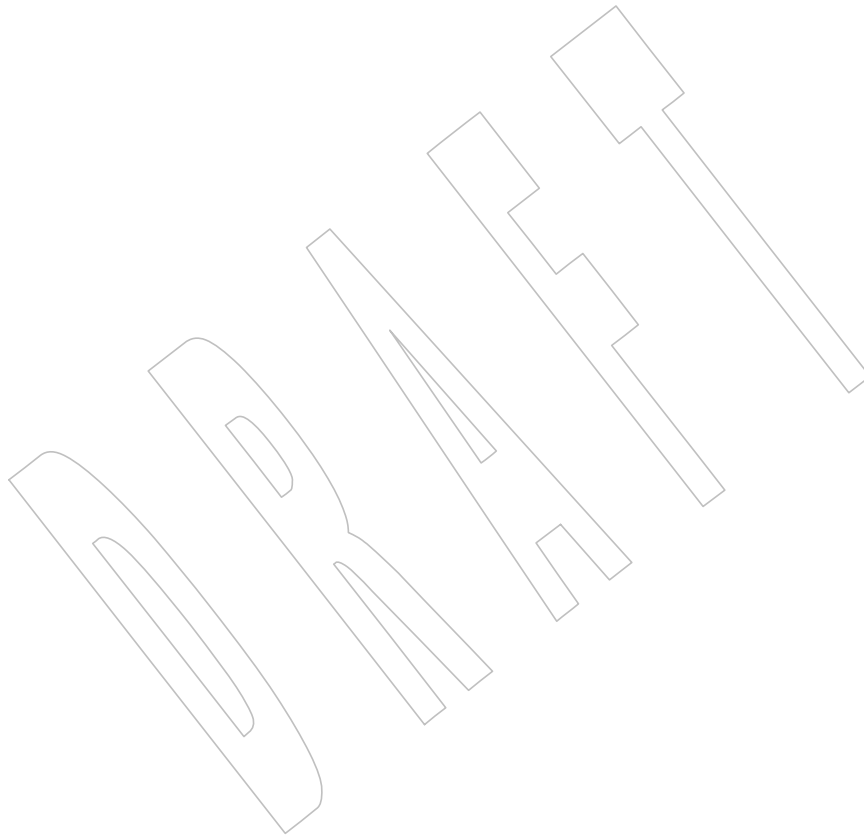
44887

The DESCRIPTION is reworded in terms of *setjmp()*.

44888

The SYNOPSIS is marked CX since the presence of this function in the **<setjmp.h>** header is an extension over the ISO C standard.

44889



44890 **NAME**

44891 sigsuspend — wait for a signal

44892 **SYNOPSIS**

```
44893 CX #include <signal.h>
44894 int sigsuspend(const sigset_t *sigmask);
```

44895 **DESCRIPTION**

44896 The *sigsuspend()* function shall replace the current signal mask of the calling thread with the set
 44897 of signals pointed to by *sigmask* and then suspend the thread until delivery of a signal whose
 44898 action is either to execute a signal-catching function or to terminate the process. This shall not
 44899 cause any other signals that may have been pending on the process to become pending on the
 44900 thread.

44901 If the action is to terminate the process then *sigsuspend()* shall never return. If the action is to
 44902 execute a signal-catching function, then *sigsuspend()* shall return after the signal-catching
 44903 function returns, with the signal mask restored to the set that existed prior to the *sigsuspend()*
 44904 call.

44905 It is not possible to block signals that cannot be ignored. This is enforced by the system without
 44906 causing an error to be indicated.

44907 **RETURN VALUE**

44908 Since *sigsuspend()* suspends thread execution indefinitely, there is no successful completion
 44909 return value. If a return occurs, *-1* shall be returned and *errno* set to indicate the error.

44910 **ERRORS**

44911 The *sigsuspend()* function shall fail if:

44912 [EINTR] A signal is caught by the calling process and control is returned from the
 44913 signal-catching function.

44914 **EXAMPLES**

44915 None.

44916 **APPLICATION USAGE**

44917 Normally, at the beginning of a critical code section, a specified set of signals is blocked using
 44918 the *sigprocmask()* function. When the thread has completed the critical section and needs to wait
 44919 for the previously blocked signal(s), it pauses by calling *sigsuspend()* with the mask that was
 44920 returned by the *sigprocmask()* call.

44921 **RATIONALE**

44922 Code which wants to avoid the ambiguity of the signal mask for thread cancellation handlers
 44923 can install an additional cancellation handler which resets the signal mask to the expected value.

```
44924 void cleanup(void *arg)
44925 {
44926     sigset_t *ss = (sigset_t *) arg;
44927     pthread_sigmask(SIG_SETMASK, ss, NULL);
44928 }
44929 int call_sigsuspend(const sigset_t *mask)
44930 {
44931     sigset_t oldmask;
44932     int result;
44933     pthread_sigmask(SIG_SETMASK, NULL, &oldmask);
```



```
44934         pthread_cleanup_push(cleanup, &oldmask);
44935         result = sigsuspend(sigmask);
44936         pthread_cleanup_pop(0);
44937         return result;
44938     }
```

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 28), *pause()*, *sigaction()*, *sigaddset()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The text in the RETURN VALUE section has been changed from “suspends process execution” to “suspends thread execution”. This reflects IEEE PASC Interpretation 1003.1c #40.

Text in the APPLICATION USAGE section has been replaced.

The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an extension over the ISO C standard.

Issue 7

SD5-XSH-ERN-122 is applied, adding the example code in the RATIONALE.

44956 **NAME**

44957 sigtimedwait, sigwaitinfo — wait for queued signals

44958 **SYNOPSIS**

```

44959 CX      #include <signal.h>
44960
44961      int sigtimedwait(const sigset_t *restrict set,
44962                    siginfo_t *restrict info,
44963                    const struct timespec *restrict timeout);
44963      int sigwaitinfo(const sigset_t *restrict set,
44964                    siginfo_t *restrict info);

```

44965 **DESCRIPTION**

44966 The *sigtimedwait()* function shall be equivalent to *sigwaitinfo()* except that if none of the signals
44967 specified by *set* are pending, *sigtimedwait()* shall wait for the time interval specified in the
44968 **timespec** structure referenced by *timeout*. If the **timespec** structure pointed to by *timeout* is zero-
44969 valued and if none of the signals specified by *set* are pending, then *sigtimedwait()* shall return
44970 immediately with an error. If *timeout* is the NULL pointer, the behavior is unspecified. If the
44971 Monotonic Clock option is supported, the CLOCK_MONOTONIC clock shall be used to
44972 measure the time interval specified by the *timeout* argument.

44973 The *sigwaitinfo()* function selects the pending signal from the set specified by *set*. Should any of
44974 multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it shall be the
44975 lowest numbered one. The selection order between realtime and non-realtime signals, or
44976 between multiple pending non-realtime signals, is unspecified. If no signal in *set* is pending at
44977 the time of the call, the calling thread shall be suspended until one or more signals in *set* become
44978 pending or until it is interrupted by an unblocked, caught signal.

44979 The *sigwaitinfo()* function shall be equivalent to the *sigwait()* function if the *info* argument is
44980 NULL. If the *info* argument is non-NULL, the *sigwaitinfo()* function shall be equivalent to
44981 *sigwait()*, except that the selected signal number shall be stored in the *si_signo* member, and the
44982 cause of the signal shall be stored in the *si_code* member. If any value is queued to the selected
44983 signal, the first such queued value shall be dequeued and, if the *info* argument is non-NULL, the
44984 value shall be stored in the *si_value* member of *info*. The system resource used to queue the
44985 signal shall be released and returned to the system for other use. If no value is queued, the
44986 content of the *si_value* member is undefined. If no further signals are queued for the selected
44987 signal, the pending indication for that signal shall be reset.

44988 **RETURN VALUE**

44989 Upon successful completion (that is, one of the signals specified by *set* is pending or is
44990 generated) *sigwaitinfo()* and *sigtimedwait()* shall return the selected signal number. Otherwise,
44991 the function shall return a value of -1 and set *errno* to indicate the error.

44992 **ERRORS**

44993 The *sigtimedwait()* function shall fail if:

44994 [EAGAIN] No signal specified by *set* was generated within the specified timeout period.

44995 The *sigtimedwait()* and *sigwaitinfo()* functions may fail if:

44996 [EINTR] The wait was interrupted by an unblocked, caught signal. It shall be
44997 documented in system documentation whether this error causes these
44998 functions to fail.

44999
45000
45001
45002
45003
45004
45005
45006
45007
45008
45009
45010
45011
45012
45013
45014
45015
45016
45017
45018
45019
45020
45021
45022
45023
45024
45025
45026
45027
45028
45029
45030
45031
45032
45033
45034
45035
45036
45037
45038
45039
45040
45041
45042
45043
45044
45045
45046
45047

The *sigtimedwait()* function may also fail if:

[EINVAL] The *timeout* argument specified a *tv_nsec* value less than zero or greater than or equal to 1 000 million.

An implementation should only check for this error if no signal is pending in *set* and it is necessary to wait.

EXAMPLES

None.

APPLICATION USAGE

The *sigtimedwait()* function times out and returns an [EAGAIN] error. Application writers should note that this is inconsistent with other functions such as *pthread_cond_timedwait()* that return [ETIMEDOUT].

RATIONALE

Existing programming practice on realtime systems uses the ability to pause waiting for a selected set of events and handle the first event that occurs in-line instead of in a signal-handling function. This allows applications to be written in an event-directed style similar to a state machine. This style of programming is useful for largescale transaction processing in which the overall throughput of an application and the ability to clearly track states are more important than the ability to minimize the response time of individual event handling.

It is possible to construct a signal-waiting macro function out of the realtime signal function mechanism defined in this volume of IEEE Std 1003.1-200x. However, such a macro has to include the definition of a generalized handler for all signals to be waited on. A significant portion of the overhead of handler processing can be avoided if the signal-waiting function is provided by the kernel. This volume of IEEE Std 1003.1-200x therefore provides two signal-waiting functions—one that waits indefinitely and one with a timeout—as part of the overall realtime signal function specification.

The specification of a function with a timeout allows an application to be written that can be broken out of a wait after a set period of time if no event has occurred. It was argued that setting a timer event before the wait and recognizing the timer event in the wait would also implement the same functionality, but at a lower performance level. Because of the performance degradation associated with the user-level specification of a timer event and the subsequent cancellation of that timer event after the wait completes for a valid event, and the complexity associated with handling potential race conditions associated with the user-level method, the separate function has been included.

Note that the semantics of the *sigwaitinfo()* function are nearly identical to that of the *sigwait()* function defined by this volume of IEEE Std 1003.1-200x. The only difference is that *sigwaitinfo()* returns the queued signal value in the *value* argument. The return of the queued value is required so that applications can differentiate between multiple events queued to the same signal number.

The two distinct functions are being maintained because some implementations may choose to implement the POSIX Threads Extension functions and not implement the queued signals extensions. Note, though, that *sigwaitinfo()* does not return the queued value if the *value* argument is NULL, so the POSIX Threads Extension *sigwait()* function can be implemented as a macro on *sigwaitinfo()*.

The *sigtimedwait()* function was separated from the *sigwaitinfo()* function to address concerns regarding the overloading of the *timeout* pointer to indicate indefinite wait (no timeout), timed wait, and immediate return, and concerns regarding consistency with other functions where the conditional and timed waits were separate functions from the pure blocking function. The semantics of *sigtimedwait()* are specified such that *sigwaitinfo()* could be implemented as a macro with a NULL pointer for *timeout*.

45048 The *sigwait* functions provide a synchronous mechanism for threads to wait for asynchronously-
 45049 generated signals. One important question was how many threads that are suspended in a call
 45050 to a *sigwait()* function for a signal should return from the call when the signal is sent. Four
 45051 choices were considered:

- 45052 1. Return an error for multiple simultaneous calls to *sigwait* functions for the same signal.
- 45053 2. One or more threads return.
- 45054 3. All waiting threads return.
- 45055 4. Exactly one thread returns.

45056 Prohibiting multiple calls to *sigwait()* for the same signal was felt to be overly restrictive. The
 45057 “one or more” behavior made implementation of conforming packages easy at the expense of
 45058 forcing POSIX threads clients to protect against multiple simultaneous calls to *sigwait()* in
 45059 application code in order to achieve predictable behavior. There was concern that the “all
 45060 waiting threads” behavior would result in “signal broadcast storms”, consuming excessive CPU
 45061 resources by replicating the signals in the general case. Furthermore, no convincing examples
 45062 could be presented that delivery to all was either simpler or more powerful than delivery to one.

45063 Thus, the consensus was that exactly one thread that was suspended in a call to a *sigwait*
 45064 function for a signal should return when that signal occurs. This is not an onerous restriction as:

- 45065 • A multi-way signal wait can be built from the single-way wait.
- 45066 • Signals should only be handled by application-level code, as library routines cannot guess
 45067 what the application wants to do with signals generated for the entire process.
- 45068 • Applications can thus arrange for a single thread to wait for any given signal and call any
 45069 needed routines upon its arrival.

45070 In an application that is using signals for interprocess communication, signal processing is
 45071 typically done in one place. Alternatively, if the signal is being caught so that process cleanup
 45072 can be done, the signal handler thread can call separate process cleanup routines for each
 45073 portion of the application. Since the application main line started each portion of the application,
 45074 it is at the right abstraction level to tell each portion of the application to clean up.

45075 Certainly, there exist programming styles where it is logical to consider waiting for a single
 45076 signal in multiple threads. A simple *sigwait_multiple()* routine can be constructed to achieve this
 45077 goal. A possible implementation would be to have each *sigwait_multiple()* caller registered as
 45078 having expressed interest in a set of signals. The caller then waits on a thread-specific condition
 45079 variable. A single server thread calls a *sigwait()* function on the union of all registered signals.
 45080 When the *sigwait()* function returns, the appropriate state is set and condition variables are
 45081 broadcast. New *sigwait_multiple()* callers may cause the pending *sigwait()* call to be canceled
 45082 and reissued in order to update the set of signals being waited for.

45083 FUTURE DIRECTIONS

45084 None.

45085 SEE ALSO

45086 Section 2.8.1 (on page 41), *pause()*, *pthread_sigmask()*, *sigaction()*, *sigpending()*, *sigsuspend()*,
 45087 *sigwait()*, the Base Definitions volume of IEEE Std 1003.1-200x, <signal.h>, <time.h>

45088 CHANGE HISTORY

45089 First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the
 45090 POSIX Threads Extension.

Issue 645091
45092
45093
45094
45095
45096
45097
45098
45099
45100
45101
45102
45103
45104
45105
45106

These functions are marked as part of the Realtime Signals Extension option.

The Open Group Corrigendum U035/3 is applied. The SYNOPSIS of the *sigwaitinfo()* function has been corrected so that the second argument is of type **siginfo_t** *.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Realtime Signals Extension option.

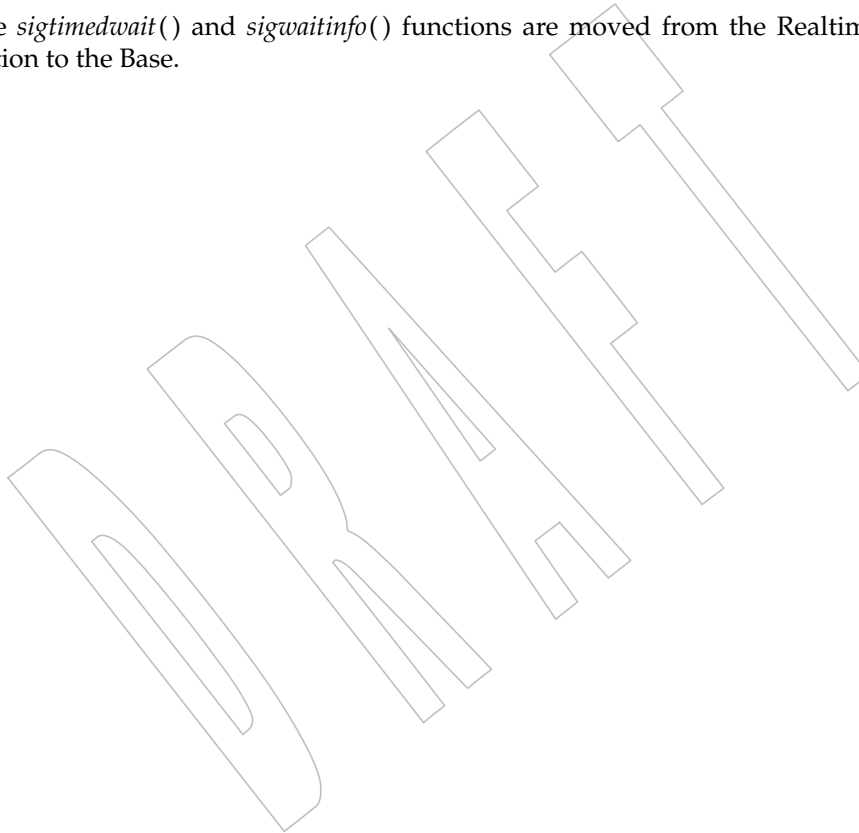
The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that the CLOCK_MONOTONIC clock, if supported, is used to measure timeout intervals.

The **restrict** keyword is added to the *sigtimedwait()* and *sigwaitinfo()* prototypes for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/130 is applied, restoring wording in the RETURN VALUE section to that in the original base document (“An implementation should only check for this error if no signal is pending in *set* and it is necessary to wait”).

Issue 7

The *sigtimedwait()* and *sigwaitinfo()* functions are moved from the Realtime Signals Extension option to the Base.



45107 **NAME**

45108 sigwait — wait for queued signals

45109 **SYNOPSIS**

```
45110 CX #include <signal.h>
45111 int sigwait(const sigset_t *restrict set, int *restrict sig);
```

45112 **DESCRIPTION**

45113 The *sigwait()* function shall select a pending signal from *set*, atomically clear it from the system's
 45114 set of pending signals, and return that signal number in the location referenced by *sig*. If prior to
 45115 the call to *sigwait()* there are multiple pending instances of a single signal number, it is
 45116 implementation-defined whether upon successful return there are any remaining pending
 45117 signals for that signal number. If the implementation supports queued signals and there are
 45118 multiple signals queued for the signal number selected, the first such queued signal shall cause a
 45119 return from *sigwait()* and the remainder shall remain queued. If no signal in *set* is pending at the
 45120 time of the call, the thread shall be suspended until one or more becomes pending. The signals
 45121 defined by *set* shall have been blocked at the time of the call to *sigwait()*; otherwise, the behavior
 45122 is undefined. The effect of *sigwait()* on the signal actions for the signals in *set* is unspecified.

45123 If more than one thread is using *sigwait()* to wait for the same signal, no more than one of these
 45124 threads shall return from *sigwait()* with the signal number. If more than a single thread is
 45125 blocked in *sigwait()* for a signal when that signal is generated for the process, it is unspecified
 45126 which of the waiting threads returns from *sigwait()*. If the signal is generated for a specific
 45127 thread, as by *pthread_kill()*, only that thread shall return.

45128 Should any of the multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it
 45129 shall be the lowest numbered one. The selection order between realtime and non-realtime
 45130 signals, or between multiple pending non-realtime signals, is unspecified.

45131 **RETURN VALUE**

45132 Upon successful completion, *sigwait()* shall store the signal number of the received signal at the
 45133 location referenced by *sig* and return zero. Otherwise, an error number shall be returned to
 45134 indicate the error.

45135 **ERRORS**45136 The *sigwait()* function may fail if:45137 [EINVAL] The *set* argument contains an invalid or unsupported signal number.45138 **EXAMPLES**

45139 None.

45140 **APPLICATION USAGE**

45141 None.

45142 **RATIONALE**

45143 To provide a convenient way for a thread to wait for a signal, this volume of
 45144 IEEE Std 1003.1-200x provides the *sigwait()* function. For most cases where a thread has to wait
 45145 for a signal, the *sigwait()* function should be quite convenient, efficient, and adequate.

45146 However, requests were made for a lower-level primitive than *sigwait()* and for semaphores that
 45147 could be used by threads. After some consideration, threads were allowed to use semaphores
 45148 and *sem_post()* was defined to be async-signal and async-cancel-safe.

45149 In summary, when it is necessary for code run in response to an asynchronous signal to notify a
 45150 thread, *sigwait()* should be used to handle the signal. Alternatively, if the implementation

45151 provides semaphores, they also can be used, either following *sigwait()* or from within a signal
45152 handling routine previously registered with *sigaction()*.

FUTURE DIRECTIONS

45153 None.

SEE ALSO

45155 [Section 2.4](#) (on page 28), [Section 2.8.1](#) (on page 41), [pause\(\)](#), [pthread_sigmask\(\)](#), [sigaction\(\)](#),
45156 [sigpending\(\)](#), [sigsuspend\(\)](#), [sigwaitinfo\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x,
45157 [<signal.h>](#), [<time.h>](#)
45158

CHANGE HISTORY

45159 First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the
45160 POSIX Threads Extension.
45161

Issue 6

45162 The **restrict** keyword is added to the *sigwait()* prototype for alignment with the
45163 ISO/IEC 9899:1999 standard.
45164

45165 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/131 is applied, updating the
45166 DESCRIPTION section to state that if more than a single thread is blocked in *sigwait()*, it is
45167 unspecified which of the waiting threads returns, and that if a signal is generated for a specific
45168 thread only that thread shall return.

Issue 7

45169 Functionality relating to the Realtime Signals Extension option is moved to the Base.
45170

45171 **NAME**
45172 sigwaitinfo — wait for queued signals

45173 **SYNOPSIS**

```
45174 #include <signal.h>  
45175 int sigwaitinfo(const sigset_t *restrict set, siginfo_t *restrict info);
```

45176 **DESCRIPTION**

45177 Refer to [sigtimedwait\(\)](#).

45178 **NAME**
 45179 `sin, sinf, sinl` — sine function

45180 **SYNOPSIS**
 45181 `#include <math.h>`
 45182 `double sin(double x);`
 45183 `float sinf(float x);`
 45184 `long double sinl(long double x);`

45185 **DESCRIPTION**
 45186 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 45187 conflict between the requirements described here and the ISO C standard is unintentional. This
 45188 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

45189 These functions shall compute the sine of their argument x , measured in radians.

45190 An application wishing to check for error situations should set *errno* to zero and call
 45191 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 45192 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 45193 zero, an error has occurred.

45194 **RETURN VALUE**
 45195 Upon successful completion, these functions shall return the sine of x .

45196 MX If x is NaN, a NaN shall be returned.
 45197 If x is ± 0 , x shall be returned.
 45198 If x is subnormal, a range error may occur and x should be returned.
 45199 If x is $\pm\text{Inf}$, a domain error shall occur, and either a NaN (if supported), or an implementation-
 45200 defined value shall be returned.

45201 **ERRORS**
 45202 These functions shall fail if:

45203 MX **Domain Error** The x argument is $\pm\text{Inf}$.
 45204 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 45205 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 45206 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 45207 shall be raised.

45208 These functions may fail if:

45209 MX **Range Error** The value of x is subnormal
 45210 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 45211 then *errno* shall be set to [ERANGE]. If the integer expression
 45212 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 45213 floating-point exception shall be raised.

45214

EXAMPLES

45215

Taking the Sine of a 45-Degree Angle

45216

```
#include <math.h>
```

45217

```
...
```

45218

```
double radians = 45.0 * M_PI / 180;
```

45219

```
double result;
```

45220

```
...
```

45221

```
result = sin(radians);
```

45222

APPLICATION USAGE

45223

These functions may lose accuracy when their argument is near a multiple of π or is far from 0.0.

45224

On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

45225

45226

RATIONALE

45227

None.

45228

FUTURE DIRECTIONS

45229

None.

45230

SEE ALSO

45231

asin(), *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

45232

45233

CHANGE HISTORY

45234

First released in Issue 1. Derived from Issue 1 of the SVID.

45235

Issue 5

45236

The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes in previous issues.

45237

45238

Issue 6

45239

The *sinf()* and *sinl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

45240

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

45241

45242

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

45243

45244 **NAME**45245 `sinh, sinhf, sinh1` — hyperbolic sine functions45246 **SYNOPSIS**

```
45247 #include <math.h>
45248
45248 double sinh(double x);
45249 float  sinhf(float x);
45250 long double sinh1(long double x);
```

45251 **DESCRIPTION**

45252 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 45253 conflict between the requirements described here and the ISO C standard is unintentional. This
 45254 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

45255 These functions shall compute the hyperbolic sine of their argument x .

45256 An application wishing to check for error situations should set *errno* to zero and call
 45257 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 45258 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 45259 zero, an error has occurred.

45260 **RETURN VALUE**

45261 Upon successful completion, these functions shall return the hyperbolic sine of x .

45262 If the result would cause an overflow, a range error shall occur and \pm HUGE_VAL,
 45263 \pm HUGE_VALF, and \pm HUGE_VALL (with the same sign as x) shall be returned as appropriate for
 45264 the type of the function.

45265 MX If x is NaN, a NaN shall be returned.

45266 If x is ± 0 or \pm Inf, x shall be returned.

45267 If x is subnormal, a range error may occur and x should be returned.

45268 **ERRORS**

45269 These functions shall fail if:

45270 Range Error The result would cause an overflow.

45271 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 45272 then *errno* shall be set to [ERANGE]. If the integer expression
 45273 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
 45274 floating-point exception shall be raised.

45275 These functions may fail if:

45276 MX Range Error The value x is subnormal.

45277 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 45278 then *errno* shall be set to [ERANGE]. If the integer expression
 45279 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 45280 floating-point exception shall be raised.

45281

EXAMPLES

45282

None.

45283

APPLICATION USAGE

45284

On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

45285

45286

RATIONALE

45287

None.

45288

FUTURE DIRECTIONS

45289

None.

45290

SEE ALSO

45291

asinh(), *cosh()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, *tanh()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

45292

45293

45294

CHANGE HISTORY

45295

First released in Issue 1. Derived from Issue 1 of the SVID.

45296

Issue 5

45297

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

45298

45299

Issue 6

45300

The *sinhf()* and *sinhl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

45301

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

45302

45303

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

45304

45305

NAME

45306

sinl — sine function

45307

SYNOPSIS

45308

#include <math.h>

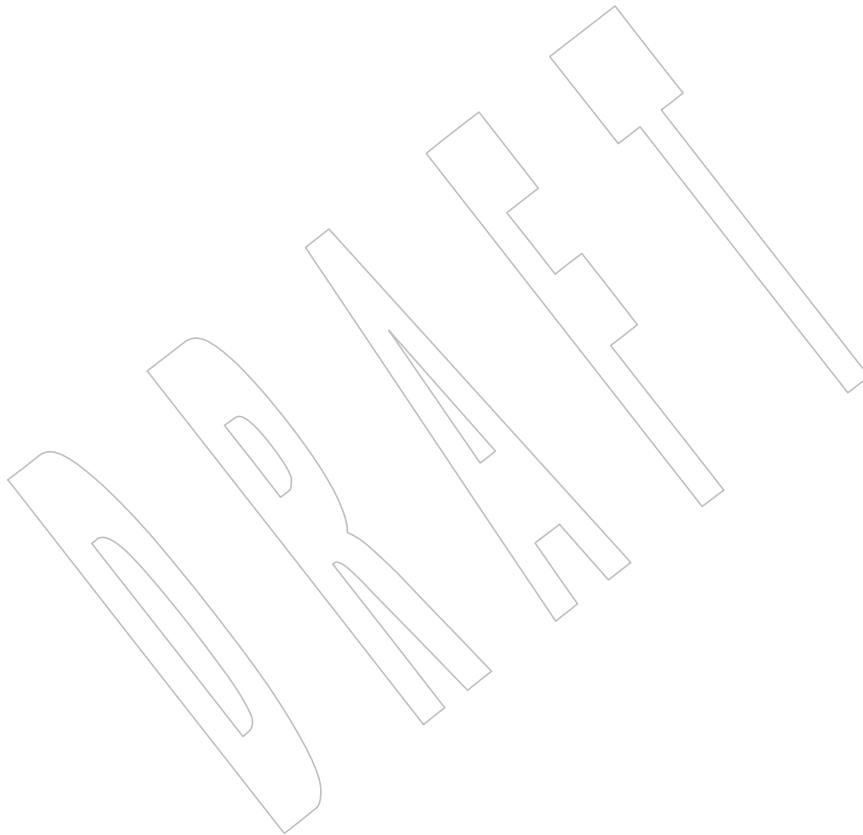
45309

long double sinl(long double x);

45310

DESCRIPTION

45311

Refer to *sin()*.

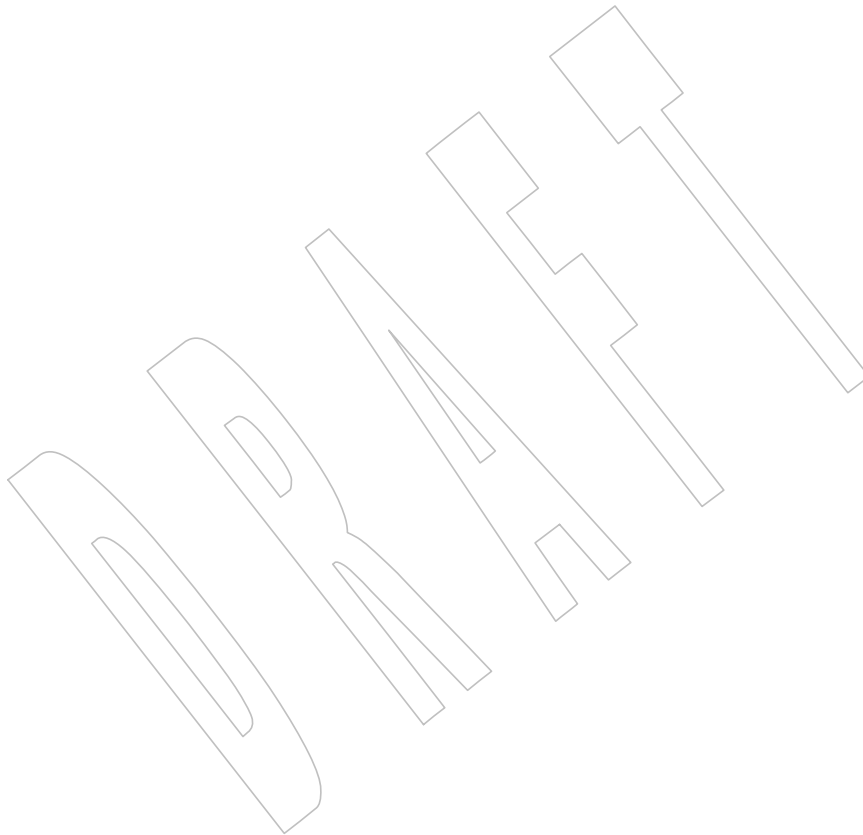
NAME

sleep — suspend execution for an interval of time

SYNOPSIS

```
#include <unistd.h>
```

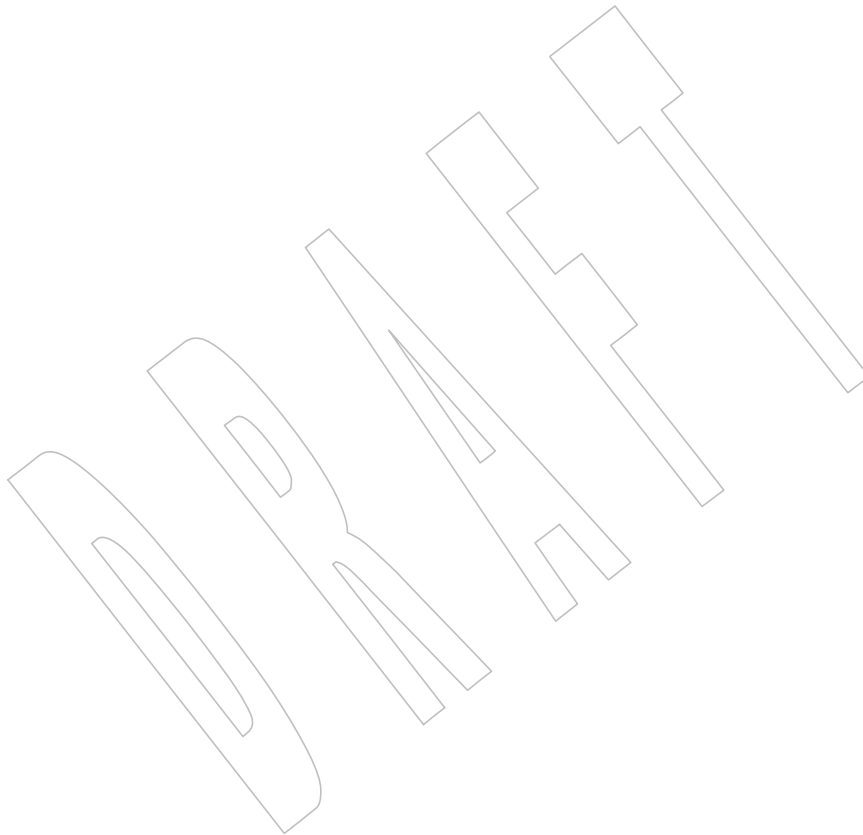
```
unsigned sleep(unsigned seconds);
```



sleep()

action previously established for SIGALRM, and whether SIGALRM was blocked. If a SIGALRM has been scheduled before the *sleep()* would ordinarily complete, the *sleep()* must be shortened to that time and a SIGALRM generated (possibly simulated by direct invocation of the signal-catching function) before *sleep()* returns. If a SIGALRM has been scheduled after the *sleep()* would ordinarily complete, it must be rescheduled for the same time before *sleep()* returns. The action and blocking for SIGALRM must be saved and restored.

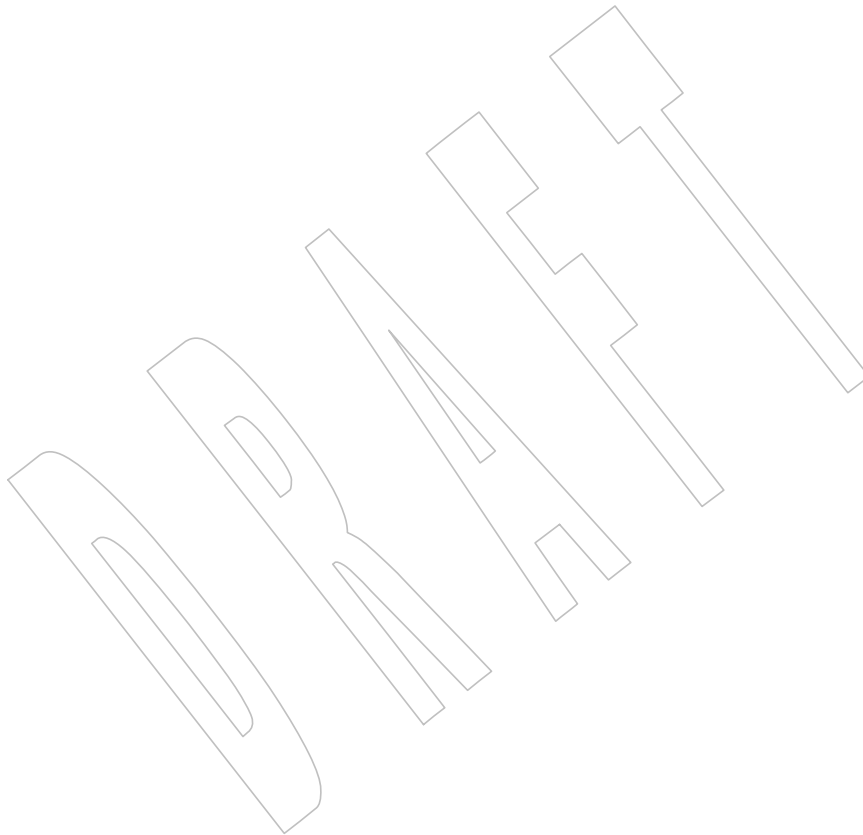
Historical implementations often implement the SIGALRM-based version using *alarm()* and *pause()*. One such implementation is prone to infinite hangups, as described in *pause()*. Another such implementation uses the C-language *setjmp()* and *longjmp()* functions to avoid that window. That implementation introduces a different problem: when the SIGALRM signal interrupts a signal-catching function installed by the user to catch a different signal, the *longjmp()* aborts that signal-catching function. An implementation based on *sigprocmask()*, *alarm()*, and *sigsuspend()* c



45400 **NAME**
45401 snprintf — print formatted output

45402 **SYNOPSIS**
45403 #include <stdio.h>
45404 int snprintf(char *restrict *s*, size_t *n*,
45405 const char *restrict *format*, ...);

45406 **DESCRIPTION**
45407 Refer to *fprintf()*.



45408 **NAME**
 45409 socketmark — determine whether a socket is at the out-of-band mark

45410 **SYNOPSIS**
 45411 #include <sys/socket.h>
 45412 int socketmark(int s);

45413 **DESCRIPTION**
 45414 The *socketmark()* function shall determine whether the socket specified by the descriptor *s* is at
 45415 the out-of-band data mark (see the System Interfaces volume of IEEE Std 1003.1-200x, Section
 45416 2.10.12, Socket Out-of-Band Data State). If the protocol for the socket supports out-of-band data
 45417 by marking the stream with an out-of-band data mark, the *socketmark()* function shall return 1
 45418 when all data preceding the mark has been read and the out-of-band data mark is the first
 45419 element in the receive queue. The *socketmark()* function shall not remove the mark from the
 45420 stream.

45421 **RETURN VALUE**
 45422 Upon successful completion, the *socketmark()* function shall return a value indicating whether
 45423 the socket is at an out-of-band data mark. If the protocol has marked the data stream and all data
 45424 preceding the mark has been read, the return value shall be 1; if there is no mark, or if data
 45425 precedes the mark in the receive queue, the *socketmark()* function shall return 0. Otherwise, it
 45426 shall return a value of -1 and set *errno* to indicate the error.

45427 **ERRORS**
 45428 The *socketmark()* function shall fail if:
 45429 [EBADF] The *s* argument is not a valid file descriptor.
 45430 [ENOTTY] The file associated with the *s* argument is not a socket.

45431 **EXAMPLES**
 45432 None.

45433 **APPLICATION USAGE**
 45434 The use of this function between receive operations allows an application to determine which
 45435 received data precedes the out-of-band data and which follows the out-of-band data.

45436 There is an inherent race condition in the use of this function. On an empty receive queue, the
 45437 current read of the location might well be at the “mark”, but the system has no way of knowing
 45438 that the next data segment that will arrive from the network will carry the mark, and
 45439 *socketmark()* will return false, and the next read operation will silently consume the mark.

45440 Hence, this function can only be used reliably when the application already knows that the out-
 45441 of-band data has been seen by the system or that it is known that there is data waiting to be read
 45442 at the socket (via SIGURG or *select()*). See [Section 2.10.11](#) (on page 62), [Section 2.10.12](#) (on page
 45443 63), [Section 2.10.14](#) (on page 63), and *pselect()* for details.

45444 **RATIONALE**
 45445 The *socketmark()* function replaces the historical SIOCATMARK command to *ioctl()* which
 45446 implemented the same functionality on many implementations. Using a wrapper function
 45447 follows the adopted conventions to avoid specifying commands to the *ioctl()* function, other
 45448 than those now included to support XSI STREAMS. The *socketmark()* function could be
 45449 implemented as follows:

```
45450 #include <sys/ioctl.h>
45451 int socketmark(int s)
45452 {
```

```
45453         int val;
45454         if (ioctl(s,SIOCATMARK,&val)==-1)
45455             return(-1);
45456         return(val);
45457     }
```

45458 The use of [ENOTTY] to indicate an incorrect descriptor type matches the historical behavior of
45459 SIOCATMARK.

45460 FUTURE DIRECTIONS

45461 None.

45462 SEE ALSO

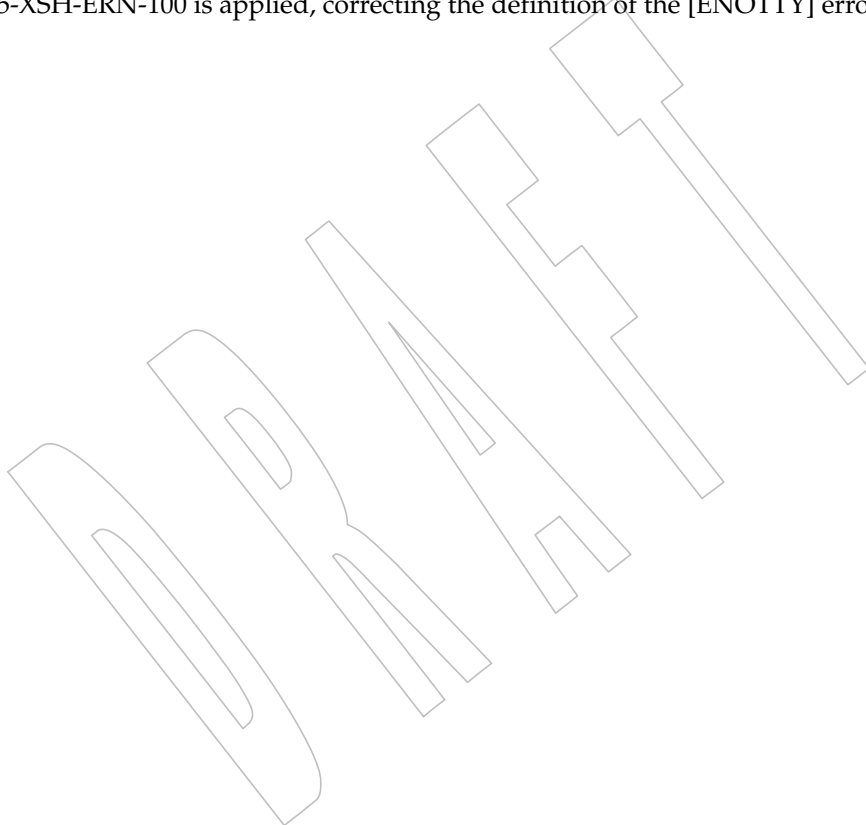
45463 *pselect()*, *recv()*, *recvmsg()*, the Base Definitions volume of IEEE Std 1003.1-200x, <sys/socket.h>

45464 CHANGE HISTORY

45465 First released in Issue 6. Derived from IEEE Std 1003.1g-2000.

45466 Issue 7

45467 SD5-XSH-ERN-100 is applied, correcting the definition of the [ENOTTY] error condition.



45468 **NAME**
 45469 socket — create an endpoint for communication

45470 **SYNOPSIS**
 45471 #include <sys/socket.h>
 45472 int socket(int domain, int type, int protocol);

45473 **DESCRIPTION**
 45474 The *socket()* function shall create an unbound socket in a communications domain, and return a
 45475 file descriptor that can be used in later function calls that operate on sockets.

45476 The *socket()* function takes the following arguments:

45477	<i>domain</i>	Specifies the communications domain in which a socket is to be created.
45478	<i>type</i>	Specifies the type of socket to be created.
45479	<i>protocol</i>	Specifies a particular protocol to be used with the socket. Specifying a <i>protocol</i> 45480 of 0 causes <i>socket()</i> to use an unspecified default protocol appropriate for the 45481 requested socket type.

45482 The *domain* argument specifies the address family used in the communications domain. The
 45483 address families supported by the system are implementation-defined.

45484 Symbolic constants that can be used for the domain argument are defined in the <sys/socket.h>
 45485 header.

45486 The *type* argument specifies the socket type, which determines the semantics of communication
 45487 over the socket. The following socket types are defined; implementations may specify additional
 45488 socket types:

45489	SOCK_STREAM	Provides sequenced, reliable, bidirectional, connection-mode byte 45490 streams, and may provide a transmission mechanism for out-of-band 45491 data.
45492	SOCK_DGRAM	Provides datagrams, which are connectionless-mode, unreliable messages 45493 of fixed maximum length.
45494	SOCK_SEQPACKET	Provides sequenced, reliable, bidirectional, connection-mode transmission 45495 paths for records. A record can be sent using one or more output 45496 operations and received using one or more input operations, but a single 45497 operation never transfers part of more than one record. Record 45498 boundaries are visible to the receiver via the MSG_EOR flag.

45499 If the *protocol* argument is non-zero, it shall specify a protocol that is supported by the address
 45500 family. If the *protocol* argument is zero, the default protocol for this address family and type shall
 45501 be used. The protocols supported by the system are implementation-defined.

45502 The process may need to have appropriate privileges to use the *socket()* function or to create
 45503 some sockets.

45504 **RETURN VALUE**
 45505 Upon successful completion, *socket()* shall return a non-negative integer, the socket file
 45506 descriptor. Otherwise, a value of -1 shall be returned and *errno* set to indicate the error.

45507 ERRORS

45508 The *socket()* function shall fail if:

45509 [EAFNOSUPPORT]

45510 The implementation does not support the specified address family.

45511 [EMFILE] All file descriptors available to the process are currently open.

45512 [ENFILE] No more file descriptors are available for the system.

45513 [EPROTONOSUPPORT]

45514 The protocol is not supported by the address family, or the protocol is not
45515 supported by the implementation.

45516 [EPROTOTYPE] The socket type is not supported by the protocol.

45517 The *socket()* function may fail if:

45518 [EACCES] The process does not have appropriate privileges.

45519 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

45520 [ENOMEM] Insufficient memory was available to fulfill the request.

45521 EXAMPLES

45522 None.

45523 APPLICATION USAGE

45524 The documentation for specific address families specifies which protocols each address family
45525 supports. The documentation for specific protocols specifies which socket types each protocol
45526 supports.

45527 The application can determine whether an address family is supported by trying to create a
45528 socket with *domain* set to the protocol in question.

45529 RATIONALE

45530 None.

45531 FUTURE DIRECTIONS

45532 None.

45533 SEE ALSO

45534 *accept()*, *bind()*, *connect()*, *getsockname()*, *getsockopt()*, *listen()*, *recv()*, *recvfrom()*, *recvmsg()*,
45535 *send()*, *sendmsg()*, *setsockopt()*, *shutdown()*, *socketpair()*, the Base Definitions volume of
45536 IEEE Std 1003.1-200x, <netinet/in.h>, <sys/socket.h>

45537 CHANGE HISTORY

45538 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

45539 **NAME**

45540 socketpair — create a pair of connected sockets

45541 **SYNOPSIS**

45542 #include <sys/socket.h>

45543 int socketpair(int domain, int type, int protocol,
45544 int socket_vector[2]);45545 **DESCRIPTION**45546 The *socketpair()* function shall create an unbound pair of connected sockets in a specified *domain*,
45547 of a specified *type*, under the protocol optionally specified by the *protocol* argument. The two
45548 sockets shall be identical. The file descriptors used in referencing the created sockets shall be
45549 returned in *socket_vector*[0] and *socket_vector*[1].45550 The *socketpair()* function takes the following arguments:

45551	<i>domain</i>	Specifies the communications domain in which the sockets are to be created.
45552	<i>type</i>	Specifies the type of sockets to be created.
45553	<i>protocol</i>	Specifies a particular protocol to be used with the sockets. Specifying a 45554 <i>protocol</i> of 0 causes <i>socketpair()</i> to use an unspecified default protocol 45555 appropriate for the requested socket type.
45556	<i>socket_vector</i>	Specifies a 2-integer array to hold the file descriptors of the created socket pair.

45557 The *type* argument specifies the socket type, which determines the semantics of communications
45558 over the socket. The following socket types are defined; implementations may specify additional
45559 socket types:

45560	SOCK_STREAM	Provides sequenced, reliable, bidirectional, connection-mode byte 45561 streams, and may provide a transmission mechanism for out-of-band 45562 data.
45563	SOCK_DGRAM	Provides datagrams, which are connectionless-mode, unreliable messages 45564 of fixed maximum length.
45565	SOCK_SEQPACKET	Provides sequenced, reliable, bidirectional, connection-mode transmission 45566 paths for records. A record can be sent using one or more output 45567 operations and received using one or more input operations, but a single 45568 operation never transfers part of more than one record. Record 45569 boundaries are visible to the receiver via the MSG_EOR flag.

45570 If the *protocol* argument is non-zero, it shall specify a protocol that is supported by the address
45571 family. If the *protocol* argument is zero, the default protocol for this address family and type shall
45572 be used. The protocols supported by the system are implementation-defined.45573 The process may need to have appropriate privileges to use the *socketpair()* function or to create
45574 some sockets.45575 **RETURN VALUE**45576 Upon successful completion, this function shall return 0; otherwise, -1 shall be returned and
45577 *errno* set to indicate the error.45578 **ERRORS**45579 The *socketpair()* function shall fail if:

- 45580 [EAFNOSUPPORT]
 45581 The implementation does not support the specified address family.
- 45582 [EMFILE] All file descriptors available to the process are currently open.
- 45583 [ENFILE] No more file descriptors are available for the system.
- 45584 [EOPNOTSUPP] The specified protocol does not permit creation of socket pairs.
- 45585 [EPROTONOSUPPORT]
 45586 The protocol is not supported by the address family, or the protocol is not
 45587 supported by the implementation.
- 45588 [EPROTOTYPE] The socket type is not supported by the protocol.
- 45589 The *socketpair()* function may fail if:
- 45590 [EACCES] The process does not have appropriate privileges.
- 45591 [ENOBUFS] Insufficient resources were available in the system to perform the operation.
- 45592 [ENOMEM] Insufficient memory was available to fulfill the request.

EXAMPLES

45593 None.
 45594

APPLICATION USAGE

45595 The documentation for specific address families specifies which protocols each address family
 45596 supports. The documentation for specific protocols specifies which socket types each protocol
 45597 supports.
 45598

45599 The *socketpair()* function is used primarily with UNIX domain sockets and need not be
 45600 supported for other domains.

RATIONALE

45601 None.
 45602

FUTURE DIRECTIONS

45603 None.
 45604

SEE ALSO

45605 *socket()*, the Base Definitions volume of IEEE Std 1003.1-200x, <sys/socket.h>
 45606

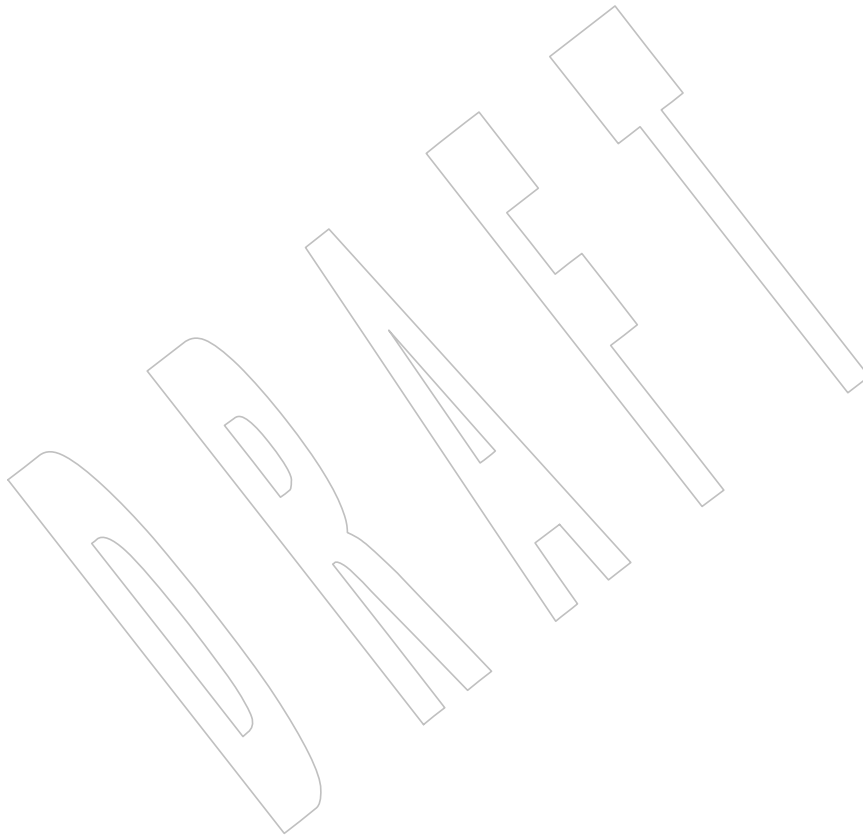
CHANGE HISTORY

45607 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
 45608

45609 **NAME**
45610 `printf` — print formatted output

45611 **SYNOPSIS**
45612 `#include <stdio.h>`
45613 `int sprintf(char *restrict s, const char *restrict format, ...);`

45614 **DESCRIPTION**
45615 Refer to *fprintf()*.



45616 **NAME**
 45617 sqrt, sqrtf, sqrtl — square root function

45618 **SYNOPSIS**
 45619 #include <math.h>
 45620 double sqrt(double x);
 45621 float sqrtf(float x);
 45622 long double sqrtl(long double x);

45623 DESCRIPTION

45624 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 45625 conflict between the requirements described here and the ISO C standard is unintentional. This
 45626 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

45627 These functions shall compute the square root of their argument x , \sqrt{x} .

45628 An application wishing to check for error situations should set *errno* to zero and call
 45629 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 45630 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 45631 zero, an error has occurred.

45632 RETURN VALUE

45633 Upon successful completion, these functions shall return the square root of x .

45634 MX For finite values of $x < -0$, a domain error shall occur, and either a NaN (if supported), or an
 45635 implementation-defined value shall be returned.

45636 MX If x is NaN, a NaN shall be returned.

45637 If x is ± 0 or $+\text{Inf}$, x shall be returned.

45638 If x is $-\text{Inf}$, a domain error shall occur, and either a NaN (if supported), or an implementation-
 45639 defined value shall be returned.

45640 ERRORS

45641 These functions shall fail if:

45642 MX Domain Error The finite value of x is < -0 , or x is $-\text{Inf}$.

45643 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 45644 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 45645 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 45646 shall be raised.

45647 EXAMPLES

45648 Taking the Square Root of 9.0

```
45649 #include <math.h>
45650 ...
45651 double x = 9.0;
45652 double result;
45653 ...
45654 result = sqrt(x);
```


45655

APPLICATION USAGE

45656

On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

45657

45658

RATIONALE

45659

None.

45660

FUTURE DIRECTIONS

45661

None.

45662

SEE ALSO

45663

feclearexcept(), *fetetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**, **<stdio.h>**

45664

45665

CHANGE HISTORY

45666

First released in Issue 1. Derived from Issue 1 of the SVID.

45667

Issue 5

45668

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

45669

45670

Issue 6

45671

The *sqrtf()* and *sqrtrl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

45672

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

45673

45674

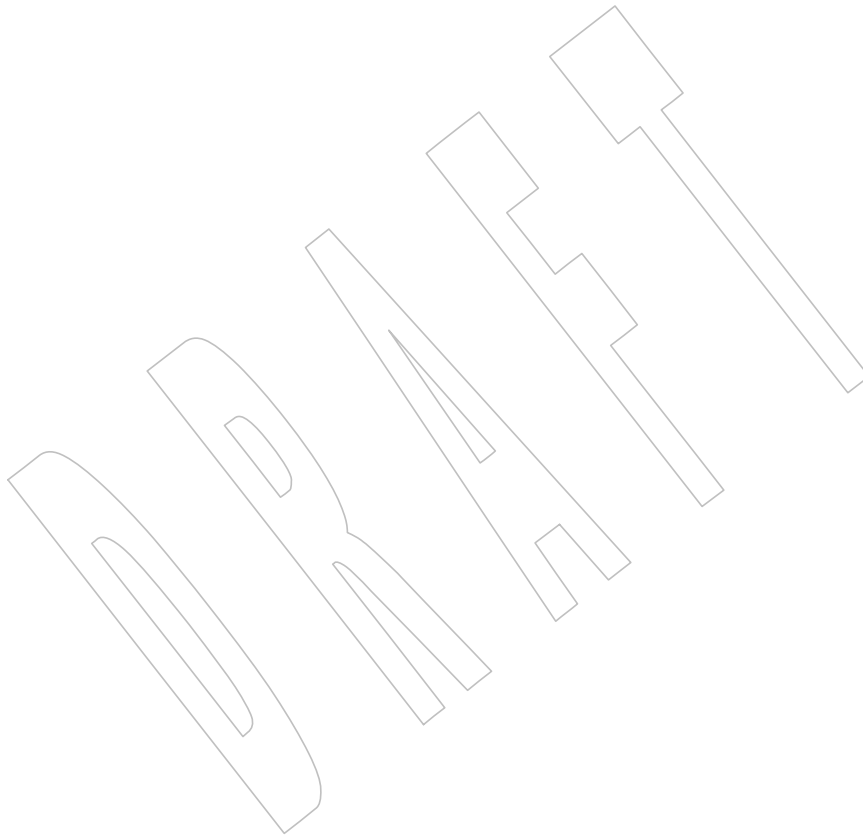
IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

45675

45676 **NAME**
45677 `srand` — pseudo-random number generator

45678 **SYNOPSIS**
45679 `#include <stdlib.h>`
45680 `void srand(unsigned seed);`

45681 **DESCRIPTION**
45682 Refer to *rand()*.



srand48()

45683 **NAME**
45684 `srand48` — seed the uniformly distributed double-precision pseudo-random number generator

SYNOPSIS

45685 XSI `#include <stdlib.h>`
45686 `void srand48(long seedval);`

DESCRIPTION

45688 Refer to *drand48()*.
45689

45690 **NAME**
45691 `srandom` — seed pseudo-random number generator

45692 **SYNOPSIS**

```
45693 XSI #include <stdlib.h>  
45694 void srandom(unsigned seed);
```

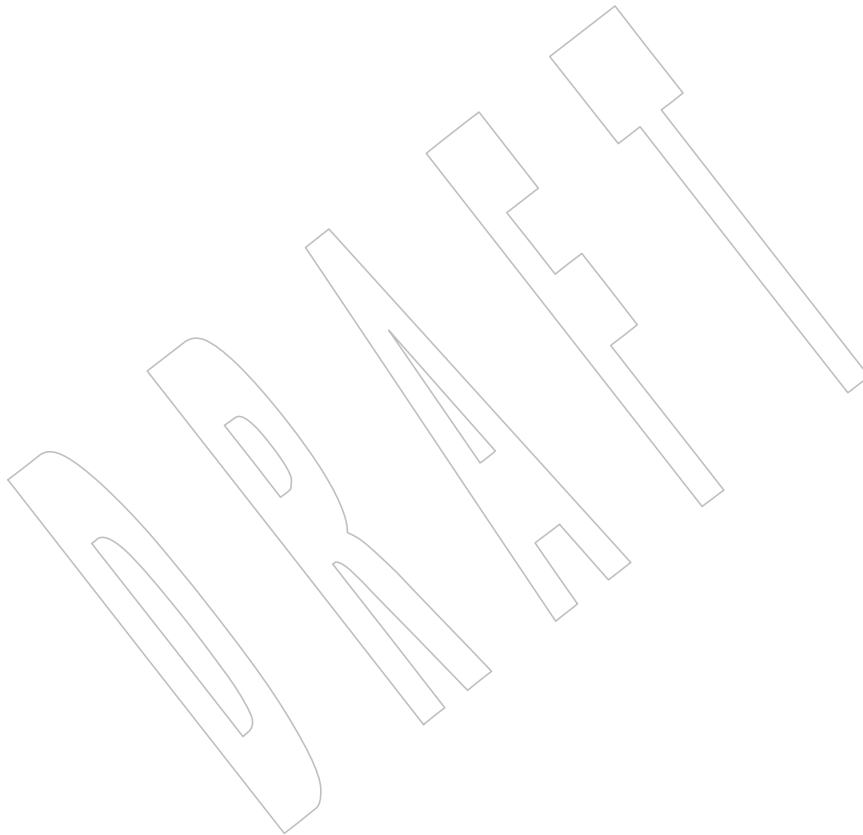
45695 **DESCRIPTION**

45696 Refer to [initstate\(\)](#).

45697 **NAME**
45698 scanf — convert formatted input

45699 **SYNOPSIS**
45700 #include <stdio.h>
45701 int sscanf(const char *restrict s, const char *restrict format, ...);

45702 **DESCRIPTION**
45703 Refer to *fscanf()*.

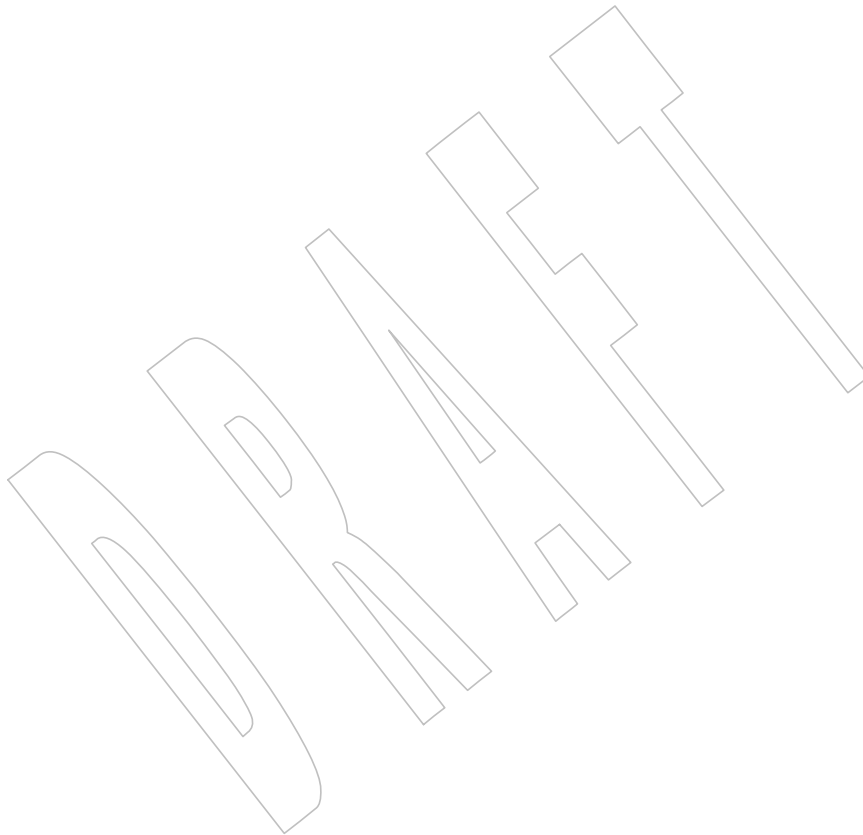


45704 **NAME**
45705 stat — get file status

45706 **SYNOPSIS**
45707 #include <sys/stat.h>

45708 int stat(const char *restrict path, struct stat *restrict buf);

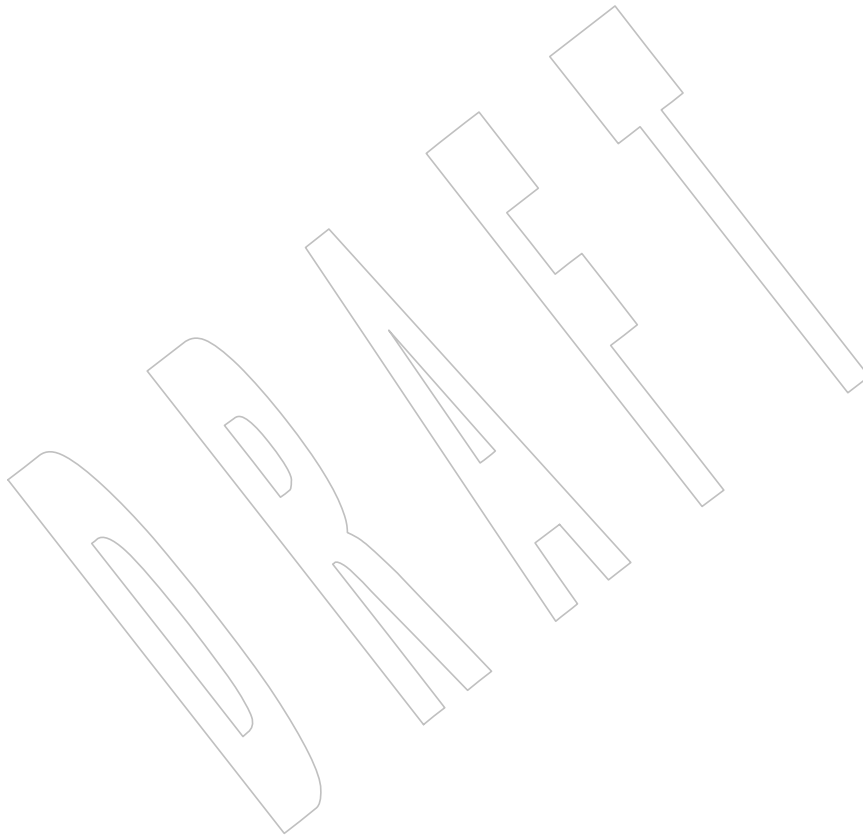
45709 **DESCRIPTION**
45710 Refer to *fstatat()*.



45711 **NAME**
45712 `statvfs` — get file system information

45713 **SYNOPSIS**
45714 `#include <sys/statvfs.h>`
45715 `int statvfs(const char *restrict path, struct statvfs *restrict buf);`

45716 **DESCRIPTION**
45717 Refer to *fstatvfs()*.



45718 **NAME**
 45719 stderr, stdin, stdout — standard I/O streams

45720 **SYNOPSIS**
 45721 #include <stdio.h>
 45722 extern FILE *stderr, *stdin, *stdout;

45723 **DESCRIPTION**
 45724 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 45725 conflict between the requirements described here and the ISO C standard is unintentional. This
 45726 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

45727 A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type
 45728 **FILE**. The *fopen()* function shall create certain descriptive data for a stream and return a pointer
 45729 to designate the stream in all further transactions. Normally, there are three open streams with
 45730 constant pointers declared in the <stdio.h> header and associated with the standard open files.

45731 At program start-up, three streams shall be predefined and need not be opened explicitly:
 45732 *standard input* (for reading conventional input), *standard output* (for writing conventional output),
 45733 and *standard error* (for writing diagnostic output). When opened, the standard error stream is not
 45734 fully buffered; the standard input and standard output streams are fully buffered if and only if
 45735 the stream can be determined not to refer to an interactive device.

45736 CX The following symbolic values in <unistd.h> define the file descriptors that shall be associated
 45737 with the C-language *stdin*, *stdout*, and *stderr* when the application is started:

45738 STDIN_FILENO Standard input value, *stdin*. Its value is 0.
 45739 STDOUT_FILENO Standard output value, *stdout*. Its value is 1.
 45740 STDERR_FILENO Standard error value, *stderr*. Its value is 2.

45741 The *stderr* stream is expected to be open for reading and writing.

45742 **RETURN VALUE**
 45743 None.

45744 **ERRORS**
 45745 No errors are defined.

45746 **EXAMPLES**
 45747 None.

45748 **APPLICATION USAGE**
 45749 None.

45750 **RATIONALE**
 45751 None.

45752 **FUTURE DIRECTIONS**
 45753 None.

45754 **SEE ALSO**
 45755 *fclose()*, *feof()*, *ferror()*, *fileno()*, *fopen()*, *fread()*, *fseek()*, *getc()*, *gets()*, *popen()*, *printf()*, *putc()*,
 45756 *puts()*, *read()*, *scanf()*, *setbuf()*, *setvbuf()*, *tmpfile()*, *ungetc()*, *vprintf()*, the Base Definitions
 45757 volume of IEEE Std 1003.1-200x, <stdio.h>, <unistd.h>

45758

CHANGE HISTORY

45759

First released in Issue 1.

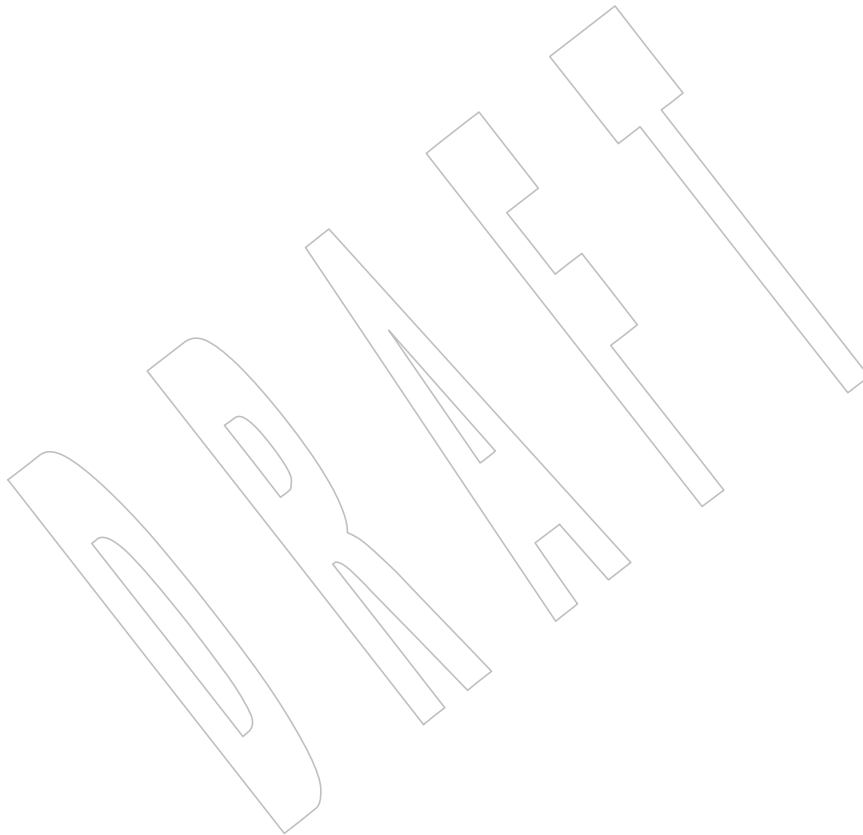
45760

Issue 6

45761

Extensions beyond the ISO C standard are marked.

45762

A note that *stderr* is expected to be open for reading and writing is added to the DESCRIPTION.

45763 **NAME**
45764 stpcpy — copy a string and return a pointer to the end of the result

45765 **SYNOPSIS**

45766 CX `#include <string.h>`
45767 `char *stpcpy(char *restrict s1, const char *restrict s2);`

45768 **DESCRIPTION**

45769 Refer to *strcpy()*.

stpncpy()

45770 **NAME**
45771 stpncpy — copy fixed length string, returning a pointer to the array end

SYNOPSIS

45772 CX `#include <string.h>`
45773 `char *stpncpy(char *restrict s1, const char *restrict s2, size_t size);`

DESCRIPTION

45775 Refer to *strncpy()*.
45776

45777 **NAME**
 45778 `strcasemp, strcasemp_l, strncasemp, strncasemp_l` — case-insensitive string comparisons

45779 **SYNOPSIS**
 45780 `#include <strings.h>`
 45781 `int strcasemp(const char *s1, const char *s2);`
 45782 `int strcasemp_l(const char *s1, const char *s2,`
 45783 `locale_t locale);`
 45784 `int strncasemp(const char *s1, const char *s2, size_t n);`
 45785 `int strncasemp_l(const char *s1, const char *s2,`
 45786 `size_t n, locale_t locale);`

45787 **DESCRIPTION**
 45788 The `strcasemp()` and `strcasemp_l()` functions shall compare, while ignoring differences in case,
 45789 the string pointed to by `s1` to the string pointed to by `s2`. The `strncasemp()` and `strncasemp_l()`
 45790 functions shall compare, while ignoring differences in case, not more than `n` bytes from the
 45791 string pointed to by `s1` to the string pointed to by `s2`.

45792 The `strcasemp()` and `strncasemp()` functions use the current locale of the process to determine
 45793 the case of the characters.

45794 The `strcasemp_l()` and `strncasemp_l()` functions use the locale represented by `locale` to determine
 45795 the case of the characters.

45796 When the `LC_CTYPE` category of the current locale is from the POSIX locale, `strcasemp()` and
 45797 `strncasemp()` shall behave as if the strings had been converted to lowercase and then a byte
 45798 comparison performed. Otherwise, the results are unspecified.

45799 **RETURN VALUE**
 45800 Upon completion, `strcasemp()` and `strcasemp_l()` shall return an integer greater than, equal to,
 45801 or less than 0, if the string pointed to by `s1` is, ignoring case, greater than, equal to, or less than
 45802 the string pointed to by `s2`, respectively.

45803 Upon successful completion, `strncasemp()` and `strncasemp_l()` shall return an integer greater
 45804 than, equal to, or less than 0, if the possibly null-terminated array pointed to by `s1` is, ignoring
 45805 case, greater than, equal to, or less than the possibly null-terminated array pointed to by `s2`,
 45806 respectively.

45807 **ERRORS**
 45808 The `strcasemp_l()` and `strncasemp_l()` functions may fail if:
 45809 [EINVAL] `locale` is not a valid locale object handle.

45810 **EXAMPLES**
 45811 None.

45812 **APPLICATION USAGE**
 45813 None.

45814 **RATIONALE**
 45815 None.

45816 **FUTURE DIRECTIONS**
 45817 None.

strcasemp()

45818

SEE ALSO

45819

wscasemp(), the Base Definitions volume of IEEE Std 1003.1-200x, <strings.h>

45820

CHANGE HISTORY

45821

First released in Issue 4, Version 2.

45822

Issue 5

45823

Moved from X/OPEN UNIX extension to BASE.

45824

Issue 7

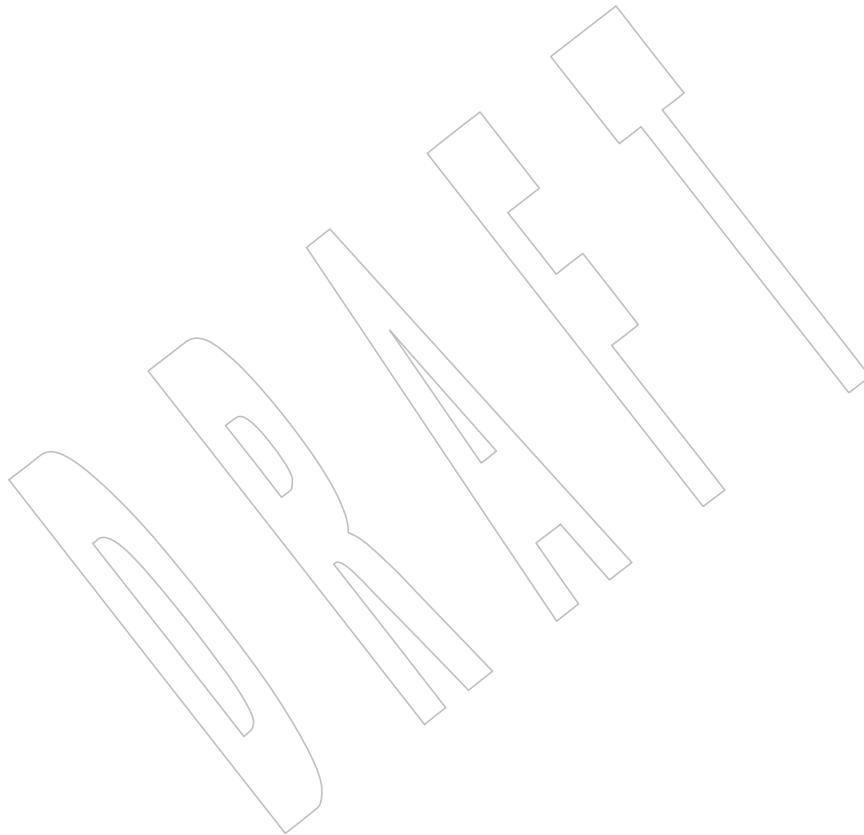
45825

The *strcasemp()* and *strncasemp()* functions are moved from the XSI option to the Base.

45826

The *strcasemp_l()* and *strncasemp_l()* functions are added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

45827



45828 **NAME**
 45829 `strcat` — concatenate two strings

45830 **SYNOPSIS**
 45831 `#include <string.h>`
 45832 `char *strcat(char *restrict s1, const char *restrict s2);`

45833 **DESCRIPTION**
 45834 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 45835 conflict between the requirements described here and the ISO C standard is unintentional. This
 45836 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

45837 The `strcat()` function shall append a copy of the string pointed to by `s2` (including the
 45838 terminating NUL character) to the end of the string pointed to by `s1`. The initial byte of `s2`
 45839 overwrites the NUL character at the end of `s1`. If copying takes place between objects that
 45840 overlap, the behavior is undefined.

45841 **RETURN VALUE**
 45842 The `strcat()` function shall return `s1`; no return value is reserved to indicate an error.

45843 **ERRORS**
 45844 No errors are defined.

45845 **EXAMPLES**
 45846 None.

45847 **APPLICATION USAGE**
 45848 This issue is aligned with the ISO C standard; this does not affect compatibility with XPG3
 45849 applications. Reliable error detection by this function was never guaranteed.

45850 **RATIONALE**
 45851 None.

45852 **FUTURE DIRECTIONS**
 45853 None.

45854 **SEE ALSO**
 45855 `strncat()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<string.h>`

45856 **CHANGE HISTORY**
 45857 First released in Issue 1. Derived from Issue 1 of the SVID.

45858 **Issue 6**
 45859 The `strcat()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

45860 **NAME**
 45861 `strchr` — string scanning operation

45862 **SYNOPSIS**
 45863 `#include <string.h>`

45864 `char *strchr(const char *s, int c);`

45865 **DESCRIPTION**

45866 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 45867 conflict between the requirements described here and the ISO C standard is unintentional. This
 45868 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

45869 The `strchr()` function shall locate the first occurrence of `c` (converted to a **char**) in the string
 45870 pointed to by `s`. The terminating NUL character is considered to be part of the string.

45871 **RETURN VALUE**

45872 Upon completion, `strchr()` shall return a pointer to the byte, or a null pointer if the byte was not
 45873 found.

45874 **ERRORS**

45875 No errors are defined.

45876 **EXAMPLES**

45877 None.

45878 **APPLICATION USAGE**

45879 None.

45880 **RATIONALE**

45881 None.

45882 **FUTURE DIRECTIONS**

45883 None.

45884 **SEE ALSO**

45885 [*strchr\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<string.h>`

45886 **CHANGE HISTORY**

45887 First released in Issue 1. Derived from Issue 1 of the SVID.

45888 **Issue 6**

45889 Extensions beyond the ISO C standard are marked.

45890 **NAME**
 45891 strcmp — compare two strings

45892 **SYNOPSIS**
 45893 #include <string.h>
 45894 int strcmp(const char *s1, const char *s2);

45895 **DESCRIPTION**
 45896 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 45897 conflict between the requirements described here and the ISO C standard is unintentional. This
 45898 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

45899 The *strcmp()* function shall compare the string pointed to by *s1* to the string pointed to by *s2*.
 45900 The sign of a non-zero return value shall be determined by the sign of the difference between the
 45901 values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the strings
 45902 being compared.

45903 **RETURN VALUE**
 45904 Upon completion, *strcmp()* shall return an integer greater than, equal to, or less than 0, if the
 45905 string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*,
 45906 respectively.

45907 **ERRORS**
 45908 No errors are defined.

45909 **EXAMPLES**

45910 **Checking a Password Entry**

45911 The following example compares the information read from standard input to the value of the
 45912 name of the user entry. If the *strcmp()* function returns 0 (indicating a match), a further check
 45913 will be made to see if the user entered the proper old password. The *crypt()* function shall
 45914 encrypt the old password entered by the user, using the value of the encrypted password in the
 45915 **passwd** structure as the salt. If this value matches the value of the encrypted **passwd** in the
 45916 structure, the entered password *oldpasswd* is the correct user's password. Finally, the program
 45917 encrypts the new password so that it can store the information in the **passwd** structure.

```

45918 #include <string.h>
45919 #include <unistd.h>
45920 #include <stdio.h>
45921 ...
45922 int valid_change;
45923 struct passwd *p;
45924 char user[100];
45925 char oldpasswd[100];
45926 char newpasswd[100];
45927 char savepasswd[100];
45928 ...
45929 if (strcmp(p->pw_name, user) == 0) {
45930     if (strcmp(p->pw_passwd, crypt(oldpasswd, p->pw_passwd)) == 0) {
45931         strcpy(savepasswd, crypt(newpasswd, user));
45932         p->pw_passwd = savepasswd;
45933         valid_change = 1;
45934     }
45935     else {
```


strcmp()

```
45936         fprintf(stderr, "Old password is not valid\n");
45937     }
45938 }
45939 ...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

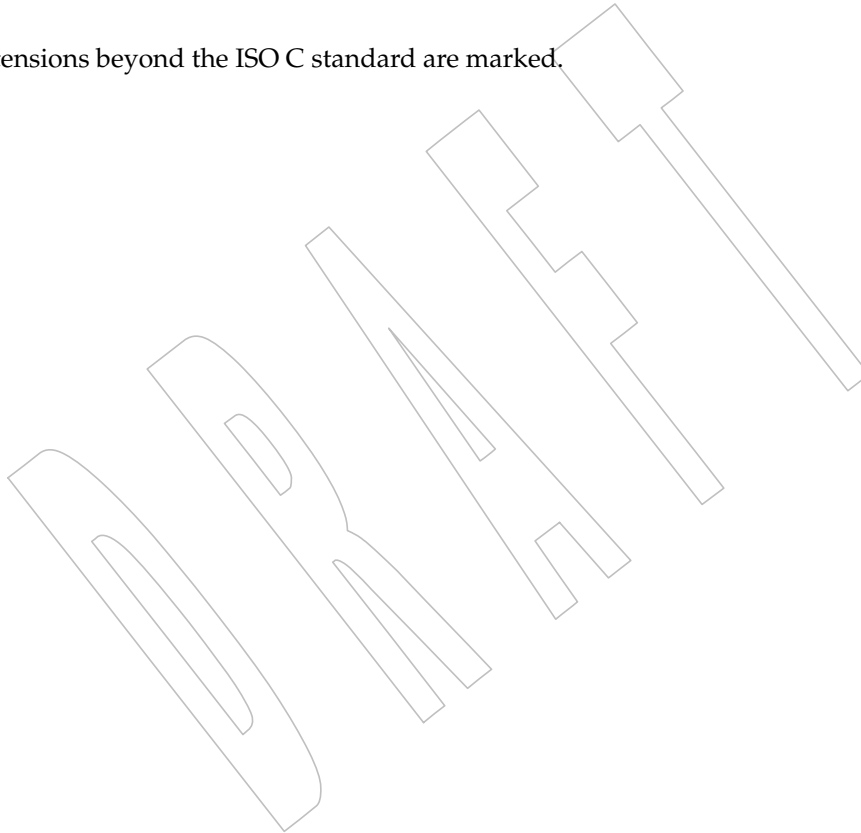
strncmp(), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are marked.



45952 **NAME**

45953 strcoll, strcoll_l — string comparison using collating information

45954 **SYNOPSIS**

45955 #include <string.h>

45956 int strcoll(const char *s1, const char *s2);

45957 CX int strcoll_l(const char *s1, const char *s2,
45958 locale_t locale);45959 **DESCRIPTION**45960 CX For *strcoll()*: The functionality described on this reference page is aligned with the ISO C
45961 standard. Any conflict between the requirements described here and the ISO C standard is
45962 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.45963 CX The *strcoll()* and *strcoll_l()* functions shall compare the string pointed to by *s1* to the string
45964 pointed to by *s2*, both interpreted as appropriate to the *LC_COLLATE* category of the current
45965 locale, or of the locale represented by *locale*, respectively.45966 CX The *strcoll()* and *strcoll_l()* functions shall not change the setting of *errno* if successful.45967 Since no return value is reserved to indicate an error, an application wishing to check for error
45968 CX situations should set *errno* to 0, then call *strcoll()*, or *strcoll_l()* then check *errno*.45969 **RETURN VALUE**45970 Upon successful completion, *strcoll()* shall return an integer greater than, equal to, or less than 0,
45971 according to whether the string pointed to by *s1* is greater than, equal to, or less than the string
45972 CX pointed to by *s2* when both are interpreted as appropriate to the current locale. On error,
45973 *strcoll()* may set *errno*, but no return value is reserved to indicate an error.45974 Upon successful completion, *strcoll_l()* shall return an integer greater than, equal to, or less than
45975 0, according to whether the string pointed to by *s1* is greater than, equal to, or less than the
45976 string pointed to by *s2* when both are interpreted as appropriate to the locale represented by
45977 *locale*. On error, *strcoll_l()* may set *errno*, but no return value is reserved to indicate an error.45978 **ERRORS**

45979 These functions may fail if:

45980 CX [EINVAL] The *s1* or *s2* arguments contain characters outside the domain of the collating
45981 sequence.45982 The *strcoll_l()* function may fail if:45983 CX [EINVAL] *locale* is not a valid locale object handle.45984 **EXAMPLES**45985 **Comparing Nodes**45986 The following example uses an application-defined function, *node_compare()*, to compare two
45987 nodes based on an alphabetical ordering of the *string* field.

45988 #include <string.h>

45989 ...

45990 struct node { /* These are stored in the table. */

45991 char *string;

45992 int length;

45993 };

```

45994     ...
45995     int node_compare(const void *node1, const void *node2)
45996     {
45997         return strcoll(((const struct node *)node1)->string,
45998                       ((const struct node *)node2)->string);
45999     }
46000     ...

```

APPLICATION USAGE

The *strxfrm()* and *strcmp()* functions should be used for sorting large lists.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

alphasort(), *strcmp()*, *strxfrm()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 5

The DESCRIPTION is updated to indicate that *errno* does not change if the function is successful.

Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EINVAL] optional error condition is added.

An example is added.

Issue 7

The *strcoll_1()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

46022 **NAME**

46023 stpcpy, strcpy — copy a string and return a pointer to the end of the result

46024 **SYNOPSIS**

46025 #include <string.h>

46026 CX char *stpcpy(char *restrict s1, const char *restrict s2);

46027 char *strcpy(char *restrict s1, const char *restrict s2);

46028 **DESCRIPTION**46029 CX For *strcpy()*: The functionality described on this reference page is aligned with the ISO C
46030 standard. Any conflict between the requirements described here and the ISO C standard is
46031 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.46032 CX The *stpcpy()* and *strcpy()* functions shall copy the string pointed to by *s2* (including the
46033 terminating NUL character) into the array pointed to by *s1*.

46034 If copying takes place between objects that overlap, the behavior is undefined.

46035 **RETURN VALUE**46036 CX The *stpcpy()* function shall return a pointer to the terminating NUL character copied into the *s1*
46037 buffer.46038 The *strcpy()* function shall return *s1*.

46039 No return values are reserved to indicate an error.

46040 **ERRORS**

46041 No errors are defined.

46042 **EXAMPLES**46043 **Construction of a Multi-Part Message in a Single Buffer**

46044 #include <string.h>

46045 #include <stdio.h>

46046 int

46047 main (void)

46048 {

46049 char buffer [10];

46050 char *name = buffer;

46051 name = stpcpy (stpcpy (stpcpy (name, "ice"), "-"), "cream");

46052 puts (buffer);

46053 return 0;

46054 }

46055 **Initializing a String**46056 The following example copies the string "-----" into the *permstring* variable.

46057 #include <string.h>

46058 ...

46059 static char permstring[11];

46060 ...

46061 strcpy(permstring, "-----");

46062 ...

46063

Storing a Key and Data

46064

46065

46066

46067

The following example allocates space for a key using *malloc()* then uses *strcpy()* to place the key there. Then it allocates space for data using *malloc()*, and uses *strcpy()* to place data there. (The user-defined function *dbfree()* frees memory previously allocated to an array of type **struct element** *.)

46068

46069

46070

46071

46072

46073

46074

46075

46076

46077

46078

46079

46080

46081

46082

46083

46084

46085

46086

46087

46088

46089

46090

46091

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
...
/* Structure used to read data and store it. */
struct element {
    char *key;
    char *data;
};

struct element *tbl, *curtbl;
char *key, *data;
int count;
...
void dbfree(struct element *, int);
...
if ((curtbl->key = malloc(strlen(key) + 1)) == NULL) {
    perror("malloc"); dbfree(tbl, count); return NULL;
}
strcpy(curtbl->key, key);

if ((curtbl->data = malloc(strlen(data) + 1)) == NULL) {
    perror("malloc"); free(curtbl->key); dbfree(tbl, count); return NULL;
}
strcpy(curtbl->data, data);
...
```

46092

APPLICATION USAGE

46093

46094

Character movement is performed differently in different implementations. Thus, overlapping moves may yield surprises.

46095

46096

This issue is aligned with the ISO C standard; this does not affect compatibility with XPG3 applications. Reliable error detection by this function was never guaranteed.

46097

RATIONALE

46098

None.

46099

FUTURE DIRECTIONS

46100

None.

46101

SEE ALSO

46102

strncpy(), *wcscpy()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

46103

CHANGE HISTORY

46104

First released in Issue 1. Derived from Issue 1 of the SVID.

46105

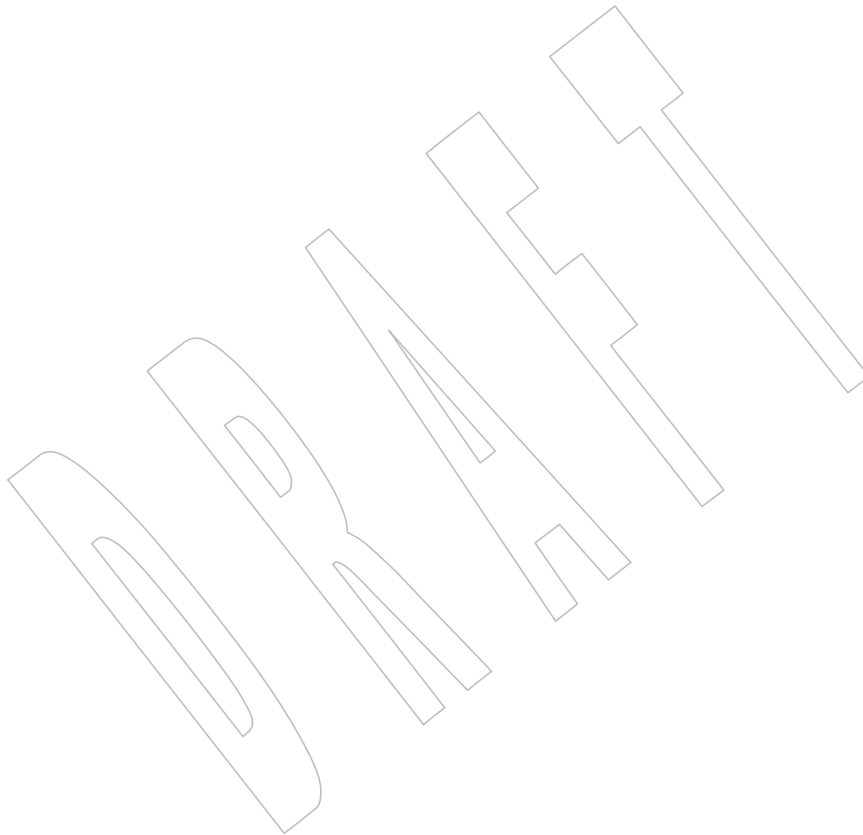
Issue 6

46106

The *strcpy()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

46107
46108
46109**Issue 7**

The *strcpy()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.



46110 **NAME**

46111 strcspn — get the length of a complementary substring

46112 **SYNOPSIS**

46113 #include <string.h>

46114 size_t strcspn(const char *s1, const char *s2);

46115 **DESCRIPTION**46116 CX The functionality described on this reference page is aligned with the ISO C standard. Any
46117 conflict between the requirements described here and the ISO C standard is unintentional. This
46118 volume of IEEE Std 1003.1-200x defers to the ISO C standard.46119 The *strcspn()* function shall compute the length (in bytes) of the maximum initial segment of the
46120 string pointed to by *s1* which consists entirely of bytes *not* from the string pointed to by *s2*.46121 **RETURN VALUE**46122 The *strcspn()* function shall return the length of the computed segment of the string pointed to
46123 by *s1*; no return value is reserved to indicate an error.46124 **ERRORS**

46125 No errors are defined.

46126 **EXAMPLES**

46127 None.

46128 **APPLICATION USAGE**

46129 None.

46130 **RATIONALE**

46131 None.

46132 **FUTURE DIRECTIONS**

46133 None.

46134 **SEE ALSO**46135 *strspn()*, the Base Definitions volume of IEEE Std 1003.1-200x, <string.h>46136 **CHANGE HISTORY**

46137 First released in Issue 1. Derived from Issue 1 of the SVID.

46138 **Issue 5**46139 The RETURN VALUE section is updated to indicate that *strcspn()* returns the length of *s1*, and
46140 not *s1* itself as was previously stated.46141 **Issue 6**46142 The Open Group Corrigendum U030/1 is applied. The text of the RETURN VALUE section is
46143 updated to indicate that the computed segment length is returned, not the *s1* length.

46144 **NAME**46145 `strdup, strndup` — duplicate a specific number of bytes from a string46146 **SYNOPSIS**

```
46147 CX    #include <string.h>
46148      char *strdup(const char *s);
46149      char *strndup(const char *s, size_t size);
```

46150 **DESCRIPTION**

46151 The `strdup()` function shall return a pointer to a new string, which is a duplicate of the string
 46152 pointed to by `s`. The returned pointer can be passed to `free()`. A null pointer is returned if the
 46153 new string cannot be created.

46154 The `strndup()` function shall be equivalent to the `strdup()` function, duplicating the provided `s` in
 46155 a new block of memory allocated as if by using `malloc()`, with the exception being that `strndup()`
 46156 copies at most `size` plus one bytes into the newly allocated memory, terminating the new string
 46157 with a NUL character. If the length of `s` is larger than `size`, only `size` bytes shall be duplicated. If
 46158 `size` is larger than the length of `s`, all bytes in `s` shall be copied into the new memory buffer,
 46159 including the terminating NUL character. The newly created string shall always be properly
 46160 terminated.

46161 **RETURN VALUE**

46162 The `strdup()` function shall return a pointer to a new string on success. Otherwise, it shall return
 46163 a null pointer and set `errno` to indicate the error.

46164 Upon successful completion, the `strndup()` function shall return a pointer to the newly allocated
 46165 memory containing the duplicated string. Otherwise, it shall return a null pointer and set `errno`
 46166 to indicate the error.

46167 **ERRORS**

46168 These functions shall fail if:

46169 [ENOMEM] Storage space available is insufficient.

46170 **EXAMPLES**

46171 None.

46172 **APPLICATION USAGE**

46173 None.

46174 **RATIONALE**

46175 None.

46176 **FUTURE DIRECTIONS**

46177 None.

46178 **SEE ALSO**46179 `free()`, `malloc()`, `wcsdup()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<string.h>`46180 **CHANGE HISTORY**

46181 First released in Issue 4, Version 2.

46182 **Issue 5**

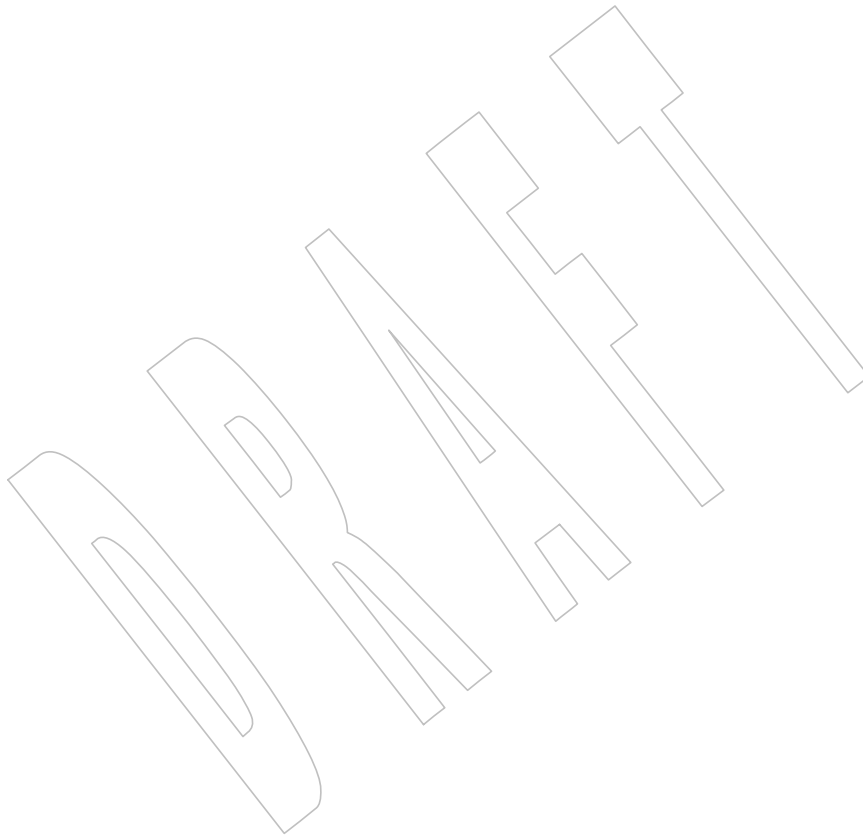
46183 Moved from X/OPEN UNIX extension to BASE.

strdup()46184
46185
46186
46187
46188
46189**Issue 7**

Austin Group Interpretation 1003.1-2001 #044 is applied, changing the “may fail” [ENOMEM] error to become a “shall fail” error.

The *strdup()* function is moved from the XSI option to the Base.

The *strndup()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.



46190 **NAME**
 46191 `strerror, strerror_l, strerror_r` — get error message string

46192 **SYNOPSIS**
 46193 `#include <string.h>`
 46194 `char *strerror(int errnum);`
 46195 CX `char *strerror_l(int errnum, locale_t locale);`
 46196 `int strerror_r(int errnum, char *strerrbuf, size_t buflen);`

46197 **DESCRIPTION**

46198 CX For `strerror()`: The functionality described on this reference page is aligned with the ISO C
 46199 standard. Any conflict between the requirements described here and the ISO C standard is
 46200 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

46201 The `strerror()` function shall map the error number in `errnum` to a locale-dependent error
 46202 message string and shall return a pointer to it. Typically, the values for `errnum` come from `errno`,
 46203 but `strerror()` shall map any value of type `int` to a message.

46204 The string pointed to shall not be modified by the application. The string may be overwritten by
 46205 CX a subsequent call to `strerror()` or `perror()`.

46206 CX The string may be overwritten by a subsequent call to `strerror_l()` in the same thread.

46207 The contents of the error message strings returned by `strerror()` should be determined by the
 46208 setting of the `LC_MESSAGES` category in the current locale.

46209 The implementation shall behave as if no function defined in this volume of
 46210 IEEE Std 1003.1-200x calls `strerror()`.

46211 CX The `strerror()` and `strerror_l()` functions shall not change the setting of `errno` if successful.

46212 Since no return value is reserved to indicate an error, an application wishing to check for error
 46213 situations should set `errno` to 0, then call `strerror()`, then check `errno`.

46214 The `strerror()` function need not be thread-safe. A function that is not required to be thread-safe
 46215 is not required to be reentrant.

46216 The `strerror_l()` function shall map the error number in `errnum` to a locale-dependent error
 46217 message string in the locale represented by `locale` and shall return a pointer to it.

46218 The `strerror_r()` function shall map the error number in `errnum` to a locale-dependent error
 46219 message string and shall return the string in the buffer pointed to by `strerrbuf`, with length
 46220 `buflen`.

46221 **RETURN VALUE**

46222 Upon completion, whether successful or not, `strerror()` shall return a pointer to the generated
 46223 CX message string. On error `errno` may be set, but no return value is reserved to indicate an error.

46224 Upon successful completion, `strerror_l()` shall return a pointer to the generated message string. If
 46225 `errnum` is not a valid error number, `errno` may be set to [EINVAL], but a pointer to a message
 46226 string shall still be returned. If any other error occurs, `errno` shall be set to indicate the error and
 46227 a null pointer shall be returned.

46228 Upon successful completion, `strerror_r()` shall return 0. Otherwise, an error number shall be
 46229 returned to indicate the error.

strerror()46230 **ERRORS**

46231 These functions may fail if:

46232 CX [EINVAL] The value of *errno* is not a valid error number.46233 The *strerror_l()* function may fail if:46234 CX [EINVAL] The *locale* argument is not a valid locale object handle.46235 The *strerror_r()* function may fail if:46236 CX [ERANGE] Insufficient storage was supplied via *strrbuf* and *buflen* to contain the
46237 generated message string.46238 **EXAMPLES**

46239 None.

46240 **APPLICATION USAGE**

46241 None.

46242 **RATIONALE**46243 The *strerror_l()* function is required to be thread-safe, thereby eliminating the need for an
46244 equivalent to the *strerror_r()* function.46245 **FUTURE DIRECTIONS**

46246 None.

46247 **SEE ALSO**46248 *perror()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**string.h**>46249 **CHANGE HISTORY**

46250 First released in Issue 3.

46251 **Issue 5**46252 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

46253 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

46254 **Issue 6**

46255 Extensions beyond the ISO C standard are marked.

46256 The following new requirements on POSIX implementations derive from alignment with the
46257 Single UNIX Specification:

- 46258
- In the RETURN VALUE section, the fact that *errno* may be set is added.
 - The [EINVAL] optional error condition is added.
- 46259

46260 The normative text is updated to avoid use of the term “must” for application requirements.

46261 The *strerror_r()* function is added in response to IEEE PASC Interpretation 1003.1c #39.46262 The *strerror_r()* function is marked as part of the Thread-Safe Functions option.46263 **Issue 7**

46264 Austin Group Interpretation 1003.1-2001 #072 is applied, updating the ERRORS section.

46265 The *strerror_l()* function is added from The Open Group Technical Standard, 2006, Extended API
46266 Set Part 4.46267 The *strerror_r()* function is moved from the Thread-Safe Functions option to the Base.

46268 **NAME**
 46269 strfmon, strfmon_l — convert monetary value to a string

46270 **SYNOPSIS**
 46271 #include <monetary.h>
 46272 ssize_t strfmon(char *restrict s, size_t maxsize,
 46273 const char *restrict format, ...);
 46274 ssize_t strfmon_l(char *restrict s, size_t maxsize,
 46275 locale_t locale, const char *restrict format, ...);

46276 **DESCRIPTION**
 46277 The *strfmon()* function shall place characters into the array pointed to by *s* as controlled by the
 46278 string pointed to by *format*. No more than *maxsize* bytes are placed into the array.

46279 The format is a character string, beginning and ending in its initial state, if any, that contains two
 46280 types of objects: *plain characters*, which are simply copied to the output stream, and *conversion*
 46281 *specifications*, each of which shall result in the fetching of zero or more arguments which are
 46282 converted and formatted. The results are undefined if there are insufficient arguments for the
 46283 format. If the format is exhausted while arguments remain, the excess arguments are simply
 46284 ignored.

46285 The application shall ensure that a conversion specification consists of the following sequence:

- 46286 • A '%' character
- 46287 • Optional flags
- 46288 • Optional field width
- 46289 • Optional left precision
- 46290 • Optional right precision
- 46291 • A required conversion specifier character that determines the conversion to be performed

46292 The *strfmon_l()* function shall be equivalent to the *strfmon()* function, except that the locale data
 46293 used is from the locale represented by *locale*.

46294 **Flags**

46295 One or more of the following optional flags can be specified to control the conversion:

- 46296 =*f* An '=' followed by a single character *f* which is used as the numeric fill character. In
 46297 order to work with precision or width counts, the fill character shall be a single byte
 46298 character; if not, the behavior is undefined. The default numeric fill character is the
 46299 <space>. This flag does not affect field width filling which always uses the <space>.
 46300 This flag is ignored unless a left precision (see below) is specified.
- 46301 ^ Do not format the currency amount with grouping characters. The default is to insert
 46302 the grouping characters if defined for the current locale.
- 46303 + or (Specify the style of representing positive and negative currency amounts. Only one of
 46304 '+' or '(' may be specified. If '+' is specified, the locale's equivalent of '+' and '-'
 46305 are used (for example, in the U.S., the empty string if positive and '-' if negative). If
 46306 '(' is specified, negative amounts are enclosed within parentheses. If neither flag is
 46307 specified, the '+' style is used.

- 46308 ! Suppress the currency symbol from the output conversion.
- 46309 – Specify the alignment. If this flag is present the result of the conversion is left-justified
46310 (padded to the right) rather than right-justified. This flag shall be ignored unless a field
46311 width (see below) is specified.

46312 Field Width

- 46313 *w* A decimal digit string *w* specifying a minimum field width in bytes in which the result
46314 of the conversion is right-justified (or left-justified if the flag ‘-’ is specified). The
46315 default is 0.

46316 Left Precision

- 46317 #*n* A ‘#’ followed by a decimal digit string *n* specifying a maximum number of digits
46318 expected to be formatted to the left of the radix character. This option can be used to
46319 keep the formatted output from multiple calls to the *strfmon()* function aligned in the
46320 same columns. It can also be used to fill unused positions with a special character as in
46321 "\$***123.45". This option causes an amount to be formatted as if it has the number
46322 of digits specified by *n*. If more than *n* digit positions are required, this conversion
46323 specification is ignored. Digit positions in excess of those actually required are filled
46324 with the numeric fill character (see the =*f* flag above).

46325 If grouping has not been suppressed with the ‘^’ flag, and it is defined for the current
46326 locale, grouping separators are inserted before the fill characters (if any) are added.
46327 Grouping separators are not applied to fill characters even if the fill character is a digit.

46328 To ensure alignment, any characters appearing before or after the number in the
46329 formatted output such as currency or sign symbols are padded as necessary with
46330 <space>s to make their positive and negative formats an equal length.

46331 Right Precision

- 46332 .*p* A period followed by a decimal digit string *p* specifying the number of digits after the
46333 radix character. If the value of the right precision *p* is 0, no radix character appears. If a
46334 right precision is not included, a default specified by the current locale is used. The
46335 amount being formatted is rounded to the specified number of digits prior to
46336 formatting.

46337 Conversion Specifier Characters

46338 The conversion specifier characters and their meanings are:

- 46339 i The **double** argument is formatted according to the locale’s international currency
46340 format (for example, in the U.S.: USD 1,234.56). If the argument is ±Inf or NaN, the
46341 result of the conversion is unspecified.
- 46342 n The **double** argument is formatted according to the locale’s national currency format
46343 (for example, in the U.S.: \$1,234.56). If the argument is ±Inf or NaN, the result of the
46344 conversion is unspecified.
- 46345 % Convert to a ‘%’; no argument is converted. The entire conversion specification shall
46346 be %%.

46347

Locale Information

46348

46349

46350

46351

46352

The *LC_MONETARY* category of the locale of the process affects the behavior of this function including the monetary radix character (which may be different from the numeric radix character affected by the *LC_NUMERIC* category), the grouping separator, the currency symbols, and formats. The international currency symbol should be conformant with the ISO 4217:2001 standard.

46353

If the value of *maxsize* is greater than `{SSIZE_MAX}`, the result is implementation-defined.

46354

RETURN VALUE

46355

46356

46357

46358

If the total number of resulting bytes including the terminating null byte is not more than *maxsize*, these functions shall return the number of bytes placed into the array pointed to by *s*, not including the terminating NUL character. Otherwise, `-1` shall be returned, the contents of the array are unspecified, and *errno* shall be set to indicate the error.

46359

ERRORS

46360

These functions shall fail if:

46361

[E2BIG] Conversion stopped due to lack of space in the buffer.

46362

The *strfmon_l()* function may fail if:

46363

[EINVAL] *locale* is not a valid locale object.

46364

EXAMPLES

46365

46366

Given a locale for the U.S. and the values 123.45, `-123.45`, and `3456.781`, the following output might be produced. Square brackets ("`[]`") are used in this example to delimit the output.

46367

`%n` [`$123.45`] Default formatting

46368

[`-$123.45`]

46369

[`$3,456.78`]

46370

`%11n` [`$123.45`] Right align within an 11-character field

46371

[`-$123.45`]

46372

[`$3,456.78`]

46373

`%#5n` [`$ 123.45`] Aligned columns for values up to 99999

46374

[`-$ 123.45`]

46375

[`$ 3,456.78`]

46376

`%=*#5n` [`$***123.45`] Specify a fill character

46377

[`-$***123.45`]

46378

[`$*3,456.78`]

46379

`%=0#5n` [`$000123.45`] Fill characters do not use grouping

46380

[`-$000123.45`]

46381

[`$03,456.78`]

46382

`%^#5n` [`$ 123.45`] Disable the grouping separator

46383

[`-$ 123.45`]

46384

[`$ 3456.78`]

46385

`%^#5.0n` [`$ 123`] Round off to whole units

46386

[`-$ 123`]

46387

[`$ 3457`]

46388

`%^#5.4n` [`$ 123.4500`] Increase the precision

46389

[`-$ 123.4500`]

46390

[`$ 3456.7810`]

46391

`%(#5n` [`$ 123.45`] Use an alternative pos/neg style

46392

[`($ 123.45)`]

46393 [\$ 3,456.78]

46394 %!(#5n [123.45] Disable the currency symbol

46395 [(123.45)

46396 [3,456.78]

46397 %-14#5.4n [\$ 123.4500] Left-justify the output

46398 [-\$ 123.4500]

46399 [\$ 3,456.7810]

46400 %14#5.4n [\$ 123.4500] Corresponding right-justified output

46401 [-\$ 123.4500]

46402 [\$ 3,456.7810]

46403 See also the EXAMPLES section in *fprintf()*.

APPLICATION USAGE

46404 None.

RATIONALE

46405 None.

FUTURE DIRECTIONS

46406 Lowercase conversion characters are reserved for future standards use and uppercase for implementation-defined use.

SEE ALSO

46407 *fprintf()*, *localeconv()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<monetary.h>**

CHANGE HISTORY

46408 First released in Issue 4.

Issue 5

46409 Moved from ENHANCED I18N to BASE.

46410 The [ENOSYS] error is removed.

46411 A sentence is added to the DESCRIPTION warning about values of *maxsize* that are greater than {SSIZE_MAX}.

Issue 6

46412 The normative text is updated to avoid use of the term “must” for application requirements.

46413 The **restrict** keyword is added to the *strfmon()* prototype for alignment with the ISO/IEC 9899:1999 standard.

46414 The EXAMPLES section is reworked, clarifying the output format.

Issue 7

46415 SD5-XSH-ERN-29 is applied, updating the examples for %(#5n and %!(#5n.

46416 The *strfmon()* function is moved from the XSI option to the Base.

46417 The *strfmon_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

46430 **NAME**
 46431 `strptime, strptime_l` — convert date and time to a string

46432 **SYNOPSIS**
 46433 `#include <time.h>`
 46434 `size_t strptime(char *restrict s, size_t maxsize,`
 46435 `const char *restrict format, const struct tm *restrict timeptr);`
 46436 CX `size_t strptime_l(char *restrict s, size_t maxsize,`
 46437 `const char *restrict format, const struct tm *restrict timeptr,`
 46438 `locale_t locale);`

46439 **DESCRIPTION**
 46440 CX For `strptime()`: The functionality described on this reference page is aligned with the ISO C
 46441 standard. Any conflict between the requirements described here and the ISO C standard is
 46442 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

46443 The `strptime()` function shall place bytes into the array pointed to by `s` as controlled by the string
 46444 pointed to by `format`. The format is a character string, beginning and ending in its initial shift
 46445 state, if any. The `format` string consists of zero or more conversion specifications and ordinary
 46446 characters. A conversion specification consists of a '%' character, possibly followed by an E or O
 46447 modifier, and a terminating conversion specifier character that determines the conversion
 46448 specification's behavior. All ordinary characters (including the terminating NUL character) are
 46449 copied unchanged into the array. If copying takes place between objects that overlap, the
 46450 behavior is undefined. No more than `maxsize` bytes are placed into the array. Each conversion
 46451 specifier is replaced by appropriate characters as described in the following list. The appropriate
 46452 characters are determined using the `LC_TIME` category of the current locale and by the values of
 46453 zero or more members of the broken-down time structure pointed to by `timeptr`, as specified in
 46454 brackets in the description. If any of the specified values are outside the normal range, the
 46455 characters stored are unspecified.

46456 CX The `strptime_l()` function shall be equivalent to the `strptime()` function, except that the locale data
 46457 used is from the locale represented by `locale`.

46458 Local timezone information is used as though `strptime()` called `tzset()`.

46459 The following conversion specifications are supported:

46460 %a Replaced by the locale's abbreviated weekday name. [`tm_wday`]
 46461 %A Replaced by the locale's full weekday name. [`tm_wday`]
 46462 %b Replaced by the locale's abbreviated month name. [`tm_mon`]
 46463 %B Replaced by the locale's full month name. [`tm_mon`]
 46464 %c Replaced by the locale's appropriate date and time representation. (See the Base
 46465 Definitions volume of IEEE Std 1003.1-200x, <**time.h**>.)
 46466 %C Replaced by the year divided by 100 and truncated to an integer, as a decimal number
 46467 [00,99]. [`tm_year`]
 46468 %d Replaced by the day of the month as a decimal number [01,31]. [`tm_mday`]
 46469 %D Equivalent to %m/%d/%y. [`tm_mon, tm_mday, tm_year`]
 46470 %e Replaced by the day of the month as a decimal number [1,31]; a single digit is preceded
 46471 by a space. [`tm_mday`]

strftime()*System Interfaces*

46472	%F	Equivalent to %Y-%m-%d (the ISO 8601:2000 standard date format). [<i>tm_year</i> , <i>tm_mon</i> , <i>tm_mday</i>]
46473		
46474	%g	Replaced by the last 2 digits of the week-based year (see below) as a decimal number [00,99]. [<i>tm_year</i> , <i>tm_wday</i> , <i>tm_yday</i>]
46475		
46476	%G	Replaced by the week-based year (see below) as a decimal number (for example, 1977). [<i>tm_year</i> , <i>tm_wday</i> , <i>tm_yday</i>]
46477		
46478	%h	Equivalent to %b. [<i>tm_mon</i>]
46479	%H	Replaced by the hour (24-hour clock) as a decimal number [00,23]. [<i>tm_hour</i>]
46480	%I	Replaced by the hour (12-hour clock) as a decimal number [01,12]. [<i>tm_hour</i>]
46481	%j	Replaced by the day of the year as a decimal number [001,366]. [<i>tm_yday</i>]
46482	%m	Replaced by the month as a decimal number [01,12]. [<i>tm_mon</i>]
46483	%M	Replaced by the minute as a decimal number [00,59]. [<i>tm_min</i>]
46484	%n	Replaced by a <newline>.
46485	%p	Replaced by the locale's equivalent of either a.m. or p.m. [<i>tm_hour</i>]
46486	CX %r	Replaced by the time in a.m. and p.m. notation; in the POSIX locale this shall be equivalent to %I:%M:%S %p. [<i>tm_hour</i> , <i>tm_min</i> , <i>tm_sec</i>]
46487		
46488	%R	Replaced by the time in 24-hour notation (%H:%M). [<i>tm_hour</i> , <i>tm_min</i>]
46489	%S	Replaced by the second as a decimal number [00,60]. [<i>tm_sec</i>]
46490	%t	Replaced by a <tab>.
46491	%T	Replaced by the time (%H:%M:%S). [<i>tm_hour</i> , <i>tm_min</i> , <i>tm_sec</i>]
46492	%u	Replaced by the weekday as a decimal number [1,7], with 1 representing Monday. [<i>tm_wday</i>]
46493		
46494	%U	Replaced by the week number of the year as a decimal number [00,53]. The first Sunday of January is the first day of week 1; days in the new year before this are in week 0. [<i>tm_year</i> , <i>tm_wday</i> , <i>tm_yday</i>]
46495		
46496		
46497	%V	Replaced by the week number of the year (Monday as the first day of the week) as a decimal number [01,53]. If the week containing 1 January has four or more days in the new year, then it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1. Both January 4th and the first Thursday of January are always in week 1. [<i>tm_year</i> , <i>tm_wday</i> , <i>tm_yday</i>]
46498		
46499		
46500		
46501		
46502	%w	Replaced by the weekday as a decimal number [0,6], with 0 representing Sunday. [<i>tm_wday</i>]
46503		
46504	%W	Replaced by the week number of the year as a decimal number [00,53]. The first Monday of January is the first day of week 1; days in the new year before this are in week 0. [<i>tm_year</i> , <i>tm_wday</i> , <i>tm_yday</i>]
46505		
46506		
46507	%x	Replaced by the locale's appropriate date representation. (See the Base Definitions volume of IEEE Std 1003.1-200x, <time.h>.)
46508		
46509	%X	Replaced by the locale's appropriate time representation. (See the Base Definitions volume of IEEE Std 1003.1-200x, <time.h>.)
46510		
46511	%y	Replaced by the last two digits of the year as a decimal number [00,99]. [<i>tm_year</i>]

46512	%Y	Replaced by the year as a decimal number (for example, 1997). [<i>tm_year</i>]
46513	%z	Replaced by the offset from UTC in the ISO 8601:2000 standard format (+hhmm or -hhmm), or by no characters if no timezone is determinable. For example, "-0430"
46514		means 4 hours 30 minutes behind UTC (west of Greenwich). If <i>tm_isdst</i> is zero, the
46515	CX	standard time offset is used. If <i>tm_isdst</i> is greater than zero, the daylight savings time
46516		offset is used. If <i>tm_isdst</i> is negative, no characters are returned. [<i>tm_isdst</i>]
46517		
46518	%Z	Replaced by the timezone name or abbreviation, or by no bytes if no timezone
46519		information exists. [<i>tm_isdst</i>]
46520	%%	Replaced by %.
46521		If a conversion specification does not correspond to any of the above, the behavior is undefined.
46522	CX	If a struct tm broken-down time structure is created by <i>localtime()</i> or <i>localtime_r()</i> , or modified
46523		by <i>mktime()</i> , and the value of <i>TZ</i> is subsequently modified, the results of the %Z and %z
46524		<i>strptime()</i> conversion specifiers are undefined, when <i>strptime()</i> is called with such a broken-down
46525		time structure.
46526		If a struct tm broken-down time structure is created or modified by <i>gmtime()</i> or <i>gmtime_r()</i> , it is
46527		unspecified whether the result of the %Z and %z conversion specifiers shall refer to UTC or the
46528		current local timezone, when <i>strptime()</i> is called with such a broken-down time structure.

46529 Modified Conversion Specifiers

46530		Some conversion specifiers can be modified by the E or O modifier characters to indicate that an
46531		alternative format or specification should be used rather than the one normally used by the
46532		unmodified conversion specifier. If the alternative format or specification does not exist for the
46533		current locale (see ERA in the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3.5,
46534		LC_TIME), the behavior shall be as if the unmodified conversion specification were used.
46535	%EC	Replaced by the locale's alternative appropriate date and time representation.
46536	%EC	Replaced by the name of the base year (period) in the locale's alternative
46537		representation.
46538	%Ex	Replaced by the locale's alternative date representation.
46539	%EX	Replaced by the locale's alternative time representation.
46540	%EY	Replaced by the offset from %EC (year only) in the locale's alternative representation.
46541	%EY	Replaced by the full alternative year representation.
46542	%Od	Replaced by the day of the month, using the locale's alternative numeric symbols, filled
46543		as needed with leading zeros if there is any alternative symbol for zero; otherwise, with
46544		leading spaces.
46545	%Oe	Replaced by the day of the month, using the locale's alternative numeric symbols, filled
46546		as needed with leading spaces.
46547	%OH	Replaced by the hour (24-hour clock) using the locale's alternative numeric symbols.
46548	%OI	Replaced by the hour (12-hour clock) using the locale's alternative numeric symbols.
46549	%Om	Replaced by the month using the locale's alternative numeric symbols.
46550	%OM	Replaced by the minutes using the locale's alternative numeric symbols.
46551	%OS	Replaced by the seconds using the locale's alternative numeric symbols.
46552	%Ou	Replaced by the weekday as a number in the locale's alternative representation
46553		(Monday=1).

46554 %OU Replaced by the week number of the year (Sunday as the first day of the week, rules
46555 corresponding to %U) using the locale's alternative numeric symbols.

46556 %OV Replaced by the week number of the year (Monday as the first day of the week, rules
46557 corresponding to %V) using the locale's alternative numeric symbols.

46558 %Ow Replaced by the number of the weekday (Sunday=0) using the locale's alternative
46559 numeric symbols.

46560 %OW Replaced by the week number of the year (Monday as the first day of the week) using
46561 the locale's alternative numeric symbols.

46562 %Oy Replaced by the year (offset from %C) using the locale's alternative numeric symbols.

46563 %g, %G, and %V give values according to the ISO 8601:2000 standard week-based year. In this
46564 system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th,
46565 which is also the week that includes the first Thursday of the year, and is also the first week that
46566 contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the
46567 preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January
46568 1999, %G is replaced by 1998 and %V is replaced by 53. If December 29th, 30th, or 31st is a
46569 Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday
46570 30th December 1997, %G is replaced by 1998 and %V is replaced by 01.

46571 If a conversion specifier is not one of the above, the behavior is undefined.

RETURN VALUE

46572 If the total number of resulting bytes including the terminating null byte is not more than
46573 *maxsize*, these functions shall return the number of bytes placed into the array pointed to by *s*,
46574 not including the terminating NUL character. Otherwise, 0 shall be returned and the contents of
46575 the array are unspecified.
46576

ERRORS

46577 The *strftime_l()* function may fail if:

46578 CX [EINVAL] *locale* is not a valid locale object handle.

EXAMPLES**Getting a Localized Date String**

46581 The following example first sets the locale to the user's default. The locale information will be
46582 used in the *nl_langinfo()* and *strftime()* functions. The *nl_langinfo()* function returns the localized
46583 date string which specifies how the date is laid out. The *strftime()* function takes this
46584 information and, using the **tm** structure for values, places the date and time information into
46585 *datestring*.
46586

```
46587       #include <time.h>
46588       #include <locale.h>
46589       #include <langinfo.h>
46590       ...
46591       struct tm *tm;
46592       char datestring[256];
46593       ...
46594       setlocale (LC_ALL, "");
46595       ...
46596       strftime (datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm);
46597       ...
```

APPLICATION USAGE

The range of values for %S is [00,60] rather than [00,59] to allow for the occasional leap second.

Some of the conversion specifications are duplicates of others. They are included for compatibility with *nl_cxtime()* and *nl_ascxtime()*, which were published in Issue 2.

Applications should use %Y (4-digit years) in preference to %y (2-digit years).

In the C locale, the E and O modifiers are ignored and the replacement strings for the following specifiers are:

%a	The first three characters of %A.
%A	One of Sunday, Monday, . . . , Saturday.
%b	The first three characters of %B.
%B	One of January, February, . . . , December.
%c	Equivalent to %a %b %e %T %Y.
%p	One of AM or PM.
%r	Equivalent to %I:%M:%S %p.
%x	Equivalent to %m/%d/%y.
%X	Equivalent to %T.
%Z	Implementation-defined.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asctime(), *clock()*, *ctime()*, *difftime()*, *getdate()*, *gmtime()*, *localtime()*, *mktime()*, *strptime()*, *time()*, *tzset()*, *uselocale()*, *utime()*, Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3.5, LC_TIME, <time.h>

CHANGE HISTORY

First released in Issue 3.

Issue 5

The description of %OV is changed to be consistent with %V and defines Monday as the first day of the week.

The description of %Oy is clarified.

Issue 6

Extensions beyond the ISO C standard are marked.

The Open Group Corrigendum U033/8 is applied. The %V conversion specifier is changed from “Otherwise, it is week 53 of the previous year, and the next week is week 1” to “Otherwise, it is the last week of the previous year, and the next week is week 1”.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The %C, %D, %e, %h, %n, %r, %R, %t, and %T conversion specifiers are added.
- The modified conversion specifiers are added for consistency with the ISO POSIX-2 standard *date* utility.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

46640

- The *strftime()* prototype is updated.

46641

- The DESCRIPTION is extensively revised.

46642

- The %z conversion specifier is added.

46643

A new example is added.

46644

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/60 is applied.

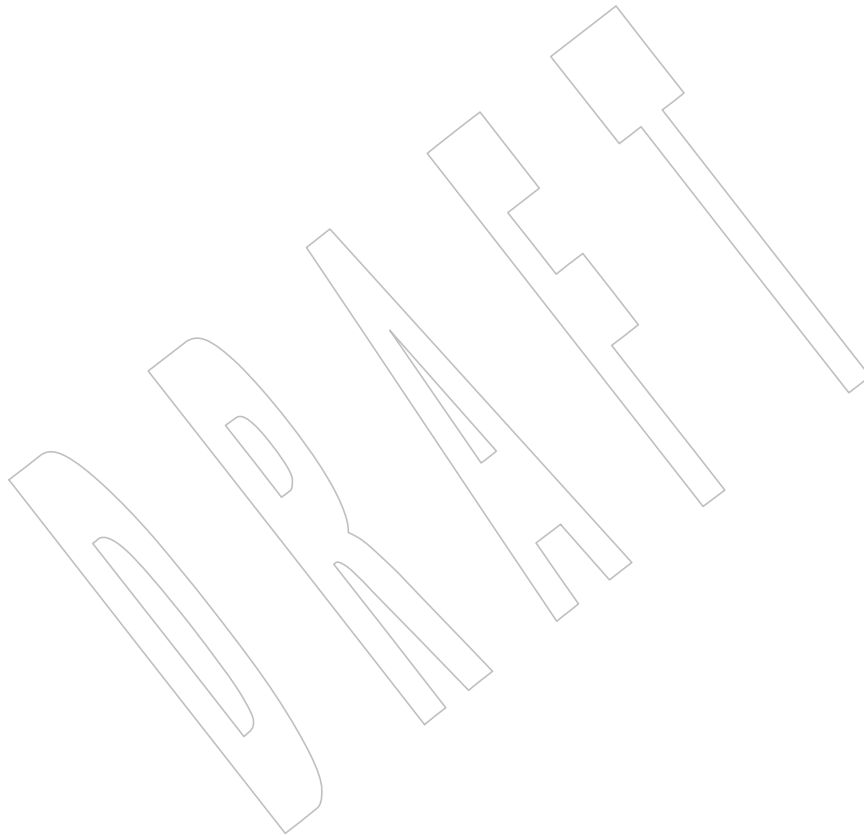
46645

Issue 7

46646

The *strftime_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

46647



46648 **NAME**
 46649 `strlen, strlen` — get length of fixed size string

46650 **SYNOPSIS**

46651 `#include <string.h>`
 46652 `size_t strlen(const char *s);`
 46653 CX `size_t strlen(const char *s, size_t maxlen);`

46654 **DESCRIPTION**

46655 CX For `strlen()`: The functionality described on this reference page is aligned with the ISO C
 46656 standard. Any conflict between the requirements described here and the ISO C standard is
 46657 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

46658 The `strlen()` function shall compute the number of bytes in the string to which `s` points, not
 46659 including the terminating NUL character.

46660 CX The `strlen()` function shall compute the smaller of the number of bytes in the string to which `s`
 46661 points, not including the terminating NUL character, or the value of the `maxlen` argument. The
 46662 `strlen()` function shall never examine more than `maxlen` bytes of the string pointed to by `s`.

46663 **RETURN VALUE**

46664 The `strlen()` function shall return the length of `s`; no return value shall be reserved to indicate an
 46665 error.

46666 CX The `strlen()` function shall return an integer containing the smaller of either the length of the
 46667 string pointed to by `s` or `maxlen`.

46668 **ERRORS**

46669 No errors are defined.

46670 **EXAMPLES**

46671 **Getting String Lengths**

46672 The following example sets the maximum length of `key` and `data` by using `strlen()` to get the
 46673 lengths of those strings.

```
46674 #include <string.h>
46675 ...
46676 struct element {
46677     char *key;
46678     char *data;
46679 };
46680 ...
46681 char *key, *data;
46682 int len;
46683
46684 *keylength = *datalength = 0;
46685 ...
46686 if ((len = strlen(key)) > *keylength)
46687     *keylength = len;
46688 if ((len = strlen(data)) > *datalength)
46689     *datalength = len;
46690 ...
```

46690 APPLICATION USAGE

46691 None.

46692 RATIONALE

46693 None.

46694 FUTURE DIRECTIONS

46695 None.

46696 SEE ALSO

46697 *wcslen()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<string.h>`

46698 CHANGE HISTORY

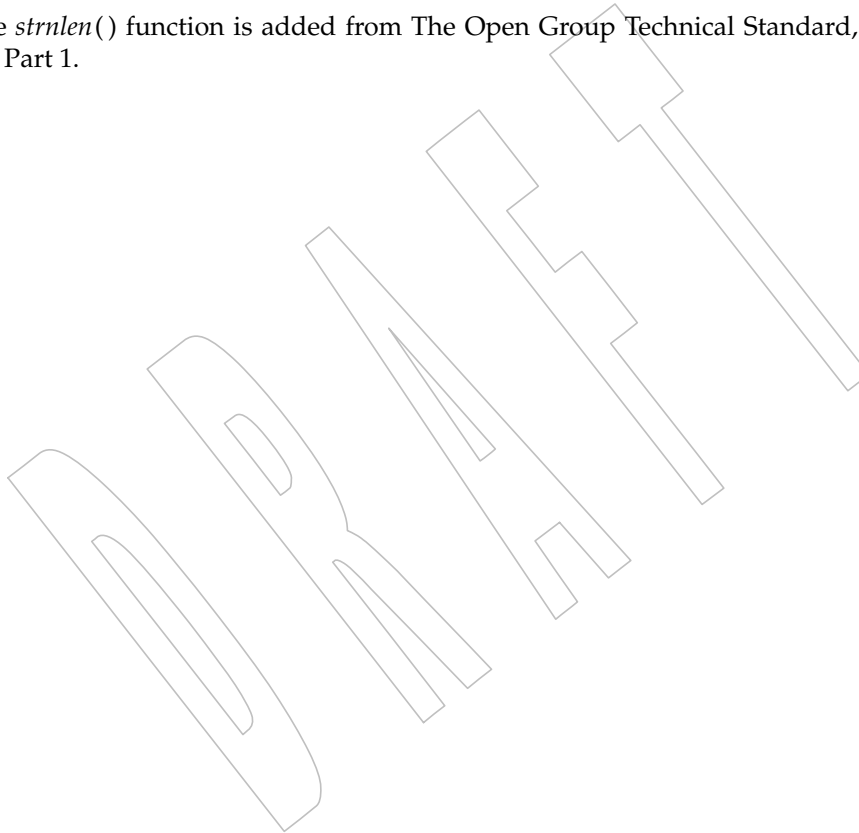
46699 First released in Issue 1. Derived from Issue 1 of the SVID.

46700 Issue 5

46701 The RETURN VALUE section is updated to indicate that *strlen()* returns the length of *s*, and not
46702 *s* itself as was previously stated.

46703 Issue 7

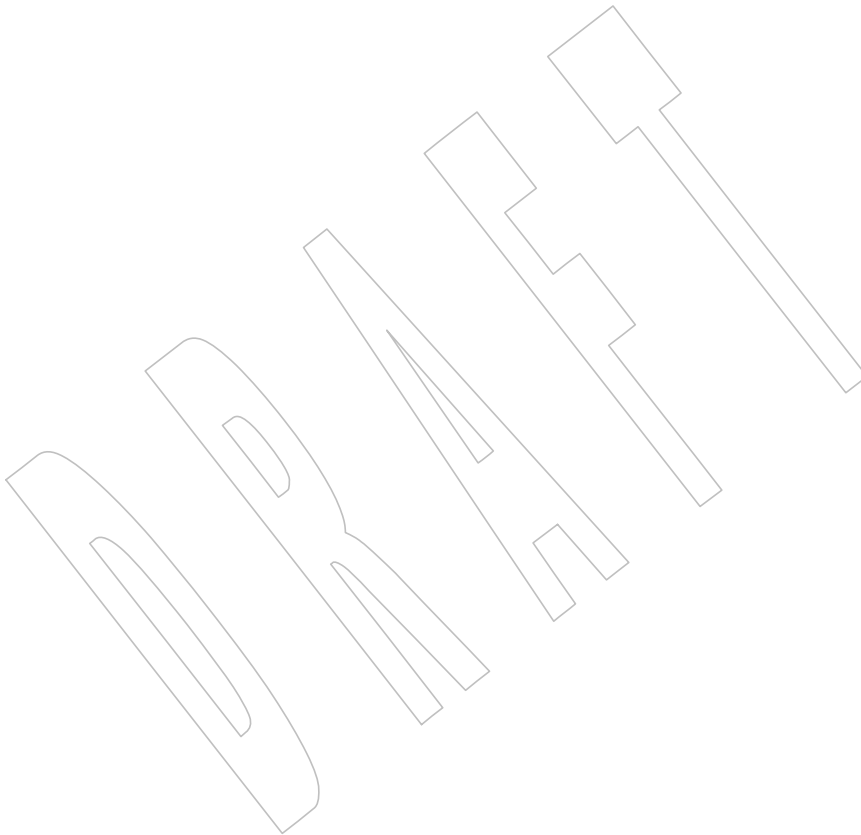
46704 The *strnlen()* function is added from The Open Group Technical Standard, 2006, Extended API
46705 Set Part 1.



46706 **NAME**
46707 `strncasecmp, strncasecmp_l` — case-insensitive string comparisons

46708 **SYNOPSIS**
46709 `#include <strings.h>`
46710 `int strncasecmp(const char *s1, const char *s2, size_t n);`
46711 `int strncasecmp_l(const char *s1, const char *s2,`
46712 `size_t n, locale_t locale);`

46713 **DESCRIPTION**
46714 Refer to *strcasecmp()*.



46715 **NAME**
 46716 `strncat` — concatenate a string with part of another

46717 **SYNOPSIS**
 46718 `#include <string.h>`

46719 `char *strncat(char *restrict s1, const char *restrict s2, size_t n);`

46720 **DESCRIPTION**

46721 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 46722 conflict between the requirements described here and the ISO C standard is unintentional. This
 46723 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

46724 The `strncat()` function shall append not more than *n* bytes (a NUL character and bytes that
 46725 follow it are not appended) from the array pointed to by *s2* to the end of the string pointed to by
 46726 *s1*. The initial byte of *s2* overwrites the NUL character at the end of *s1*. A terminating NUL
 46727 character is always appended to the result. If copying takes place between objects that overlap,
 46728 the behavior is undefined.

46729 **RETURN VALUE**

46730 The `strncat()` function shall return *s1*; no return value shall be reserved to indicate an error.

46731 **ERRORS**

46732 No errors are defined.

46733 **EXAMPLES**

46734 None.

46735 **APPLICATION USAGE**

46736 None.

46737 **RATIONALE**

46738 None.

46739 **FUTURE DIRECTIONS**

46740 None.

46741 **SEE ALSO**

46742 `strcat()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<string.h>`

46743 **CHANGE HISTORY**

46744 First released in Issue 1. Derived from Issue 1 of the SVID.

46745 **Issue 6**

46746 The `strncat()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

46747 **NAME**
 46748 `strncmp` — compare part of two strings

46749 **SYNOPSIS**
 46750 `#include <string.h>`

46751 `int strncmp(const char *s1, const char *s2, size_t n);`

46752 **DESCRIPTION**

46753 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 46754 conflict between the requirements described here and the ISO C standard is unintentional. This
 46755 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

46756 The `strncmp()` function shall compare not more than *n* bytes (bytes that follow a NUL character
 46757 are not compared) from the array pointed to by *s1* to the array pointed to by *s2*.

46758 The sign of a non-zero return value is determined by the sign of the difference between the
 46759 values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the strings
 46760 being compared.

46761 **RETURN VALUE**

46762 Upon successful completion, `strncmp()` shall return an integer greater than, equal to, or less than
 46763 0, if the possibly null-terminated array pointed to by *s1* is greater than, equal to, or less than the
 46764 possibly null-terminated array pointed to by *s2* respectively.

46765 **ERRORS**

46766 No errors are defined.

46767 **EXAMPLES**

46768 None.

46769 **APPLICATION USAGE**

46770 None.

46771 **RATIONALE**

46772 None.

46773 **FUTURE DIRECTIONS**

46774 None.

46775 **SEE ALSO**

46776 `strcmp()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<string.h>`

46777 **CHANGE HISTORY**

46778 First released in Issue 1. Derived from Issue 1 of the SVID.

46779 **Issue 6**

46780 Extensions beyond the ISO C standard are marked.

46781 **NAME**
 46782 `stpncpy`, `strncpy` — copy fixed length string, returning a pointer to the array end

46783 **SYNOPSIS**
 46784 `#include <string.h>`

46785 CX `char *stpncpy(char *restrict s1, const char *restrict s2, size_t n);`
 46786 `char *strncpy(char *restrict s1, const char *restrict s2, size_t n);`

46787 **DESCRIPTION**

46788 CX For `strncpy()`: The functionality described on this reference page is aligned with the ISO C
 46789 standard. Any conflict between the requirements described here and the ISO C standard is
 46790 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

46791 CX The `stpncpy()` and `strncpy()` functions shall copy not more than n bytes (bytes that follow a NUL
 46792 character are not copied) from the array pointed to by $s2$ to the array pointed to by $s1$.

46793 If the array pointed to by $s2$ is a string that is shorter than n bytes, NUL characters shall be
 46794 appended to the copy in the array pointed to by $s1$, until n bytes in all are written.

46795 If copying takes place between objects that overlap, the behavior is undefined.

46796 **RETURN VALUE**

46797 CX If a NUL character is written to the destination, the `stpncpy()` function shall return the address of
 46798 the first such NUL character. Otherwise, it shall return `&s2[n]`.

46799 The `strncpy()` function shall return $s1$.

46800 No return values are reserved to indicate an error.

46801 **ERRORS**

46802 No errors are defined.

46803 **EXAMPLES**

46804 None.

46805 **APPLICATION USAGE**

46806 Applications must provide the space in $s1$ for the n bytes to be transferred, as well as ensure that
 46807 the $s2$ and $s1$ arrays do not overlap.

46808 Character movement is performed differently in different implementations. Thus, overlapping
 46809 moves may yield surprises.

46810 If there is no NUL character byte in the first n bytes of the array pointed to by $s2$, the result is not
 46811 null-terminated.

46812 **RATIONALE**

46813 None.

46814 **FUTURE DIRECTIONS**

46815 None.

46816 **SEE ALSO**

46817 `strcpy()`, `wcsncpy()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<string.h>`

46818 **CHANGE HISTORY**

46819 First released in Issue 1. Derived from Issue 1 of the SVID.

46820

Issue 6

46821

The *strncpy()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

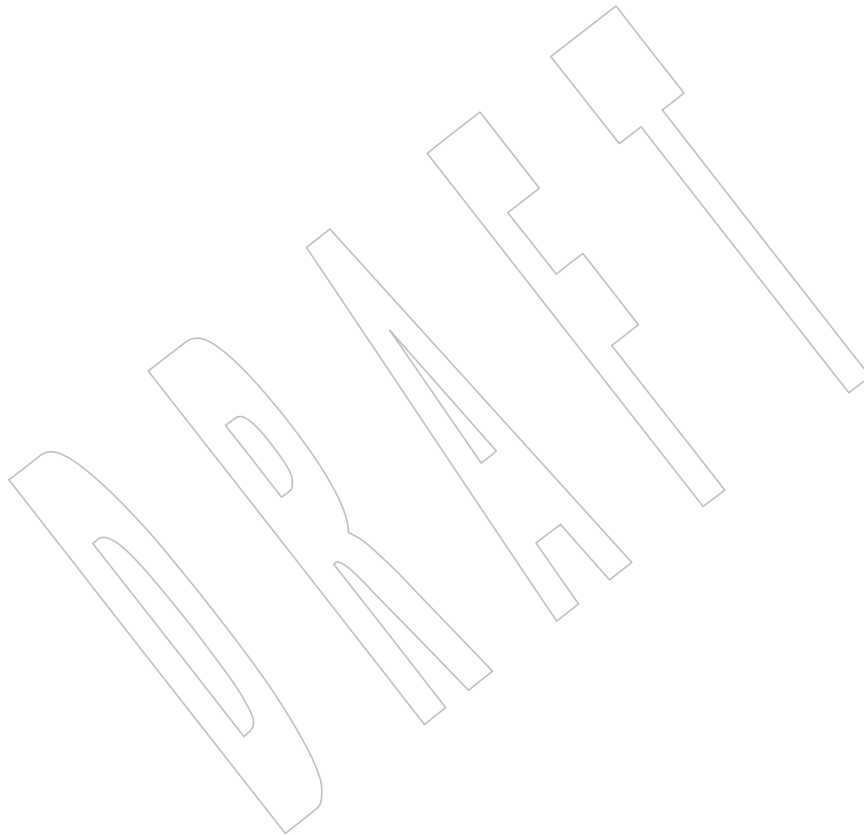
46822

Issue 7

46823

The *stpncpy()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

46824



strndup()

46825 **NAME**
46826 `strndup` — duplicate a specific number of bytes from a string

46827 **SYNOPSIS**

46828 CX `#include <string.h>`
46829 `char *strndup(const char *s, size_t size);`

46830 **DESCRIPTION**

46831 Refer to *strdup()*.

46832 **NAME**
46833 `strnlen` — get length of fixed size string

46834 **SYNOPSIS**

46835 CX `#include <string.h>`
46836 `size_t strnlen(const char *s, size_t maxlen);`

46837 **DESCRIPTION**

46838 Refer to *strlen()*.

46839 **NAME**
 46840 `strpbrk` — scan a string for a byte

46841 **SYNOPSIS**
 46842 `#include <string.h>`
 46843 `char *strpbrk(const char *s1, const char *s2);`

46844 **DESCRIPTION**
 46845 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 46846 conflict between the requirements described here and the ISO C standard is unintentional. This
 46847 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

46848 The `strpbrk()` function shall locate the first occurrence in the string pointed to by `s1` of any byte
 46849 from the string pointed to by `s2`.

46850 **RETURN VALUE**
 46851 Upon successful completion, `strpbrk()` shall return a pointer to the byte or a null pointer if no
 46852 byte from `s2` occurs in `s1`.

46853 **ERRORS**
 46854 No errors are defined.

46855 **EXAMPLES**
 46856 None.

46857 **APPLICATION USAGE**
 46858 None.

46859 **RATIONALE**
 46860 None.

46861 **FUTURE DIRECTIONS**
 46862 None.

46863 **SEE ALSO**
 46864 `strchr()`, `strrchr()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<string.h>`

46865 **CHANGE HISTORY**
 46866 First released in Issue 1. Derived from Issue 1 of the SVID.

46867 **NAME**

46868 strptime — date and time conversion

46869 **SYNOPSIS**

```
46870 XSI #include <time.h>
46871 char *strptime(const char *restrict buf, const char *restrict format,
46872 struct tm *restrict tm);
```

46873 **DESCRIPTION**

46874 The *strptime()* function shall convert the character string pointed to by *buf* to values which are
 46875 stored in the **tm** structure pointed to by *tm*, using the format specified by *format*.

46876 The *format* is composed of zero or more directives. Each directive is composed of one of the
 46877 following: one or more white-space characters (as specified by *isspace()*); an ordinary character
 46878 (neither '%' nor a white-space character); or a conversion specification. Each conversion
 46879 specification is composed of a '%' character followed by a conversion character which specifies
 46880 the replacement required. The conversions are determined using the *LC_TIME* category of the
 46881 current locale. The application shall ensure that there is white-space or other non-alphanumeric
 46882 characters between any two conversion specifications. In the following list, where numeric
 46883 ranges of values are given (represented by the pattern [*x*,*y*]), the value shall fall within the
 46884 range given (both bounds being inclusive), and the number of characters scanned (excluding the
 46885 one matching the next directive) shall be no more than the maximum number required to
 46886 represent any value in the range without leading zeros. The following conversion specifications
 46887 are supported:

46888	%a	The day of the week, using the locale's weekday names; either the abbreviated or full name may be specified.
46889		
46890	%A	Equivalent to %a.
46891	%b	The month, using the locale's month names; either the abbreviated or full name may be specified.
46892		
46893	%B	Equivalent to %b.
46894	%c	Replaced by the locale's appropriate date and time representation.
46895	%C	The century number [00,99]; leading zeros shall be permitted but shall not be required.
46896	%d	The day of the month [01,31]; leading zeros shall be permitted but shall not be required.
46897	%D	The date as %m/%d/%y.
46898	%e	Equivalent to %d.
46899	%h	Equivalent to %b.
46900	%H	The hour (24-hour clock) [00,23]; leading zeros shall be permitted but shall not be required.
46901		
46902	%I	The hour (12-hour clock) [01,12]; leading zeros shall be permitted but shall not be required.
46903		
46904	%j	The day number of the year [001,366]; leading zeros shall be permitted but shall not be required.
46905		
46906	%m	The month number [01,12]; leading zeros shall be permitted but shall not be required.

46907	%M	The minute [00,59]; leading zeros shall be permitted but shall not be required.
46908	%n	Any white space.
46909	%p	The locale's equivalent of a.m. or p.m.
46910	%r	12-hour clock time using the AM/PM notation if t_fmt_ampm is not an empty string in the <i>LC_TIME</i> portion of the current locale; in the POSIX locale, this shall be equivalent to %I:%M:%S %p.
46911		
46912		
46913	%R	The time as %H:%M.
46914	%S	The seconds [00,60]; leading zeros shall be permitted but shall not be required.
46915	%t	Any white space.
46916	%T	The time as %H:%M:%S.
46917	%U	The week number of the year (Sunday as the first day of the week) as a decimal number [00,53]; leading zeros shall be permitted but shall not be required.
46918		
46919	%w	The weekday as a decimal number [0,6], with 0 representing Sunday.
46920	%W	The week number of the year (Monday as the first day of the week) as a decimal number [00,53]; leading zeros shall be permitted but shall not be required.
46921		
46922	%x	The date, using the locale's date format.
46923	%X	The time, using the locale's time format.
46924	%y	The year within century. When a century is not otherwise specified, values in the range [69,99] shall refer to years 1969 to 1999 inclusive, and values in the range [00,68] shall refer to years 2000 to 2068 inclusive; leading zeros shall be permitted but shall not be required.
46925		
46926		
46927		
46928	Note:	It is expected that in a future version of IEEE Std 1003.1-200x the default century inferred from a 2-digit year will change. (This would apply to all commands accepting a 2-digit year as input.)
46929		
46930		
46931	%Y	The year, including the century (for example, 1988).
46932	%%	Replaced by %.

Modified Conversion Specifiers

46934		Some conversion specifiers can be modified by the E and O modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specifier. If the alternative format or specification does not exist in the current locale, the behavior shall be as if the unmodified conversion specification were used.
46935		
46936		
46937		
46938	%Ec	The locale's alternative appropriate date and time representation.
46939	%EC	The name of the base year (period) in the locale's alternative representation.
46940	%Ex	The locale's alternative date representation.
46941	%EX	The locale's alternative time representation.
46942	%Ey	The offset from %EC (year only) in the locale's alternative representation.
46943	%EY	The full alternative year representation.
46944	%Od	The day of the month using the locale's alternative numeric symbols; leading zeros shall be permitted but shall not be required.
46945		

46946	%Oe	Equivalent to %Od.
46947	%OH	The hour (24-hour clock) using the locale's alternative numeric symbols.
46948	%OI	The hour (12-hour clock) using the locale's alternative numeric symbols.
46949	%Om	The month using the locale's alternative numeric symbols.
46950	%OM	The minutes using the locale's alternative numeric symbols.
46951	%OS	The seconds using the locale's alternative numeric symbols.
46952	%OU	The week number of the year (Sunday as the first day of the week) using the locale's alternative numeric symbols.
46953		
46954	%Ow	The number of the weekday (Sunday=0) using the locale's alternative numeric symbols.
46955		
46956	%OW	The week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.
46957		
46958	%Oy	The year (offset from %C) using the locale's alternative numeric symbols.
46959		A conversion specification composed of white-space characters is executed by scanning input up to the first character that is not white-space (which remains unscanned), or until no more characters can be scanned.
46960		
46961		
46962		A conversion specification that is an ordinary character is executed by scanning the next character from the buffer. If the character scanned from the buffer differs from the one comprising the directive, the directive fails, and the differing and subsequent characters remain unscanned.
46963		
46964		
46965		
46966		A series of conversion specifications composed of %n, %t, white-space characters, or any combination is executed by scanning up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned.
46967		
46968		
46969		Any other conversion specification is executed by scanning characters until a character matching the next directive is scanned, or until no more characters can be scanned. These characters, except the one matching the next directive, are then compared to the locale values associated with the conversion specifier. If a match is found, values for the appropriate tm structure members are set to values corresponding to the locale information. Case is ignored when matching items in <i>buf</i> such as month or weekday names. If no match is found, <i>strptime()</i> fails and no more characters are scanned.
46970		
46971		
46972		
46973		
46974		
46975		

RETURN VALUE

Upon successful completion, *strptime()* shall return a pointer to the character following the last character parsed. Otherwise, a null pointer shall be returned.

ERRORS

No errors are defined.

EXAMPLES**Convert a Data-Plus-Time String to Broken-Down Time and Then into Seconds**

The following example demonstrates the use of *strptime()* to convert a string into broken-down time. The broken-down time is then converted into seconds since the Epoch using *mktime()*.

```
#include <time.h>
...

struct tm tm;
time_t t;
```

```

46989     if (strptime("6 Dec 2001 12:33:45", "%d %b %Y %H:%M:%S", &tm) == NULL)
46990         /* Handle error */;

46991     printf("year: %d; month: %d; day: %d;\n",
46992           tm.tm_year, tm.tm_mon, tm.tm_mday);
46993     printf("hour: %d; minute: %d; second: %d\n",
46994           tm.tm_hour, tm.tm_min, tm.tm_sec);
46995     printf("week day: %d; year day: %d\n", tm.tm_wday, tm.tm_yday);

46996     tm.tm_isdst = -1;      /* Not set by strptime(); tells mktime()
46997                           to determine whether daylight saving time
46998                           is in effect */

46999     t = mktime(&tm);
47000     if (t == -1)
47001         /* Handle error */;
47002     printf("seconds since the Epoch: %ld\n", (long) t);

```

APPLICATION USAGE

Several “equivalent to” formats and the special processing of white-space characters are provided in order to ease the use of identical *format* strings for *strptime()* and *strptime()*.

It should be noted that dates constructed by the *strptime()* function with the %Y or %C%y conversion specifiers may have values larger than 9999. If the *strptime()* function is used to read this value using %C%y, these values will be truncated to four digits.

Applications should use %Y (year including century) in preference to %y (2-digit years).

It is unspecified whether multiple calls to *strptime()* using the same **tm** structure will update the current contents of the structure or overwrite all contents of the structure. Conforming applications should make a single call to *strptime()* with a format and all data needed to completely specify the date and time being converted.

RATIONALE

None.

FUTURE DIRECTIONS

The *strptime()* function is expected to be mandatory in the next version of this volume of IEEE Std 1003.1-200x.

SEE ALSO

scanf(), *strptime()*, *time()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**time.h**>

CHANGE HISTORY

First released in Issue 4.

Issue 5

Moved from ENHANCED I18N to BASE.

The [ENOSYS] error is removed.

The exact meaning of the %y and %Oy specifiers is clarified in the DESCRIPTION.

Issue 6

The Open Group Corrigendum U033/5 is applied. The %r specifier description is reworded.

The normative text is updated to avoid use of the term “must” for application requirements.

The **restrict** keyword is added to the *strptime()* prototype for alignment with the ISO/IEC 9899:1999 standard.

The Open Group Corrigendum U047/2 is applied.

The DESCRIPTION is updated to use the terms “conversion specifier” and “conversion

47034

specification” for consistency with *strptime()*.

47035

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/133 is applied, adding the example to the EXAMPLES section.

47036

47037

Issue 7

47038

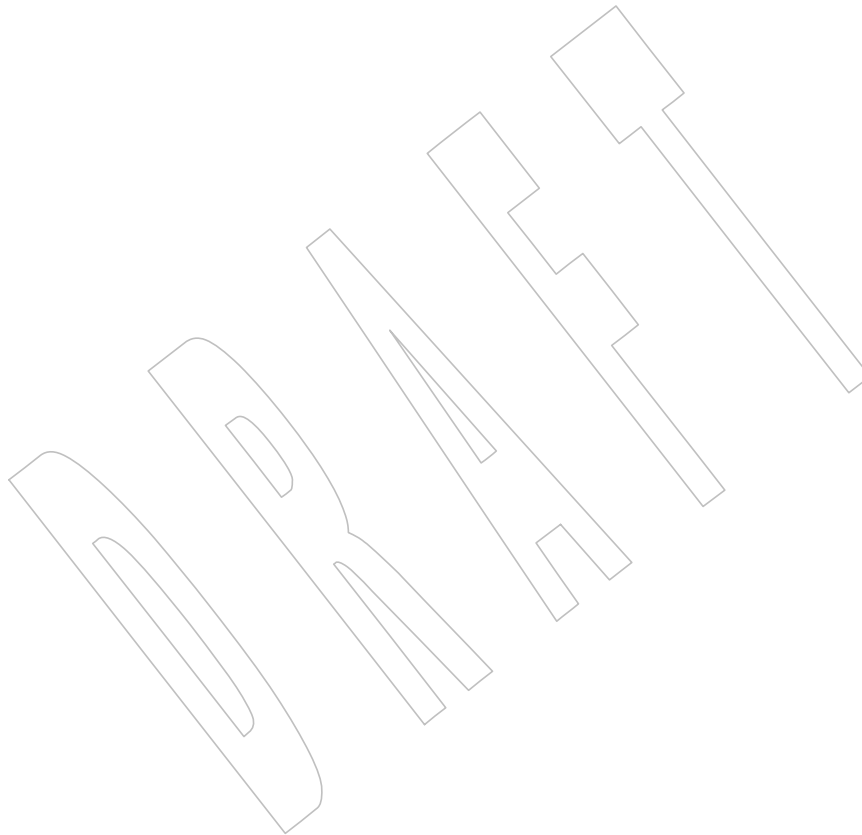
SD5-XSH-ERN-67 is applied, correcting the APPLICATION USAGE to remove the impression that %Y is four digit years.

47039

47040

Austin Group Interpretation 1003.1-2001 #041 is applied, updating the DESCRIPTION and APPLICATION USAGE sections.

47041



47042 **NAME**47043 `strrchr` — string scanning operation47044 **SYNOPSIS**47045 `#include <string.h>`47046 `char *strrchr(const char *s, int c);`47047 **DESCRIPTION**

47048 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 47049 conflict between the requirements described here and the ISO C standard is unintentional. This
 47050 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47051 The `strrchr()` function shall locate the last occurrence of `c` (converted to a **char**) in the string
 47052 pointed to by `s`. The terminating NUL character is considered to be part of the string.

47053 **RETURN VALUE**

47054 Upon successful completion, `strrchr()` shall return a pointer to the byte or a null pointer if `c` does
 47055 not occur in the string.

47056 **ERRORS**

47057 No errors are defined.

47058 **EXAMPLES**47059 **Finding the Base Name of a File**

47060 The following example uses `strrchr()` to get a pointer to the base name of a file. The `strrchr()`
 47061 function searches backwards through the name of the file to find the last `'/'` character in `name`.
 47062 This pointer (plus one) will point to the base name of the file.

```
47063 #include <string.h>
47064 ...
47065 const char *name;
47066 char *basename;
47067 ...
47068 basename = strrchr(name, '/') + 1;
47069 ...
```

47070 **APPLICATION USAGE**

47071 None.

47072 **RATIONALE**

47073 None.

47074 **FUTURE DIRECTIONS**

47075 None.

47076 **SEE ALSO**47077 `strchr()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<string.h>`47078 **CHANGE HISTORY**

47079 First released in Issue 1. Derived from Issue 1 of the SVID.

47080 **NAME**
 47081 strsignal — get name of signal

47082 **SYNOPSIS**

47083 CX `#include <string.h>`
 47084 `char *strsignal(int signum);`

47085 **DESCRIPTION**

47086 The *strsignal()* function shall map the signal number in *signum* to an implementation-defined
 47087 string and shall return a pointer to it. It shall use the same set of messages as the *psignal()*
 47088 function.

47089 The string pointed to shall not be modified by the application, but may be overwritten by a
 47090 subsequent call to *strsignal()* or *setlocale()*.

47091 The contents of the message strings returned by *strsignal()* should be determined by the setting
 47092 of the *LC_MESSAGES* category in the current locale.

47093 The implementation shall behave as if no function defined in this standard calls *strsignal()*.

47094 Since no return value is reserved to indicate an error, an application wishing to check for error
 47095 situations should set *errno* to 0, then call *strsignal()*, then check *errno*.

47096 The *strsignal()* function need not be reentrant. A function that is not required to be reentrant is
 47097 not required to be thread-safe.

47098 **RETURN VALUE**

47099 Upon successful completion, *strsignal()* shall return a pointer to a string. Otherwise, if *signum* is
 47100 not a valid signal number, the return value is unspecified.

47101 **ERRORS**

47102 No errors are defined.

47103 **EXAMPLES**

47104 None.

47105 **APPLICATION USAGE**

47106 None.

47107 **RATIONALE**

47108 If *signum* is not a valid signal number, some implementations return NULL, while for others the
 47109 *strsignal()* function returns a pointer to a string containing an unspecified message denoting an
 47110 unknown signal. This standard leaves this return value unspecified.

47111 **FUTURE DIRECTIONS**

47112 None.

47113 **SEE ALSO**

47114 *psiginfo()*, *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<string.h>`

47115 **CHANGE HISTORY**

47116 First released in Issue 7.

47117 **NAME**

47118 strspn — get length of a substring

47119 **SYNOPSIS**

47120 #include <string.h>

47121 size_t strspn(const char *s1, const char *s2);

47122 **DESCRIPTION**

47123 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 47124 conflict between the requirements described here and the ISO C standard is unintentional. This
 47125 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47126 The *strspn()* function shall compute the length (in bytes) of the maximum initial segment of the
 47127 string pointed to by *s1* which consists entirely of bytes from the string pointed to by *s2*.

47128 **RETURN VALUE**

47129 The *strspn()* function shall return the length of *s1*; no return value is reserved to indicate an
 47130 error.

47131 **ERRORS**

47132 No errors are defined.

47133 **EXAMPLES**

47134 None.

47135 **APPLICATION USAGE**

47136 None.

47137 **RATIONALE**

47138 None.

47139 **FUTURE DIRECTIONS**

47140 None.

47141 **SEE ALSO**47142 *strcspn()*, the Base Definitions volume of IEEE Std 1003.1-200x, <string.h>47143 **CHANGE HISTORY**

47144 First released in Issue 1. Derived from Issue 1 of the SVID.

47145 **Issue 5**

47146 The RETURN VALUE section is updated to indicate that *strspn()* returns the length of *s*, and not
 47147 *s* itself as was previously stated.

47148 **NAME**
 47149 *strstr* — find a substring

47150 **SYNOPSIS**
 47151 #include <string.h>
 47152 char *strstr(const char *s1, const char *s2);

47153 **DESCRIPTION**
 47154 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 47155 conflict between the requirements described here and the ISO C standard is unintentional. This
 47156 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47157 The *strstr()* function shall locate the first occurrence in the string pointed to by *s1* of the
 47158 sequence of bytes (excluding the terminating NUL character) in the string pointed to by *s2*.

47159 **RETURN VALUE**
 47160 Upon successful completion, *strstr()* shall return a pointer to the located string or a null pointer
 47161 if the string is not found.

47162 If *s2* points to a string with zero length, the function shall return *s1*.

47163 **ERRORS**
 47164 No errors are defined.

47165 **EXAMPLES**
 47166 None.

47167 **APPLICATION USAGE**
 47168 None.

47169 **RATIONALE**
 47170 None.

47171 **FUTURE DIRECTIONS**
 47172 None.

47173 **SEE ALSO**
 47174 *strchr()*, the Base Definitions volume of IEEE Std 1003.1-200x, <string.h>

47175 **CHANGE HISTORY**
 47176 First released in Issue 3. Included for alignment with the ANSI C standard.

47177 **NAME**

47178 strtod, strtodf, strtold — convert a string to a double-precision number

47179 **SYNOPSIS**

47180 #include <stdlib.h>

47181 double strtod(const char *restrict *nptr*, char **restrict *endptr*);47182 float strtodf(const char *restrict *nptr*, char **restrict *endptr*);47183 long double strtold(const char *restrict *nptr*, char **restrict *endptr*);47184 **DESCRIPTION**47185 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47186 conflict between the requirements described here and the ISO C standard is unintentional. This
47187 volume of IEEE Std 1003.1-200x defers to the ISO C standard.47188 These functions shall convert the initial portion of the string pointed to by *nptr* to **double**, **float**,
47189 and **long double** representation, respectively. First, they decompose the input string into three
47190 parts:

- 47191 1. An initial, possibly empty, sequence of white-space characters (as specified by
- isspace()*
-)
-
- 47192 2. A subject sequence interpreted as a floating-point constant or representing infinity or
-
- 47193 NaN
-
- 47194 3. A final string of one or more unrecognized characters, including the terminating NUL
-
- 47195 character of the input string

47196 Then they shall attempt to convert the subject sequence to a floating-point number, and return
47197 the result.47198 The expected form of the subject sequence is an optional plus or minus sign, then one of the
47199 following:

- 47200 • A non-empty sequence of decimal digits optionally containing a radix character; then an
-
- 47201 optional exponent part consisting of the character 'e' or the character 'E', optionally
-
- 47202 followed by a '+' or '-' character, and then followed by one or more decimal digits
-
- 47203 • A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix
-
- 47204 character; then an optional binary exponent part consisting of the character 'p' or the
-
- 47205 character 'P', optionally followed by a '+' or '-' character, and then followed by one or
-
- 47206 more decimal digits
-
- 47207 • One of INF or INFINITY, ignoring case
-
- 47208 • One of NAN or NAN(
- n-char-sequence_{opt}*
-), ignoring case in the NAN part, where:

47209 n-char-sequence:
47210 digit
47211 nondigit
47212 n-char-sequence digit
47213 n-char-sequence nondigit47214 The subject sequence is defined as the longest initial subsequence of the input string, starting
47215 with the first non-white-space character, that is of the expected form. The subject sequence
47216 contains no characters if the input string is not of the expected form.47217 If the subject sequence has the expected form for a floating-point number, the sequence of
47218 characters starting with the first digit or the decimal-point character (whichever occurs first)
47219 shall be interpreted as a floating constant of the C language, except that the radix character shall
47220 be used in place of a period, and that if neither an exponent part nor a radix character appears in

47221 a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal
 47222 floating-point number, an exponent part of the appropriate type with value zero is assumed to
 47223 follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence
 47224 shall be interpreted as negated. A character sequence INF or INFINITY shall be interpreted as an
 47225 infinity, if representable in the return type, else as if it were a floating constant that is too large
 47226 for the range of the return type. A character sequence NAN or NAN(*n-char-sequence_{opt}*) shall be
 47227 interpreted as a quiet NaN, if supported in the return type, else as if it were a subject sequence
 47228 part that does not have the expected form; the meaning of the *n-char* sequences is
 47229 implementation-defined. A pointer to the final string is stored in the object pointed to by *endptr*,
 47230 provided that *endptr* is not a null pointer.

47231 If the subject sequence has the hexadecimal form and FLT_RADIX is a power of 2, the value
 47232 resulting from the conversion is correctly rounded.

47233 CX The radix character is defined in the locale of the process (category *LC_NUMERIC*). In the
 47234 POSIX locale, or in a locale where the radix character is not defined, the radix character shall
 47235 default to a period (`'.'`).

47236 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
 47237 accepted.

47238 If the subject sequence is empty or does not have the expected form, no conversion shall be
 47239 performed; the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not
 47240 a null pointer.

47241 CX The *strtod()* function shall not change the setting of *errno* if successful.

47242 Since 0 is returned on error and is also a valid return on success, an application wishing to check
 47243 for error situations should set *errno* to 0, then call *strtod()*, *strtof()*, or *strtold()*, then check *errno*.

47244 RETURN VALUE

47245 Upon successful completion, these functions shall return the converted value. If no conversion
 47246 could be performed, 0 shall be returned, and *errno* may be set to [EINVAL].

47247 If the correct value is outside the range of representable values, \pm HUGE_VAL, \pm HUGE_VALF, or
 47248 \pm HUGE_VALL shall be returned (according to the sign of the value), and *errno* shall be set to
 47249 [ERANGE].

47250 If the correct value would cause an underflow, a value whose magnitude is no greater than the
 47251 smallest normalized positive number in the return type shall be returned and *errno* set to
 47252 [ERANGE].

47253 ERRORS

47254 These functions shall fail if:

47255 CX [ERANGE] The value to be returned would cause overflow or underflow.

47256 These functions may fail if:

47257 CX [EINVAL] No conversion could be performed.

47258 EXAMPLES

47259 None.

47260 APPLICATION USAGE

47261 If the subject sequence has the hexadecimal form and FLT_RADIX is not a power of 2, and the
 47262 result is not exactly representable, the result should be one of the two numbers in the
 47263 appropriate internal format that are adjacent to the hexadecimal floating source value, with the
 47264 extra stipulation that the error should have a correct sign for the current rounding direction.

47265 If the subject sequence has the decimal form and at most DECIMAL_DIG (defined in `<float.h>`)
 47266 significant digits, the result should be correctly rounded. If the subject sequence *D* has the

47267 decimal form and more than DECIMAL_DIG significant digits, consider the two bounding,
 47268 adjacent decimal strings *L* and *U*, both having DECIMAL_DIG significant digits, such that the
 47269 values of *L*, *D*, and *U* satisfy $L \leq D \leq U$. The result should be one of the (equal or adjacent)
 47270 values that would be obtained by correctly rounding *L* and *U* according to the current rounding
 47271 direction, with the extra stipulation that the error with respect to *D* should have a correct sign
 47272 for the current rounding direction.

47273 The changes to *strtod()* introduced by the ISO/IEC 9899:1999 standard can alter the behavior of
 47274 well-formed applications complying with the ISO/IEC 9899:1990 standard and thus earlier
 47275 versions of the base documents. One such example would be:

```

47276 int
47277 what_kind_of_number (char *s)
47278 {
47279     char *endp;
47280     double d;
47281     long l;
47282
47283     d = strtod(s, &endp);
47284     if (s != endp && *endp == '\0')
47285         printf("It's a float with value %g\n", d);
47286     else
47287     {
47288         l = strtol(s, &endp, 0);
47289         if (s != endp && *endp == '\0')
47290             printf("It's an integer with value %ld\n", l);
47291         else
47292             return 1;
47293     }
47294     return 0;
  
```

47295 If the function is called with:

```
47296 what_kind_of_number ("0x10")
```

47297 an ISO/IEC 9899:1990 standard-compliant library will result in the function printing:

```
47298 It's an integer with value 16
```

47299 With the ISO/IEC 9899:1999 standard, the result is:

```
47300 It's a float with value 16
```

47301 The change in behavior is due to the inclusion of floating-point numbers in hexadecimal
 47302 notation without requiring that either a decimal point or the binary exponent be present.

47303 RATIONALE

47304 None.

47305 FUTURE DIRECTIONS

47306 None.

47307 SEE ALSO

47308 [isspace\(\)](#), [localeconv\(\)](#), [scanf\(\)](#), [setlocale\(\)](#), [strtol\(\)](#), the Base Definitions volume of
 47309 IEEE Std 1003.1-200x, Chapter 7, Locale, [<float.h>](#), [<stdlib.h>](#)

47310
47311
47312
47313
47314
47315
47316
47317
47318
47319
47320
47321
47322
47323
47324
47325
47326
47327
47328

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The *strtod()* function is updated.
- The *strtof()* and *strtold()* functions are added.
- The DESCRIPTION is extensively revised.

ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/61 is applied, correcting the second paragraph in the RETURN VALUE section. This change clarifies the sign of the return value.

Issue 7

Austin Group Interpretation 1003.1-2001 #015 is applied.

47329 **NAME**
 47330 `strtoimax`, `strtoumax` — convert string to integer type

47331 **SYNOPSIS**
 47332 `#include <inttypes.h>`
 47333 `intmax_t strtoimax(const char *restrict nptr, char **restrict endptr,`
 47334 `int base);`
 47335 `uintmax_t strtoumax(const char *restrict nptr, char **restrict endptr,`
 47336 `int base);`

47337 **DESCRIPTION**
 47338 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 47339 conflict between the requirements described here and the ISO C standard is unintentional. This
 47340 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47341 These functions shall be equivalent to the `strtol()`, `strtoll()`, `strtoul()`, and `strtoull()` functions,
 47342 except that the initial portion of the string shall be converted to `intmax_t` and `uintmax_t`
 47343 representation, respectively.

47344 **RETURN VALUE**
 47345 These functions shall return the converted value, if any.
 47346 If no conversion could be performed, zero shall be returned.
 47347 If the correct value is outside the range of representable values, `{INTMAX_MAX}`,
 47348 `{INTMAX_MIN}`, or `{UINTMAX_MAX}` shall be returned (according to the return type and sign
 47349 of the value, if any), and `errno` shall be set to `[ERANGE]`.

47350 **ERRORS**
 47351 These functions shall fail if:
 47352 `[ERANGE]` The value to be returned is not representable.
 47353 These functions may fail if:
 47354 `[EINVAL]` The value of `base` is not supported.

47355 **EXAMPLES**
 47356 None.

47357 **APPLICATION USAGE**
 47358 None.

47359 **RATIONALE**
 47360 None.

47361 **FUTURE DIRECTIONS**
 47362 None.

47363 **SEE ALSO**
 47364 `strtol()`, `strtoul()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<inttypes.h>`

47365 **CHANGE HISTORY**
 47366 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

47367 **NAME**
 47368 `strtok, strtok_r` — split string into tokens

47369 **SYNOPSIS**

```
47370 #include <string.h>
47371 char *strtok(char *restrict s1, const char *restrict s2);
47372 CX char *strtok_r(char *restrict s, const char *restrict sep,
47373 char **restrict lasts);
```

47374 **DESCRIPTION**

47375 CX For `strtok()`: The functionality described on this reference page is aligned with the ISO C
 47376 standard. Any conflict between the requirements described here and the ISO C standard is
 47377 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47378 A sequence of calls to `strtok()` breaks the string pointed to by `s1` into a sequence of tokens, each
 47379 of which is delimited by a byte from the string pointed to by `s2`. The first call in the sequence
 47380 has `s1` as its first argument, and is followed by calls with a null pointer as their first argument.
 47381 The separator string pointed to by `s2` may be different from call to call.

47382 The first call in the sequence searches the string pointed to by `s1` for the first byte that is *not*
 47383 contained in the current separator string pointed to by `s2`. If no such byte is found, then there
 47384 are no tokens in the string pointed to by `s1` and `strtok()` shall return a null pointer. If such a byte
 47385 is found, it is the start of the first token.

47386 The `strtok()` function then searches from there for a byte that *is* contained in the current separator
 47387 string. If no such byte is found, the current token extends to the end of the string pointed to by
 47388 `s1`, and subsequent searches for a token shall return a null pointer. If such a byte is found, it is
 47389 overwritten by a NUL character, which terminates the current token. The `strtok()` function saves
 47390 a pointer to the following byte, from which the next search for a token shall start.

47391 Each subsequent call, with a null pointer as the value of the first argument, starts searching from
 47392 the saved pointer and behaves as described above.

47393 The implementation shall behave as if no function defined in this volume of
 47394 IEEE Std 1003.1-200x calls `strtok()`.

47395 CX The `strtok()` function need not be thread-safe. A function that is not required to be thread-safe is
 47396 not required to be reentrant.

47397 The `strtok_r()` function considers the null-terminated string `s` as a sequence of zero or more text
 47398 tokens separated by spans of one or more characters from the separator string `sep`. The
 47399 argument `lasts` points to a user-provided pointer which points to stored information necessary
 47400 for `strtok_r()` to continue scanning the same string.

47401 In the first call to `strtok_r()`, `s` points to a null-terminated string, `sep` to a null-terminated string of
 47402 separator characters, and the value pointed to by `lasts` is ignored. The `strtok_r()` function shall
 47403 return a pointer to the first character of the first token, write a null character into `s` immediately
 47404 following the returned token, and update the pointer to which `lasts` points.

47405 In subsequent calls, `s` is a null pointer and `lasts` shall be unchanged from the previous call so that
 47406 subsequent calls shall move through the string `s`, returning successive tokens until no tokens
 47407 remain. The separator string `sep` may be different from call to call. When no token remains in `s`, a
 47408 null pointer shall be returned.

47409 **RETURN VALUE**

47410 Upon successful completion, *strtok()* shall return a pointer to the first byte of a token. Otherwise,
 47411 if there is no token, *strtok()* shall return a null pointer.

47412 CX The *strtok_r()* function shall return a pointer to the token found, or a null pointer when no token
 47413 is found.

47414 **ERRORS**

47415 No errors are defined.

47416 **EXAMPLES**47417 **Searching for Word Separators**

47418 The following example searches for tokens separated by <space>s.

```
47419 #include <string.h>
47420 ...
47421 char *token;
47422 char *line = "LINE TO BE SEPARATED";
47423 char *search = " ";

47424 /* Token will point to "LINE". */
47425 token = strtok(line, search);

47426 /* Token will point to "TO". */
47427 token = strtok(NULL, search);
```

47428 **Breaking a Line**

47429 The following example uses *strtok()* to break a line into two character strings separated by any
 47430 combination of <space>s, <tab>s, or <newline>s.

```
47431 #include <string.h>
47432 ...
47433 struct element {
47434     char *key;
47435     char *data;
47436 };
47437 ...
47438 char line[LINE_MAX];
47439 char *key, *data;
47440 ...
47441 key = strtok(line, " \n");
47442 data = strtok(NULL, " \n");
47443 ...
```

47444 **APPLICATION USAGE**

47445 The *strtok_r()* function is thread-safe and stores its state in a user-supplied buffer instead of
 47446 possibly using a static data area that may be overwritten by an unrelated call from another
 47447 thread.

47448 **RATIONALE**

47449 The *strtok()* function searches for a separator string within a larger string. It returns a pointer to
 47450 the last substring between separator strings. This function uses static storage to keep track of
 47451 the current string position between calls. The new function, *strtok_r()*, takes an additional
 47452 argument, *lasts*, to keep track of the current position in the string.

47453

FUTURE DIRECTIONS

47454

None.

47455

SEE ALSO

47456

The Base Definitions volume of IEEE Std 1003.1-200x, `<string.h>`

47457

CHANGE HISTORY

47458

First released in Issue 1. Derived from Issue 1 of the SVID.

47459

Issue 5

47460

The `strtok_r()` function is included for alignment with the POSIX Threads Extension.

47461

A note indicating that the `strtok()` function need not be reentrant is added to the DESCRIPTION.

47462

Issue 6

47463

Extensions beyond the ISO C standard are marked.

47464

The `strtok_r()` function is marked as part of the Thread-Safe Functions option.

47465

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

47466

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

47467

47468

The `restrict` keyword is added to the `strtok()` and `strtok_r()` prototypes for alignment with the ISO/IEC 9899:1999 standard.

47469

47470

Issue 7

47471

The `strtok_r()` function is moved from the Thread-Safe Functions option to the Base.

DRAFT

47472 NAME

47473 strtol, strtoll — convert a string to a long integer

47474 SYNOPSIS

47475 #include <stdlib.h>

47476 long strtol(const char *restrict *str*, char **restrict *endptr*, int *base*);47477 long long strtoll(const char *restrict *str*, char **restrict *endptr*,
47478 int *base*)

47479 DESCRIPTION

47480 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47481 conflict between the requirements described here and the ISO C standard is unintentional. This
47482 volume of IEEE Std 1003.1-200x defers to the ISO C standard.47483 These functions shall convert the initial portion of the string pointed to by *str* to a type **long** and
47484 **long long** representation, respectively. First, they decompose the input string into three parts:

- 47485 1. An initial, possibly empty, sequence of white-space characters (as specified by
- isspace()*
-)
-
- 47486 2. A subject sequence interpreted as an integer represented in some radix determined by the
-
- 47487 value of
- base*
-
- 47488 3. A final string of one or more unrecognized characters, including the terminating NUL
-
- 47489 character of the input string.

47490 Then they shall attempt to convert the subject sequence to an integer, and return the result.

47491 If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant,
47492 octal constant, or hexadecimal constant, any of which may be preceded by a '+' or '-' sign. A
47493 decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An
47494 octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to
47495 '7' only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the
47496 decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.47497 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
47498 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
47499 by a '+' or '-' sign. The letters from 'a' (or 'A') to 'z' (or 'Z') inclusive are ascribed the
47500 values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the
47501 value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and
47502 digits, following the sign if present.47503 The subject sequence is defined as the longest initial subsequence of the input string, starting
47504 with the first non-white-space character that is of the expected form. The subject sequence shall
47505 contain no characters if the input string is empty or consists entirely of white-space characters,
47506 or if the first non-white-space character is other than a sign or a permissible letter or digit.47507 If the subject sequence has the expected form and the value of *base* is 0, the sequence of
47508 characters starting with the first digit shall be interpreted as an integer constant. If the subject
47509 sequence has the expected form and the value of *base* is between 2 and 36, it shall be used as the
47510 base for conversion, ascribing to each letter its value as given above. If the subject sequence
47511 begins with a minus sign, the value resulting from the conversion shall be negated. A pointer to
47512 the final string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null
47513 pointer.47514 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
47515 accepted.

47516 If the subject sequence is empty or does not have the expected form, no conversion is performed;

47517 the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null
 47518 pointer.

47519 CX The *strtol()* function shall not change the setting of *errno* if successful.

47520 Since 0, {LONG_MIN} or {LLONG_MIN}, and {LONG_MAX} or {LLONG_MAX} are returned
 47521 on error and are also valid returns on success, an application wishing to check for error
 47522 situations should set *errno* to 0, then call *strtol()* or *strtoll()*, then check *errno*.

47523 RETURN VALUE

47524 Upon successful completion, these functions shall return the converted value, if any. If no
 47525 conversion could be performed, 0 shall be returned and *errno* may be set to [EINVAL].

47526 If the correct value is outside the range of representable values, {LONG_MIN}, {LONG_MAX},
 47527 {LLONG_MIN}, or {LLONG_MAX} shall be returned (according to the sign of the value), and
 47528 *errno* set to [ERANGE].

47529 ERRORS

47530 These functions shall fail if:

47531 [ERANGE] The value to be returned is not representable.

47532 These functions may fail if:

47533 CX [EINVAL] The value of *base* is not supported.

47534 EXAMPLES

47535 None.

47536 APPLICATION USAGE

47537 None.

47538 RATIONALE

47539 None.

47540 FUTURE DIRECTIONS

47541 None.

47542 SEE ALSO

47543 *isalpha()*, *scanf()*, *strtod()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdlib.h>

47544 CHANGE HISTORY

47545 First released in Issue 1. Derived from Issue 1 of the SVID.

47546 Issue 5

47547 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

47548 Issue 6

47549 Extensions beyond the ISO C standard are marked.

47550 The following new requirements on POSIX implementations derive from alignment with the
 47551 Single UNIX Specification:

- 47552 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
- 47553 added if no conversion could be performed.

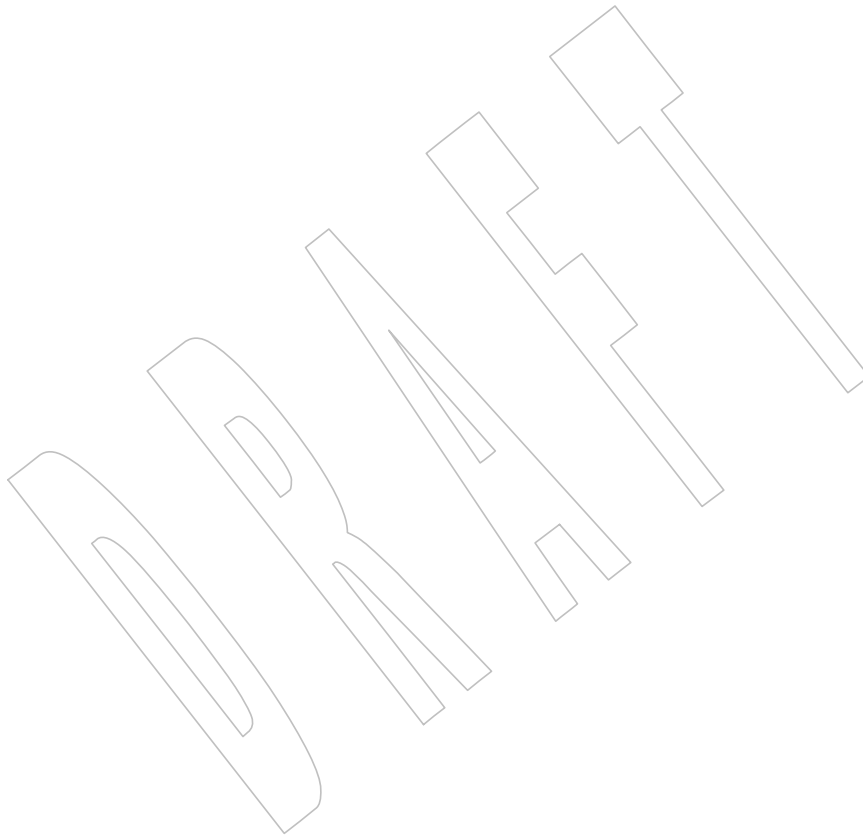
47554 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 47555 • The *strtol()* prototype is updated.
- 47556 • The *strtoll()* function is added.

47557 **NAME**
47558 **strtold** — convert a string to a double-precision number

47559 **SYNOPSIS**
47560 #include <stdlib.h>
47561 long double strtold(const char *restrict *nptr*, char **restrict *endptr*);

47562 **DESCRIPTION**
47563 Refer to *strtod()*.



47564 **NAME**
47565 strtoll — convert a string to a long integer

47566 **SYNOPSIS**
47567 #include <stdlib.h>
47568 long long strtoll(const char *restrict str, char **restrict endptr,
47569 int base);

47570 **DESCRIPTION**
47571 Refer to *strtol()*.



47572 **NAME**
 47573 `strtol, strtoull` — convert a string to an unsigned long

47574 **SYNOPSIS**
 47575 `#include <stdlib.h>`
 47576 `unsigned long strtol(const char *restrict str,`
 47577 `char **restrict endptr, int base);`
 47578 `unsigned long long strtoull(const char *restrict str,`
 47579 `char **restrict endptr, int base);`

47580 **DESCRIPTION**

47581 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 47582 conflict between the requirements described here and the ISO C standard is unintentional. This
 47583 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47584 These functions shall convert the initial portion of the string pointed to by *str* to a type **unsigned**
 47585 **long** and **unsigned long long** representation, respectively. First, they decompose the input string
 47586 into three parts:

- 47587 1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace()*)
- 47588 2. A subject sequence interpreted as an integer represented in some radix determined by the
 47589 value of *base*
- 47590 3. A final string of one or more unrecognized characters, including the terminating NUL
 47591 character of the input string

47592 Then they shall attempt to convert the subject sequence to an unsigned integer, and return the
 47593 result.

47594 If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant,
 47595 octal constant, or hexadecimal constant, any of which may be preceded by a '+' or '-' sign. A
 47596 decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An
 47597 octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to
 47598 '7' only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the
 47599 decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

47600 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
 47601 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
 47602 by a '+' or '-' sign. The letters from 'a' (or 'A') to 'z' (or 'Z') inclusive are ascribed the
 47603 values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the
 47604 value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and
 47605 digits, following the sign if present.

47606 The subject sequence is defined as the longest initial subsequence of the input string, starting
 47607 with the first non-white-space character that is of the expected form. The subject sequence shall
 47608 contain no characters if the input string is empty or consists entirely of white-space characters,
 47609 or if the first non-white-space character is other than a sign or a permissible letter or digit.

47610 If the subject sequence has the expected form and the value of *base* is 0, the sequence of
 47611 characters starting with the first digit shall be interpreted as an integer constant. If the subject
 47612 sequence has the expected form and the value of *base* is between 2 and 36, it shall be used as the
 47613 base for conversion, ascribing to each letter its value as given above. If the subject sequence
 47614 begins with a minus sign, the value resulting from the conversion shall be negated. A pointer to
 47615 the final string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null
 47616 pointer.

47617 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
47618 accepted.

47619 If the subject sequence is empty or does not have the expected form, no conversion shall be
47620 performed; the value of *str* shall be stored in the object pointed to by *endptr*, provided that *endptr*
47621 is not a null pointer.

47622 CX The *strtol()* function shall not change the setting of *errno* if successful.

47623 Since 0, {ULONG_MAX}, and {ULLONG_MAX} are returned on error and are also valid returns
47624 on success, an application wishing to check for error situations should set *errno* to 0, then call
47625 *strtol()* or *strtoll()*, then check *errno*.

47626 RETURN VALUE

47627 Upon successful completion, these functions shall return the converted value, if any. If no
47628 CX conversion could be performed, 0 shall be returned and *errno* may be set to [EINVAL]. If the
47629 correct value is outside the range of representable values, {ULONG_MAX} or {ULLONG_MAX}
47630 shall be returned and *errno* set to [ERANGE].

47631 ERRORS

47632 These functions shall fail if:

47633 CX [EINVAL] The value of *base* is not supported.

47634 [ERANGE] The value to be returned is not representable.

47635 These functions may fail if:

47636 CX [EINVAL] No conversion could be performed.

47637 EXAMPLES

47638 None.

47639 APPLICATION USAGE

47640 None.

47641 RATIONALE

47642 None.

47643 FUTURE DIRECTIONS

47644 None.

47645 SEE ALSO

47646 *isalpha()*, *scanf()*, *strtod()*, *strtol()*, the Base Definitions volume of IEEE Std 1003.1-200x,
47647 <stdlib.h>

47648 CHANGE HISTORY

47649 First released in Issue 4. Derived from the ANSI C standard.

47650 Issue 5

47651 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

47652 Issue 6

47653 Extensions beyond the ISO C standard are marked.

47654 The following new requirements on POSIX implementations derive from alignment with the
47655 Single UNIX Specification:

- 47656 • The [EINVAL] error condition is added for when the value of *base* is not supported.

47657 In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
47658 added if no conversion could be performed.

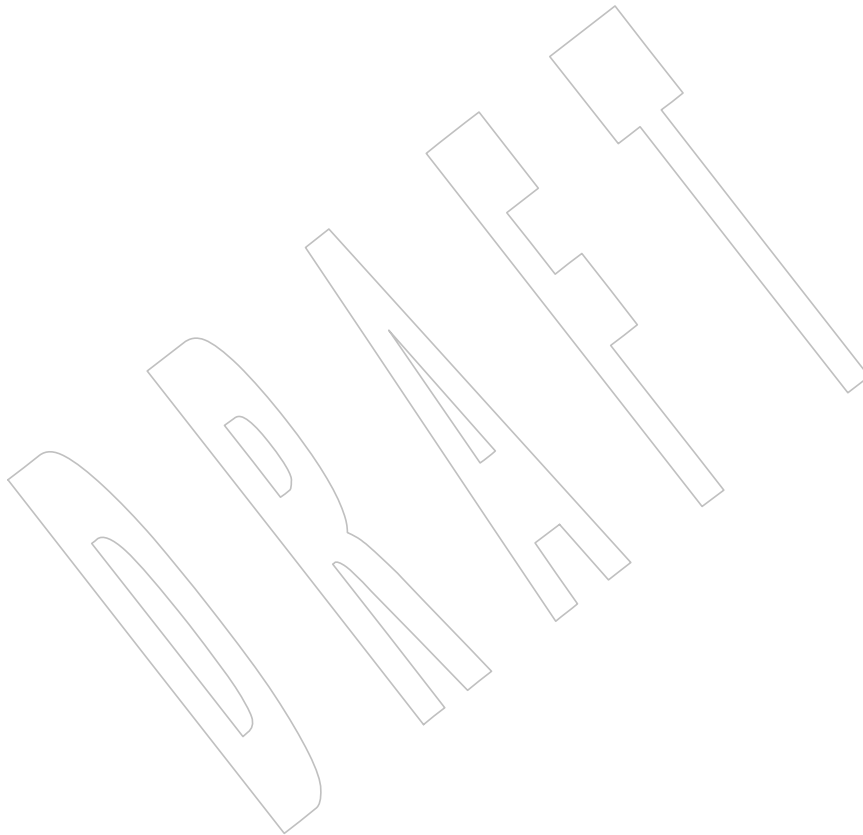
47659 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

strtoul()

47660

47661

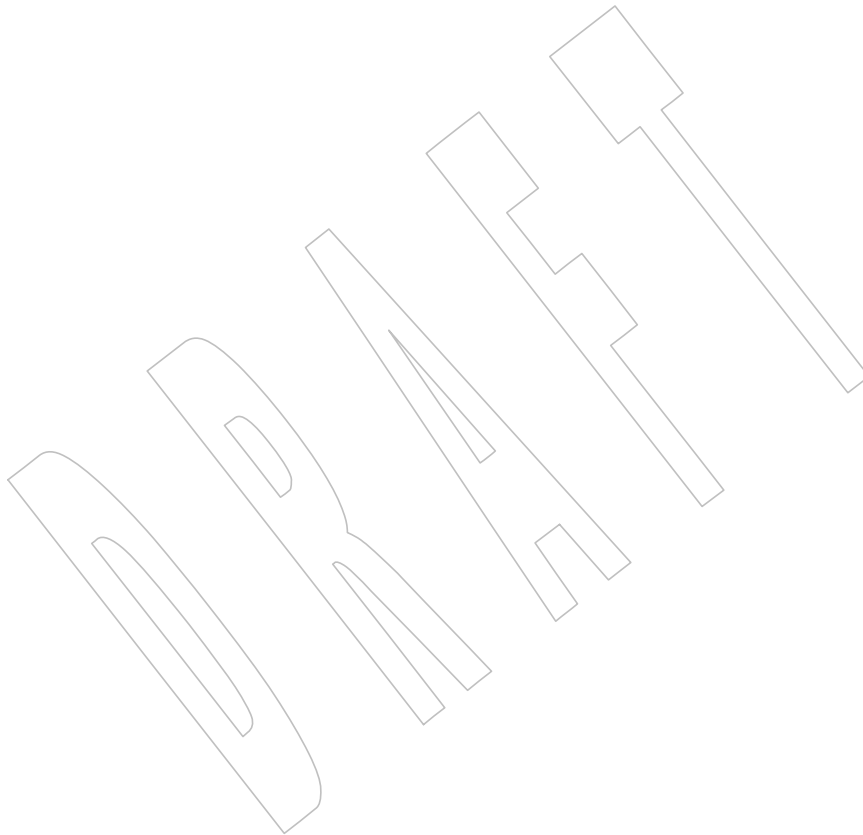
- The *strtoul()* prototype is updated.
- The *strtoull()* function is added.



47662 **NAME**
47663 **strtoumax** — convert a string to an integer type

47664 **SYNOPSIS**
47665 #include <inttypes.h>
47666 uintmax_t strtoumax(const char *restrict *nptr*, char **restrict *endptr*,
47667 int *base*);

47668 **DESCRIPTION**
47669 Refer to *strtoimax()*.



47670 **NAME**47671 `strxfrm, strxfrm_l` — string transformation47672 **SYNOPSIS**47673 `#include <string.h>`47674 `size_t strxfrm(char *restrict s1, const char *restrict s2, size_t n);`47675 CX `size_t strxfrm_l(char *restrict s1, const char *restrict s2,`
47676 `size_t n, locale_t locale);`47677 **DESCRIPTION**47678 CX For `strxfrm()`: The functionality described on this reference page is aligned with the ISO C
47679 standard. Any conflict between the requirements described here and the ISO C standard is
47680 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.47681 CX The `strxfrm()` and `strxfrm_l()` functions shall transform the string pointed to by `s2` and place the
47682 resulting string into the array pointed to by `s1`. The transformation is such that if `strcmp()` is
47683 applied to two transformed strings, it shall return a value greater than, equal to, or less than 0,
47684 corresponding to the result of `strcoll()` or `strcoll_l()`, respectively, applied to the same two
47685 original strings with the same locale. No more than `n` bytes are placed into the resulting array
47686 pointed to by `s1`, including the terminating NUL character. If `n` is 0, `s1` is permitted to be a null
47687 pointer. If copying takes place between objects that overlap, the behavior is undefined.47688 CX The `strxfrm()` and `strxfrm_l()` functions shall not change the setting of `errno` if successful.47689 Since no return value is reserved to indicate an error, an application wishing to check for error
47690 situations should set `errno` to 0, then call `strxfrm()` or `strxfrm_l()`, then check `errno`.47691 **RETURN VALUE**47692 CX Upon successful completion, `strxfrm()` and `strxfrm_l()` shall return the length of the
47693 transformed string (not including the terminating NUL character). If the value returned is `n` or
47694 more, the contents of the array pointed to by `s1` are unspecified.47695 CX On error, `strxfrm()` and `strxfrm_l()` may set `errno` but no return value is reserved to indicate an
47696 error.47697 **ERRORS**

47698 These functions may fail if:

47699 CX [EINVAL] The string pointed to by the `s2` argument contains characters outside the
47700 domain of the collating sequence.47701 The `strxfrm_l()` function may fail if:47702 CX [EINVAL] `locale` is not a valid locale object.47703 **EXAMPLES**

47704 None.

47705 **APPLICATION USAGE**47706 The transformation function is such that two transformed strings can be ordered by `strcmp()` as
47707 appropriate to collating sequence information in the locale of the process (category
47708 `LC_COLLATE`).47709 The fact that when `n` is 0 `s1` is permitted to be a null pointer is useful to determine the size of the
47710 `s1` array prior to making the transformation.

47711
47712
47713
47714
47715
47716
47717
47718
47719
47720
47721
47722
47723
47724
47725
47726
47727
47728
47729
47730

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strcmp(), *strcoll()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<string.h>`

CHANGE HISTORY

First released in Issue 3. Included for alignment with the ISO C standard.

Issue 5

The DESCRIPTION is updated to indicate that *errno* does not change if the function is successful.

Issue 6

Extensions beyond the ISO C standard are marked.

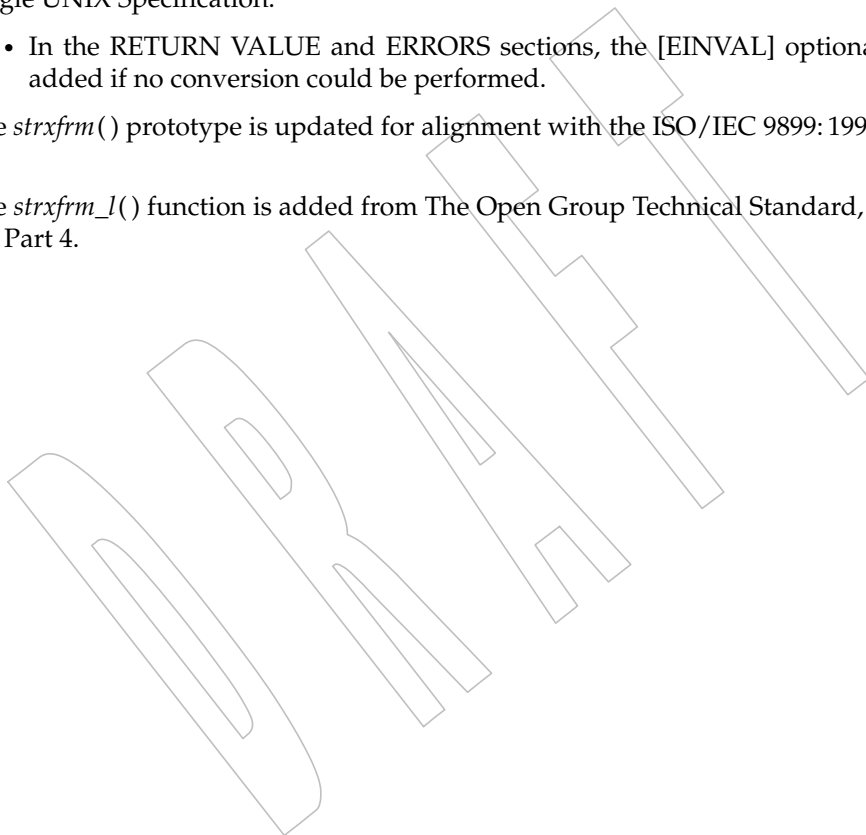
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The *strxfrm()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

Issue 7

The *strxfrm_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



47731 **NAME**
 47732 `swab` — swap bytes

47733 **SYNOPSIS**

```
47734 XSI #include <unistd.h>
47735 void swab(const void *restrict src, void *restrict dest,
47736          ssize_t nbytes);
```

47737 **DESCRIPTION**

47738 The `swab()` function shall copy `nbytes` bytes, which are pointed to by `src`, to the object pointed to
 47739 by `dest`, exchanging adjacent bytes. The `nbytes` argument should be even. If `nbytes` is odd, `swab()`
 47740 copies and exchanges `nbytes-1` bytes and the disposition of the last byte is unspecified. If
 47741 copying takes place between objects that overlap, the behavior is undefined. If `nbytes` is
 47742 negative, `swab()` does nothing.

47743 **RETURN VALUE**

47744 None.

47745 **ERRORS**

47746 No errors are defined.

47747 **EXAMPLES**

47748 None.

47749 **APPLICATION USAGE**

47750 None.

47751 **RATIONALE**

47752 None.

47753 **FUTURE DIRECTIONS**

47754 None.

47755 **SEE ALSO**

47756 The Base Definitions volume of IEEE Std 1003.1-200x, `<unistd.h>`

47757 **CHANGE HISTORY**

47758 First released in Issue 1. Derived from Issue 1 of the SVID.

47759 **Issue 6**

47760 The `restrict` keyword is added to the `swab()` prototype for alignment with the
 47761 ISO/IEC 9899:1999 standard.

47762
47763
47764
47765
47766
47767
47768
47769
47770

NAME

swprintf — print formatted wide-character output

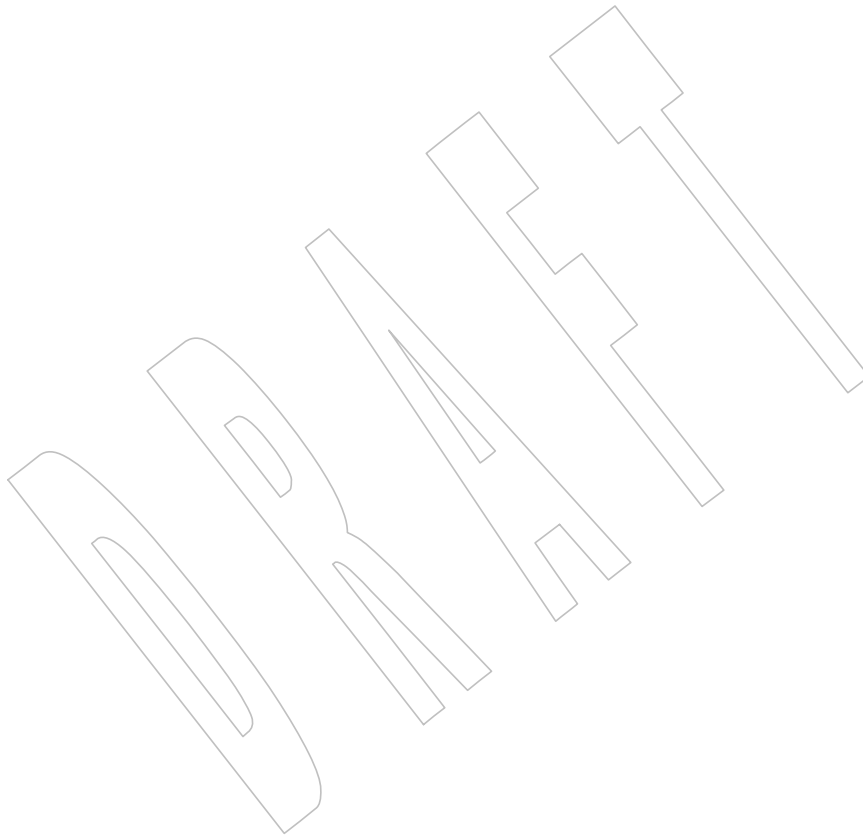
SYNOPSIS

```
#include <stdio.h>  
#include <wchar.h>
```

```
int swprintf(wchar_t *restrict ws, size_t n,  
             const wchar_t *restrict format, ...);
```

DESCRIPTION

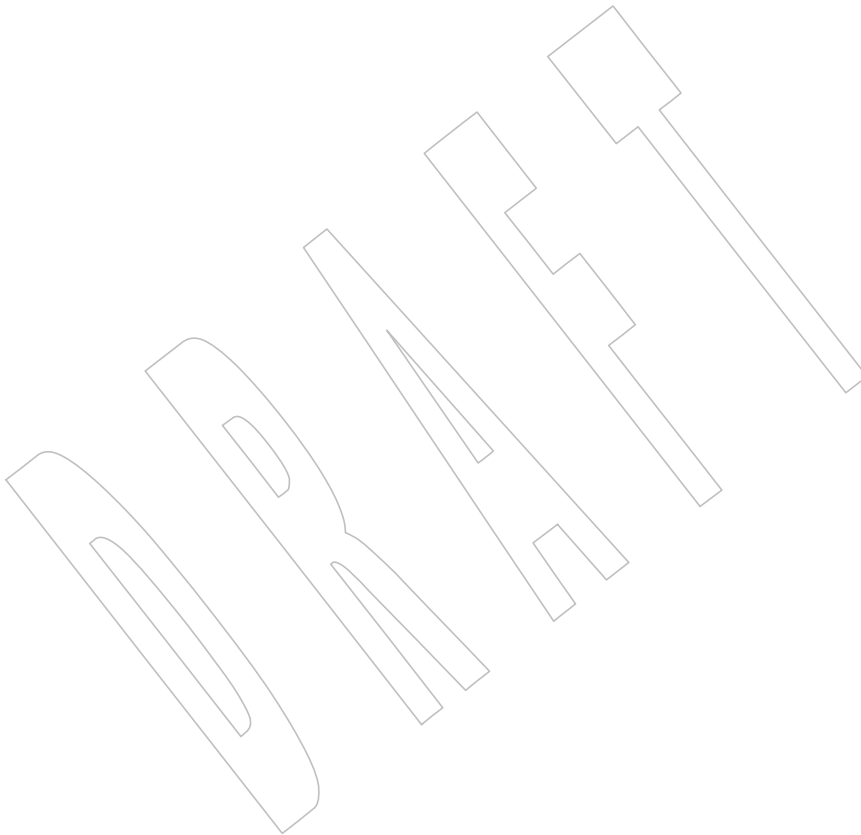
Refer to [fwprintf\(\)](#).



47771 **NAME**
47772 `swscanf` — convert formatted wide-character input

47773 **SYNOPSIS**
47774 `#include <stdio.h>`
47775 `#include <wchar.h>`
47776 `int swscanf(const wchar_t *restrict ws,`
47777 `const wchar_t *restrict format, ...);`

47778 **DESCRIPTION**
47779 Refer to *fwscanf()*.



47780 **NAME**
 47781 `symlink, symlinkat` — make a symbolic link relative to directory file descriptor

47782 **SYNOPSIS**
 47783 `#include <unistd.h>`
 47784 `int symlink(const char *path1, const char *path2);`
 47785 `int symlinkat(const char *path1, int fd, const char *path2);`

47786 **DESCRIPTION**
 47787 The `symlink()` function shall create a symbolic link called `path2` that contains the string pointed to
 47788 by `path1` (`path2` is the name of the symbolic link created, `path1` is the string contained in the
 47789 symbolic link).

47790 The string pointed to by `path1` shall be treated only as a character string and shall not be
 47791 validated as a pathname.

47792 If the `symlink()` function fails for any reason other than [EIO], any file named by `path2` shall be
 47793 unaffected.

47794 The `symlinkat()` function shall be equivalent to the `symlink()` function except in the case where
 47795 `path2` specifies a relative path. In this case the symbolic link is created relative to the directory
 47796 associated with the file descriptor `fd` instead of the current working directory. It is unspecified
 47797 whether directory searches are permitted based on whether the file was opened with search
 47798 permission or on the current permissions of the directory underlying the file descriptor.

47799 If `symlinkat()` is passed the special value `AT_FDCWD` in the `fd` parameter, the current working
 47800 directory is used and the behavior shall be identical to a call to `symlink()`.

47801 **RETURN VALUE**
 47802 Upon successful completion, these functions shall return 0. Otherwise, these functions shall
 47803 return `-1` and set `errno` to indicate the error.

47804 **ERRORS**
 47805 These functions shall fail if:

47806 47807 47808	[EACCES]	Write permission is denied in the directory where the symbolic link is being created, or search permission is denied for a component of the path prefix of <code>path2</code> .
47809	[EEXIST]	The <code>path2</code> argument names an existing file or symbolic link.
47810	[EIO]	An I/O error occurs while reading from or writing to the file system.
47811 47812	[ELOOP]	A loop exists in symbolic links encountered during resolution of the <code>path2</code> argument.
47813 47814 47815 47816	[ENAMETOOLONG]	The length of the <code>path2</code> argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX} or the length of the <code>path1</code> argument is longer than {SYMLINK_MAX}.
47817 47818	[ENOENT]	A component of <code>path2</code> does not name an existing file or <code>path2</code> is an empty string.
47819 47820 47821 47822 47823	[ENOSPC]	The directory in which the entry for the new symbolic link is being placed cannot be extended because no space is left on the file system containing the directory, or the new symbolic link cannot be created because no space is left on the file system which shall contain the link, or the file system is out of file-allocation resources.

- 47824 [ENOTDIR] A component of the path prefix of *path2* is not a directory.
- 47825 [EROFS] The new symbolic link would reside on a read-only file system.
- 47826 The *symlinkat()* function shall fail if:
- 47827 [EBADF] The *path2* argument does not specify an absolute path and the *fd* argument is
47828 neither AT_FDCWD nor a valid file descriptor open for searching.
- 47829 These functions may fail if:
- 47830 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
47831 resolution of the *path2* argument.
- 47832 [ENAMETOOLONG]
- 47833 As a result of encountering a symbolic link in resolution of the *path2*
47834 argument, the length of the substituted pathname string exceeded
47835 {PATH_MAX} bytes (including the terminating null byte), or the length of the
47836 string pointed to by *path1* exceeded {SYMLINK_MAX}.
- 47837 The *symlinkat()* function may fail if:
- 47838 [ENOTDIR] The *path2* argument is not an absolute path and *fd* is neither AT_FDCWD nor a
47839 file descriptor associated with a directory.

EXAMPLES

47840 None.
47841

APPLICATION USAGE

47842 Like a hard link, a symbolic link allows a file to have multiple logical names. The presence of a
47843 hard link guarantees the existence of a file, even after the original name has been removed. A
47844 symbolic link provides no such assurance; in fact, the file named by the *path1* argument need not
47845 exist when the link is created. A symbolic link can cross file system boundaries.
47846

47847 Normal permission checks are made on each component of the symbolic link pathname during
47848 its resolution.

RATIONALE

47849 Since IEEE Std 1003.1-200x does not require any association of file times with symbolic links,
47850 there is no requirement that file times be updated by *symlink()*.
47851

47852 The purpose of the *symlinkat()* function is to create symbolic links in directories other than the
47853 current working directory without exposure to race conditions. Any part of the path of a file
47854 could be changed in parallel to a call to *symlink()*, resulting in unspecified behavior. By opening
47855 a file descriptor for the target directory and using the *symlinkat()* function it can be guaranteed
47856 that the created symbolic link is located relative to the desired directory.

FUTURE DIRECTIONS

47857 None.
47858

SEE ALSO

47859 *fdopendir()*, *fstatat()*, *lchown()*, *link()*, *open()*, *readlink()*, *rename()*, *unlink()*, the Base Definitions
47860 volume of IEEE Std 1003.1-200x, <unistd.h>
47861

CHANGE HISTORY

47862 First released in Issue 4, Version 2.
47863

Issue 5

47864 Moved from X/OPEN UNIX extension to BASE.
47865

47866

Issue 6

47867

The following changes were made to align with the IEEE P1003.1a draft standard:

47868

- The DESCRIPTION text is updated.

47869

- The [ELOOP] optional error condition is added.

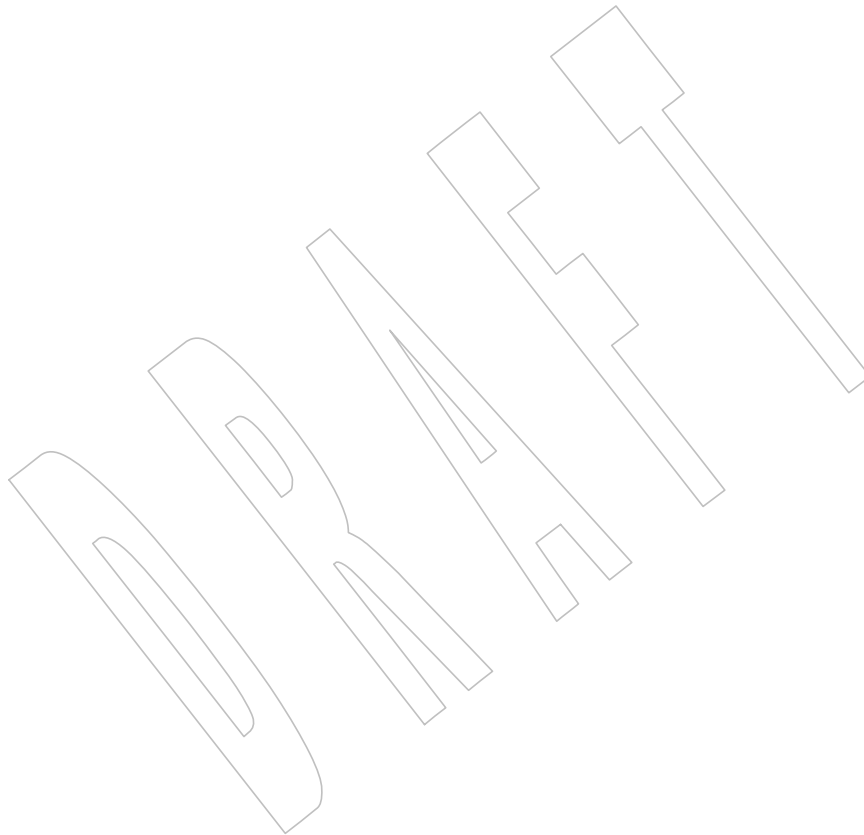
47870

Issue 7

47871

The *symlinkat()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

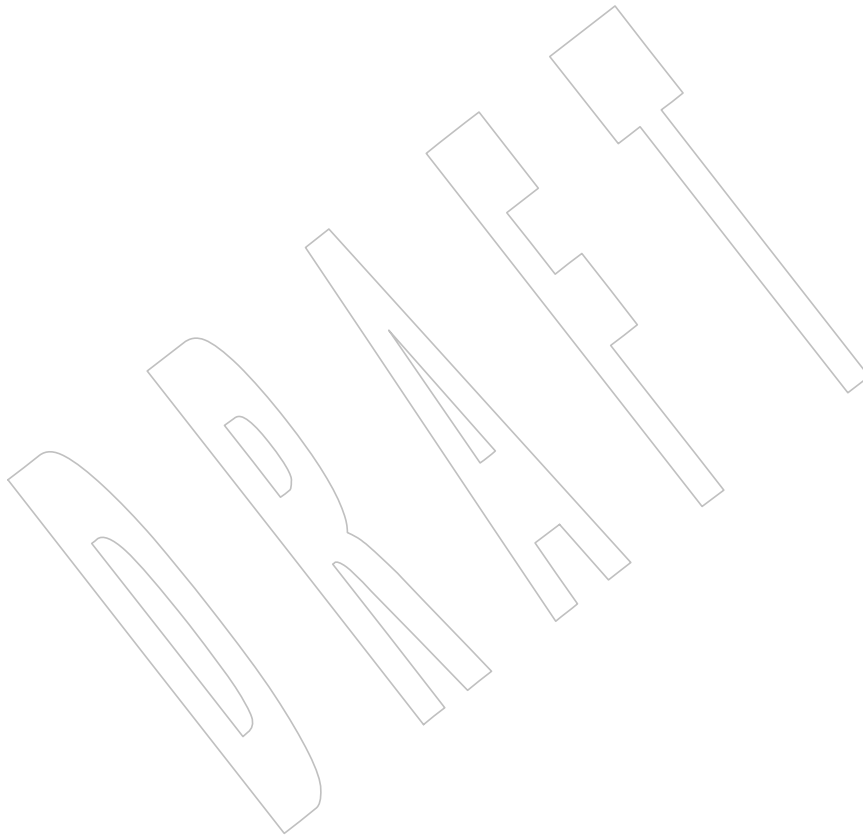
47872



47873 **NAME**
47874 `symlinkat` — make a symbolic link relative to directory file descriptor

47875 **SYNOPSIS**
47876 `#include <unistd.h>`
47877 `int symlinkat(const char *path1, int fd, const char *path2);`

47878 **DESCRIPTION**
47879 Refer to *symlink()*.



47880 **NAME**
 47881 sync — schedule file system updates

47882 **SYNOPSIS**

47883 XSI `#include <unistd.h>`
 47884 `void sync(void);`

47885 **DESCRIPTION**

47886 The *sync()* function shall cause all information in memory that updates file systems to be
 47887 scheduled for writing out to all file systems.

47888 The writing, although scheduled, is not necessarily complete upon return from *sync()*.

47889 **RETURN VALUE**

47890 The *sync()* function shall not return a value.

47891 **ERRORS**

47892 No errors are defined.

47893 **EXAMPLES**

47894 None.

47895 **APPLICATION USAGE**

47896 None.

47897 **RATIONALE**

47898 None.

47899 **FUTURE DIRECTIONS**

47900 None.

47901 **SEE ALSO**

47902 *fsync()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<unistd.h>`

47903 **CHANGE HISTORY**

47904 First released in Issue 4, Version 2.

47905 **Issue 5**

47906 Moved from X/OPEN UNIX extension to BASE.

47907 **NAME**
 47908 sysconf — get configurable system variables

47909 **SYNOPSIS**
 47910 #include <unistd.h>

47911 long sysconf(int name);

47912 **DESCRIPTION**
 47913 The *sysconf()* function provides a method for the application to determine the current value of a
 47914 configurable system limit or option (*variable*). The implementation shall support all of the
 47915 variables listed in the following table and may support others.

47916 The *name* argument represents the system variable to be queried. The following table lists the
 47917 minimal set of system variables from <limits.h> or <unistd.h> that can be returned by *sysconf()*,
 47918 and the symbolic constants defined in <unistd.h> that are the corresponding values used for
 47919 *name*.

Variable	Value of Name
{AIO_LISTIO_MAX}	_SC_AIO_LISTIO_MAX
{AIO_MAX}	_SC_AIO_MAX
{AIO_PRIO_DELTA_MAX}	_SC_AIO_PRIO_DELTA_MAX
{ARG_MAX}	_SC_ARG_MAX
{ATEXIT_MAX}	_SC_ATEXIT_MAX
{BC_BASE_MAX}	_SC_BC_BASE_MAX
{BC_DIM_MAX}	_SC_BC_DIM_MAX
{BC_SCALE_MAX}	_SC_BC_SCALE_MAX
{BC_STRING_MAX}	_SC_BC_STRING_MAX
{CHILD_MAX}	_SC_CHILD_MAX
Clock ticks/second	_SC_CLK_TCK
{COLL_WEIGHTS_MAX}	_SC_COLL_WEIGHTS_MAX
{DELAYTIMER_MAX}	_SC_DELAYTIMER_MAX
{EXPR_NEST_MAX}	_SC_EXPR_NEST_MAX
{HOST_NAME_MAX}	_SC_HOST_NAME_MAX
{IOV_MAX}	_SC_IOV_MAX
{LINE_MAX}	_SC_LINE_MAX
{LOGIN_NAME_MAX}	_SC_LOGIN_NAME_MAX
{NGROUPS_MAX}	_SC_NGROUPS_MAX
Maximum size of <i>getgrgid_r()</i> and <i>getgrnam_r()</i> data buffers	_SC_GETGR_R_SIZE_MAX
Maximum size of <i>getpwuid_r()</i> and <i>getpwnam_r()</i> data buffers	_SC_GETPW_R_SIZE_MAX
{MQ_OPEN_MAX}	_SC_MQ_OPEN_MAX
{MQ_PRIO_MAX}	_SC_MQ_PRIO_MAX
{OPEN_MAX}	_SC_OPEN_MAX
_POSIX_ADVISORY_INFO	_SC_ADVISORY_INFO
_POSIX_BARRIERS	_SC_BARRIERS
_POSIX_ASYNCHRONOUS_IO	_SC_ASYNCHRONOUS_IO
_POSIX_CLOCK_SELECTION	_SC_CLOCK_SELECTION
_POSIX_CPUTIME	_SC_CPUTIME
_POSIX_FSYNC	_SC_FSYNC
_POSIX_IPV6	_SC_IPV6
_POSIX_JOB_CONTROL	_SC_JOB_CONTROL

	Variable	Value of Name
47955	_POSIX_MAPPED_FILES	_SC_MAPPED_FILES
47956	_POSIX_MEMLOCK	_SC_MEMLOCK
47957	_POSIX_MEMLOCK_RANGE	_SC_MEMLOCK_RANGE
47958	_POSIX_MEMORY_PROTECTION	_SC_MEMORY_PROTECTION
47959	_POSIX_MESSAGE_PASSING	_SC_MESSAGE_PASSING
47960	_POSIX_MONOTONIC_CLOCK	_SC_MONOTONIC_CLOCK
47961	_POSIX_PRIORITIZED_IO	_SC_PRIORITIZED_IO
47962	_POSIX_PRIORITY_SCHEDULING	_SC_PRIORITY_SCHEDULING
47963	_POSIX_RAW_SOCKETS	_SC_RAW_SOCKETS
47964	_POSIX_READER_WRITER_LOCKS	_SC_READER_WRITER_LOCKS
47965	_POSIX_REALTIME_SIGNALS	_SC_REALTIME_SIGNALS
47966	_POSIX_REGEX	_SC_REGEX
47967	_POSIX_SAVED_IDS	_SC_SAVED_IDS
47968	_POSIX_SEMAPHORES	_SC_SEMAPHORES
47969	_POSIX_SHARED_MEMORY_OBJECTS	_SC_SHARED_MEMORY_OBJECTS
47970	_POSIX_SHELL	_SC_SHELL
47971	_POSIX_SPAWN	_SC_SPAWN
47972	_POSIX_SPIN_LOCKS	_SC_SPIN_LOCKS
47973	_POSIX_SPORADIC_SERVER	_SC_SPORADIC_SERVER
47974	_POSIX_SS_REPL_MAX	_SC_SS_REPL_MAX
47975	_POSIX_SYNCHRONIZED_IO	_SC_SYNCHRONIZED_IO
47976	_POSIX_THREAD_ATTR_STACKADDR	_SC_THREAD_ATTR_STACKADDR
47977	_POSIX_THREAD_ATTR_STACKSIZE	_SC_THREAD_ATTR_STACKSIZE
47978	_POSIX_THREAD_CPUTIME	_SC_THREAD_CPUTIME
47979	_POSIX_THREAD_PRIO_INHERIT	_SC_THREAD_PRIO_INHERIT
47980	_POSIX_THREAD_PRIO_PROTECT	_SC_THREAD_PRIO_PROTECT
47981	_POSIX_THREAD_PRIORITY_SCHEDULING	_SC_THREAD_PRIORITY_SCHEDULING
47982	_POSIX_THREAD_PROCESS_SHARED	_SC_THREAD_PROCESS_SHARED
47983	_POSIX_THREAD_ROBUST_PRIO_INHERIT	_SC_THREAD_ROBUST_PRIO_INHERIT
47984	_POSIX_THREAD_ROBUST_PRIO_PROTECT	_SC_THREAD_ROBUST_PRIO_PROTECT
47985	_POSIX_THREAD_SAFE_FUNCTIONS	_SC_THREAD_SAFE_FUNCTIONS
47986	_POSIX_THREAD_SPAWNING	_SC_THREAD_SPAWNING
47987	_POSIX_THREADS	_SC_THREADS
47988	_POSIX_TIMEOUTS	_SC_TIMEOUTS
47989	_POSIX_TIMERS	_SC_TIMERS
47990	_POSIX_TRACE	_SC_TRACE
47991	_POSIX_TRACE_EVENT_FILTER	_SC_TRACE_EVENT_FILTER
47992	_POSIX_TRACE_EVENT_NAME_MAX	_SC_TRACE_EVENT_NAME_MAX
47993	_POSIX_TRACE_INHERIT	_SC_TRACE_INHERIT
47994	_POSIX_TRACE_LOG	_SC_TRACE_LOG
47995	_POSIX_TRACE_NAME_MAX	_SC_TRACE_NAME_MAX
47996	_POSIX_TRACE_SYS_MAX	_SC_TRACE_SYS_MAX
47997	_POSIX_TRACE_USER_EVENT_MAX	_SC_TRACE_USER_EVENT_MAX
47998	_POSIX_TYPED_MEMORY_OBJECTS	_SC_TYPED_MEMORY_OBJECTS
47999	_POSIX_VERSION	_SC_VERSION
48000	_POSIX_V7_ILP32_OFF32	_SC_V7_ILP32_OFF32
48001	_POSIX_V7_ILP32_OFFBIG	_SC_V7_ILP32_OFFBIG
48002	_POSIX_V7_LP64_OFF64	_SC_V7_LP64_OFF64
48003	_POSIX_V7_LP64_OFFBIG	_SC_V7_LP64_OFFBIG
48004	_POSIX_V7_LP64_OFFBIG	_SC_V7_LP64_OFFBIG

	Variable	Value of Name
48005		
48006	OB	
48007		
48008		
48009		
48010		
48011		
48012		
48013		
48014		
48015		
48016		
48017		
48018		
48019		
48020		
48021		
48022		
48023		
48024		
48025		
48026		
48027		
48028		
48029		
48030		
48031		
48032		
48033		
48034		
48035		
48036		
48037		
48038		
48039		
48040		
48041		
48042		
48043		
48044		
48045		
48046		
48047		
48048		

RETURN VALUE

48049
48050 If *name* is an invalid value, *sysconf()* shall return -1 and set *errno* to indicate the error. If the
48051 variable corresponding to *name* has no limit, *sysconf()* shall return -1 without changing the value
48052 of *errno*. Note that indefinite limits do not imply infinite limits; see **<limits.h>**.

48053 Otherwise, *sysconf()* shall return the current variable value on the system. The value returned
48054 shall not be more restrictive than the corresponding value described to the application when it
48055 was compiled with the implementation's **<limits.h>** or **<unistd.h>**. The value shall not change
48056 during the lifetime of the calling process, except that *sysconf(SC_OPEN_MAX)* may return
48057 different values before and after a call to *setrlimit()* which changes the RLIMIT_NOFILE soft

48058
48059
48060
48061
48062
48063
48064
48065
48066
48067
48068
48069
48070
48071
48072
48073
48074
48075
48076
48077
48078
48079
48080
48081
48082
48083
48084
48085
48086
48087
48088
48089
48090
48091
48092
48093
48094
48095
48096
48097
48098
48099
48100
48101
48102
48103
48104

limit.

If the variable corresponding to *name* is dependent on an unsupported option, the results are unspecified.

ERRORS

The *sysconf()* function shall fail if:

[EINVAL] The value of the *name* argument is invalid.

EXAMPLES

None.

APPLICATION USAGE

As -1 is a permissible return value in a successful situation, an application wishing to check for error situations should set *errno* to 0, then call *sysconf()*, and, if it returns -1 , check to see if *errno* is non-zero.

Application writers should check whether an option, such as `_POSIX_TRACE`, is supported prior to obtaining and using values for related variables, such as `_POSIX_TRACE_NAME_MAX`.

RATIONALE

This functionality was added in response to requirements of application developers and of system vendors who deal with many international system configurations. It is closely related to *pathconf()* and *fpathconf()*.

Although a conforming application can run on all systems by never demanding more resources than the minimum values published in this volume of IEEE Std 1003.1-200x, it is useful for that application to be able to use the actual value for the quantity of a resource available on any given system. To do this, the application makes use of the value of a symbolic constant in `<limits.h>` or `<unistd.h>`.

However, once compiled, the application must still be able to cope if the amount of resource available is increased. To that end, an application may need a means of determining the quantity of a resource, or the presence of an option, at execution time.

Two examples are offered:

1. Applications may wish to act differently on systems with or without job control. Applications vendors who wish to distribute only a single binary package to all instances of a computer architecture would be forced to assume job control is never available if it were to rely solely on the `<unistd.h>` value published in this volume of IEEE Std 1003.1-200x.
2. International applications vendors occasionally require knowledge of the number of clock ticks per second. Without these facilities, they would be required to either distribute their applications partially in source form or to have 50 Hz and 60 Hz versions for the various countries in which they operate.

It is the knowledge that many applications are actually distributed widely in executable form that leads to this facility. If limited to the most restrictive values in the headers, such applications would have to be prepared to accept the most limited environments offered by the smallest microcomputers. Although this is entirely portable, there was a consensus that they should be able to take advantage of the facilities offered by large systems, without the restrictions associated with source and object distributions.

During the discussions of this feature, it was pointed out that it is almost always possible for an application to discern what a value might be at runtime by suitably testing the various functions themselves. And, in any event, it could always be written to adequately deal with error returns from the various functions. In the end, it was felt that this imposed an unreasonable level of complication and sophistication on the application writer.

48105 This runtime facility is not meant to provide ever-changing values that applications have to
 48106 check multiple times. The values are seen as changing no more frequently than once per system
 48107 initialization, such as by a system administrator or operator with an automatic configuration
 48108 program. This volume of IEEE Std 1003.1-200x specifies that they shall not change within the
 48109 lifetime of the process.

48110 Some values apply to the system overall and others vary at the file system or directory level. The
 48111 latter are described in *pathconf()*.

48112 Note that all values returned must be expressible as integers. String values were considered, but
 48113 the additional flexibility of this approach was rejected due to its added complexity of
 48114 implementation and use.

48115 Some values, such as {PATH_MAX}, are sometimes so large that they must not be used to, say,
 48116 allocate arrays. The *sysconf()* function returns a negative value to show that this symbolic
 48117 constant is not even defined in this case.

48118 Similar to *pathconf()*, this permits the implementation not to have a limit. When one resource is
 48119 infinite, returning an error indicating that some other resource limit has been reached is
 48120 conforming behavior.

48121 FUTURE DIRECTIONS

48122 None.

48123 SEE ALSO

48124 *confstr()*, *pathconf()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<limits.h>`,
 48125 `<unistd.h>`, the Shell and Utilities volume of IEEE Std 1003.1-200x, *getconf*

48126 CHANGE HISTORY

48127 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

48128 Issue 5

48129 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
 48130 Threads Extension.

48131 The `_XBS_` variables and name values are added to the table of system variables in the
 48132 DESCRIPTION. These are all marked EX.

48133 Issue 6

48134 The symbol `CLK_TCK` is obsolescent and removed. It is replaced with the phrase “clock ticks
 48135 per second”.

48136 The symbol {PASS_MAX} is removed.

48137 The following changes were made to align with the IEEE P1003.1a draft standard:

- 48138 • Table entries are added for the following variables: `_SC_REGEX`, `_SC_SHELL`,
- 48139 `_SC_REGEX_VERSION`, `_SC_SYMLINK_MAX`.

48140 The following *sysconf()* variables and their associated names are added for alignment with
 48141 IEEE Std 1003.1d-1999:

```
48142     _POSIX_ADVISORY_INFO
48143     _POSIX_CPUTIME
48144     _POSIX_SPAWN
48145     _POSIX_SPARADIC_SERVER
48146     _POSIX_THREAD_CPUTIME
48147     _POSIX_THREAD_SPARADIC_SERVER
48148     _POSIX_TIMEOUTS
```

48149 The following changes are made to the DESCRIPTION for alignment with IEEE Std 1003.1j-2000:

- 48150 • A statement expressing the dependency of support for some system variables on
48151 implementation options is added.
- 48152 • The following system variables are added:
- 48153 _POSIX_BARRIERS
48154 _POSIX_CLOCK_SELECTION
48155 _POSIX_MONOTONIC_CLOCK
48156 _POSIX_READER_WRITER_LOCKS
48157 _POSIX_SPIN_LOCKS
48158 _POSIX_TYPED_MEMORY_OBJECTS

48159 The following system variables are added for alignment with IEEE Std 1003.2d-1994:

48160 _POSIX2_PBS
48161 _POSIX2_PBS_ACCOUNTING
48162 _POSIX2_PBS_LOCATE
48163 _POSIX2_PBS_MESSAGE
48164 _POSIX2_PBS_TRACK

48165 The following *sysconf()* variables and their associated names are added for alignment with
48166 IEEE Std 1003.1q-2000:

48167 _POSIX_TRACE
48168 _POSIX_TRACE_EVENT_FILTER
48169 _POSIX_TRACE_INHERIT
48170 _POSIX_TRACE_LOG

48171 The macros associated with the *c89* programming models are marked LEGACY, and new
48172 equivalent macros associated with *c99* are introduced.

48173 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/62 is applied, updating the
48174 DESCRIPTION to denote that the *_PC** and *_SC** symbols are now required to be supported. A
48175 corresponding change has been made in the Base Definitions volume of IEEE Std 1003.1-200x.
48176 The deletion in the second paragraph removes some duplicated text. Additional symbols that
48177 were erroneously omitted from this reference page have been added.

48178 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/63 is applied, making it clear in the
48179 RETURN VALUE section that the value returned for *sysconf(_SC_OPEN_MAX)* may change if a
48180 call to *setrlimit()* adjusts the RLIMIT_NOFILE soft limit.

48181 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/134 is applied, updating the
48182 DESCRIPTION to remove an erroneous entry for *_POSIX_SYMLINK_MAX*. This corrects an
48183 error in IEEE Std 1003.1-2001/Cor 1-2002.

48184 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/135 is applied, removing
48185 *_POSIX_FILE_LOCKING*, *_POSIX_MULTI_PROCESS*, *_POSIX2_C_VERSION*, and
48186 *_XOPEN_XCU_VERSION* (and their associated *_SC_** variables) from the DESCRIPTION and
48187 APPLICATION USAGE sections.

48188 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/136 is applied, adding the following
48189 constants (and their associated *_SC_** variables) to the DESCRIPTION:

48190 _POSIX_SS_REPL_MAX
48191 _POSIX_TRACE_EVENT_NAME_MAX
48192 _POSIX_TRACE_NAME_MAX
48193 _POSIX_TRACE_SYS_MAX
48194 _POSIX_TRACE_USER_EVENT_MAX

48195 The RETURN VALUE and APPLICATION USAGE sections are updated to note that if variables
48196 are dependent on unsupported options, the results are unspecified.

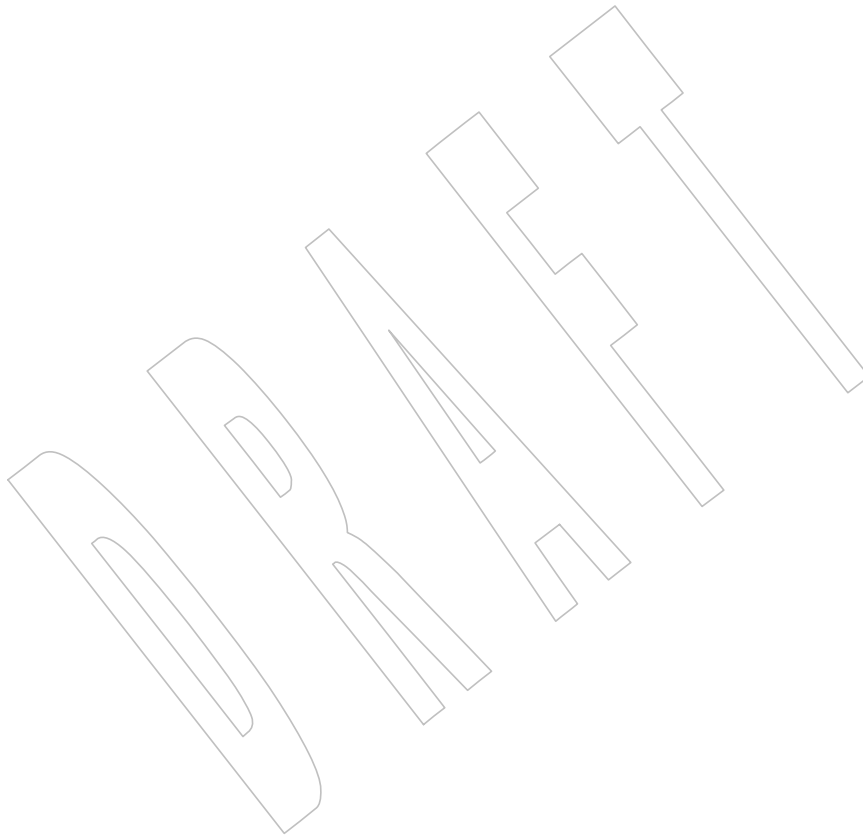
48197 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/137 is applied, removing
48198 _REGEX_VERSION and _SC_REGEX_VERSION.

48199 **Issue 7**

48200 The variables for the supported programming environments are updated to be V7 and the
48201 LEGACY variables are removed.

48202 The following constants are added:

48203 _POSIX_THREAD_ROBUST_PRIO_INHERIT
48204 _POSIX_THREAD_ROBUST_PRIO_PROTECT



48205 **NAME**
48206 syslog — log a message

48207 **SYNOPSIS**

```
48208 XSI #include <syslog.h>  
48209 void syslog(int priority, const char *message, ... /* argument */);
```

48210 **DESCRIPTION**

48211 Refer to *closelog()*.

48212 **NAME**

48213 system — issue a command

48214 **SYNOPSIS**

48215 #include <stdlib.h>

48216 int system(const char *command);

48217 **DESCRIPTION**

48218 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 48219 conflict between the requirements described here and the ISO C standard is unintentional. This
 48220 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

48221 If *command* is a null pointer, the *system()* function shall determine whether the host environment
 48222 has a command processor. If *command* is not a null pointer, the *system()* function shall pass the
 48223 string pointed to by *command* to that command processor to be executed in an implementation-
 48224 defined manner; this might then cause the program calling *system()* to behave in a non-
 48225 conforming manner or to terminate.

48226 CX The *system()* function shall behave as if a child process were created using *fork()*, and the child
 48227 process invoked the *sh* utility using *execl()* as follows:

48228

```
execl(<shell path>, "sh", "-c", command, (char *)0);
```

48229 where <shell path> is an unspecified pathname for the *sh* utility. It is unspecified whether the
 48230 handlers registered with *pthread_atfork()* are called as part of the creation of the child process.

48231 The *system()* function shall ignore the SIGINT and SIGQUIT signals, and shall block the
 48232 SIGCHLD signal, while waiting for the command to terminate. If this might cause the
 48233 application to miss a signal that would have killed it, then the application should examine the
 48234 return value from *system()* and take whatever action is appropriate to the application if the
 48235 command terminated due to receipt of a signal.

48236 The *system()* function shall not affect the termination status of any child of the calling processes
 48237 other than the process or processes it itself creates.

48238 The *system()* function shall not return until the child process has terminated.

48239 The *system()* function need not be thread-safe. A function that is not required to be thread-safe is
 48240 not required to be reentrant.

48241 **RETURN VALUE**

48242 If *command* is a null pointer, *system()* shall return non-zero to indicate that a command processor
 48243 CX is available, or zero if none is available. The *system()* function shall always return non-zero
 48244 when *command* is NULL.

48245 CX If *command* is not a null pointer, *system()* shall return the termination status of the command
 48246 language interpreter in the format specified by *waitpid()*. The termination status shall be as
 48247 defined for the *sh* utility; otherwise, the termination status is unspecified. If some error prevents
 48248 the command language interpreter from executing after the child process is created, the return
 48249 value from *system()* shall be as if the command language interpreter had terminated using
 48250 *exit(127)* or *_exit(127)*. If a child process cannot be created, or if the termination status for the
 48251 command language interpreter cannot be obtained, *system()* shall return -1 and set *errno* to
 48252 indicate the error.

48253 **ERRORS**48254 CX The *system()* function may set *errno* values as described by *fork()*.48255 In addition, *system()* may fail if:48256 CX [ECHILD] The status of the child process created by *system()* is no longer available.48257 **EXAMPLES**

48258 None.

48259 **APPLICATION USAGE**48260 If the return value of *system()* is not -1 , its value can be decoded through the use of the macros
48261 described in `<sys/wait.h>`. For convenience, these macros are also provided in `<stdlib.h>`.48262 Note that, while *system()* must ignore SIGINT and SIGQUIT and block SIGCHLD while waiting
48263 for the child to terminate, the handling of signals in the executed command is as specified by
48264 *fork()* and *exec*. For example, if SIGINT is being caught or is set to SIG_DFL when *system()* is
48265 called, then the child is started with SIGINT handling set to SIG_DFL.48266 Ignoring SIGINT and SIGQUIT in the parent process prevents coordination problems (two
48267 processes reading from the same terminal, for example) when the executed command ignores or
48268 catches one of the signals. It is also usually the correct action when the user has given a
48269 command to the application to be executed synchronously (as in the '!' command in many
48270 interactive applications). In either case, the signal should be delivered only to the child process,
48271 not to the application itself. There is one situation where ignoring the signals might have less
48272 than the desired effect. This is when the application uses *system()* to perform some task invisible
48273 to the user. If the user typed the interrupt character ("^C", for example) while *system()* is being
48274 used in this way, one would expect the application to be killed, but only the executed command
48275 is killed. Applications that use *system()* in this way should carefully check the return status from
48276 *system()* to see if the executed command was successful, and should take appropriate action
48277 when the command fails.48278 Blocking SIGCHLD while waiting for the child to terminate prevents the application from
48279 catching the signal and obtaining status from *system()*'s child process before *system()* can get the
48280 status itself.48281 The context in which the utility is ultimately executed may differ from that in which *system()*
48282 was called. For example, file descriptors that have the FD_CLOEXEC flag set are closed, and the
48283 process ID and parent process ID are different. Also, if the executed utility changes its
48284 environment variables or its current working directory, that change is not reflected in the caller's
48285 context.48286 There is no defined way for an application to find the specific path for the shell. However,
48287 *confstr()* can provide a value for *PATH* that is guaranteed to find the *sh* utility.48288 Using the *system()* function in more than one thread in a process or when the SIGCHLD signal is
48289 being manipulated by more than one thread in a process may produce unexpected results.48290 **RATIONALE**48291 The *system()* function should not be used by programs that have set user (or group) ID
48292 privileges. The *fork()* and *exec* family of functions (except *execlp()* and *execvp()*), should be used
48293 instead. This prevents any unforeseen manipulation of the environment of the user that could
48294 cause execution of commands not anticipated by the calling program.48295 There are three levels of specification for the *system()* function. The ISO C standard gives the
48296 most basic. It requires that the function exists, and defines a way for an application to query
48297 whether a command language interpreter exists. It says nothing about the command language
48298 or the environment in which the command is interpreted.48299 IEEE Std 1003.1-200x places additional restrictions on *system()*. It requires that if there is a
48300 command language interpreter, the environment must be as specified by *fork()* and *exec*. This

48301 ensures, for example, that *close-on-exec* works, that file locks are not inherited, and that the
 48302 process ID is different. It also specifies the return value from *system()* when the command line
 48303 can be run, thus giving the application some information about the command's completion
 48304 status.

48305 Finally, IEEE Std 1003.1-200x requires the command to be interpreted as in the shell command
 48306 language defined in the Shell and Utilities volume of IEEE Std 1003.1-200x.

48307 Note that, *system(NULL)* is required to return non-zero, indicating that there is a command
 48308 language interpreter. At first glance, this would seem to conflict with the ISO C standard which
 48309 allows *system(NULL)* to return zero. There is no conflict, however. A system must have a
 48310 command language interpreter, and is non-conforming if none is present. It is therefore
 48311 permissible for the *system()* function on such a system to implement the behavior specified by
 48312 the ISO C standard as long as it is understood that the implementation does not conform to
 48313 IEEE Std 1003.1-200x if *system(NULL)* returns zero.

48314 It was explicitly decided that when *command* is NULL, *system()* should not be required to check
 48315 to make sure that the command language interpreter actually exists with the correct mode, that
 48316 there are enough processes to execute it, and so on. The call *system(NULL)* could, theoretically,
 48317 check for such problems as too many existing child processes, and return zero. However, it
 48318 would be inappropriate to return zero due to such a (presumably) transient condition. If some
 48319 condition exists that is not under the control of this application and that would cause any
 48320 *system()* call to fail, that system has been rendered non-conforming.

48321 Early drafts required, or allowed, *system()* to return with *errno* set to [EINTR] if it was
 48322 interrupted with a signal. This error return was removed, and a requirement that *system()* not
 48323 return until the child has terminated was added. This means that if a *waitpid()* call in *system()*
 48324 exits with *errno* set to [EINTR], *system()* must reissue the *waitpid()*. This change was made for
 48325 two reasons:

- 48326 1. There is no way for an application to clean up if *system()* returns [EINTR], short of calling
 48327 *wait()*, and that could have the undesirable effect of returning the status of children other
 48328 than the one started by *system()*.
- 48329 2. While it might require a change in some historical implementations, those
 48330 implementations already have to be changed because they use *wait()* instead of *waitpid()*.

48331 Note that if the application is catching SIGCHLD signals, it will receive such a signal before a
 48332 successful *system()* call returns.

48333 To conform to IEEE Std 1003.1-200x, *system()* must use *waitpid()*, or some similar function,
 48334 instead of *wait()*.

48335 The following code sample illustrates how *system()* might be implemented on an
 48336 implementation conforming to IEEE Std 1003.1-200x.

```

48337 #include <signal.h>
48338 int system(const char *cmd)
48339 {
48340     int stat;
48341     pid_t pid;
48342     struct sigaction sa, savintr, savequit;
48343     sigset_t saveblock;
48344     if (cmd == NULL)
48345         return(1);
48346     sa.sa_handler = SIG_IGN;
48347     sigemptyset(&sa.sa_mask);
48348     sa.sa_flags = 0;
48349     sigemptyset(&savintr.sa_mask);
  
```

```

48350     sigemptyset(&savequit.sa_mask);
48351     sigaction(SIGINT, &sa, &saveintr);
48352     sigaction(SIGQUIT, &sa, &savequit);
48353     sigaddset(&sa.sa_mask, SIGCHLD);
48354     sigprocmask(SIG_BLOCK, &sa.sa_mask, &saveblock);
48355     if ((pid = fork()) == 0) {
48356         sigaction(SIGINT, &saveintr, (struct sigaction *)0);
48357         sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
48358         sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
48359         execl("/bin/sh", "sh", "-c", cmd, (char *)0);
48360         _exit(127);
48361     }
48362     if (pid == -1) {
48363         stat = -1; /* errno comes from fork() */
48364     } else {
48365         while (waitpid(pid, &stat, 0) == -1) {
48366             if (errno != EINTR){
48367                 stat = -1;
48368                 break;
48369             }
48370         }
48371     }
48372     sigaction(SIGINT, &saveintr, (struct sigaction *)0);
48373     sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
48374     sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
48375     return(stat);
48376 }

```

48377 Note that, while a particular implementation of *system()* (such as the one above) can assume a
48378 particular path for the shell, such a path is not necessarily valid on another system. The above
48379 example is not portable, and is not intended to be.

48380 One reviewer suggested that an implementation of *system()* might want to use an environment
48381 variable such as *SHELL* to determine which command interpreter to use. The supposed
48382 implementation would use the default command interpreter if the one specified by the
48383 environment variable was not available. This would allow a user, when using an application that
48384 prompts for command lines to be processed using *system()*, to specify a different command
48385 interpreter. Such an implementation is discouraged. If the alternate command interpreter did not
48386 follow the command line syntax specified in the Shell and Utilities volume of
48387 IEEE Std 1003.1-200x, then changing *SHELL* would render *system()* non-conforming. This would
48388 affect applications that expected the specified behavior from *system()*, and since the Shell and
48389 Utilities volume of IEEE Std 1003.1-200x does not mention that *SHELL* affects *system()*, the
48390 application would not know that it needed to unset *SHELL*.

48391 FUTURE DIRECTIONS

48392 None.

48393 SEE ALSO

48394 *exec*, *pipe()*, *pthread_atfork()*, *waitpid()*, the Base Definitions volume of IEEE Std 1003.1-200x,
48395 *<limits.h>*, *<signal.h>*, *<stdlib.h>*, *<sys/wait.h>*, the Shell and Utilities volume of
48396 IEEE Std 1003.1-200x, *sh*

48397 CHANGE HISTORY

48398 First released in Issue 1. Derived from Issue 1 of the SVID.

48399

Issue 6

48400

Extensions beyond the ISO C standard are marked.

48401

Issue 7

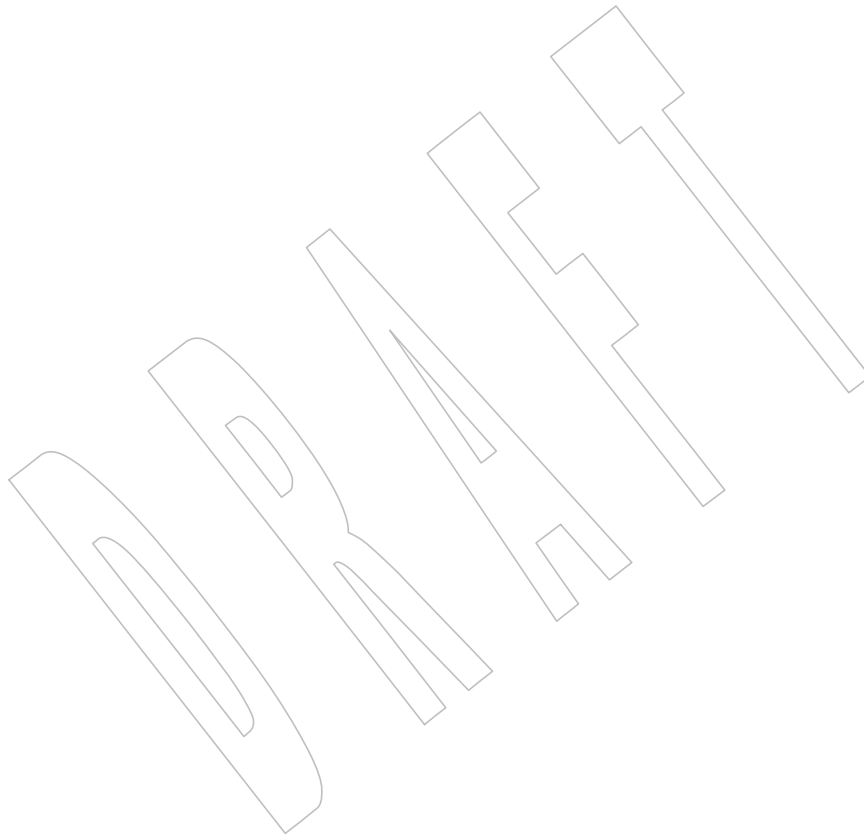
48402

SD5-XSH-ERN-30 is applied.

48403

Austin Group Interpretation 1003.1-2001 #055 is applied, clarifying the thread-safety of this function and treatment of *at_fork()* handlers.

48404



48405 **NAME**

48406 tan, tanf, tanl — tangent function

48407 **SYNOPSIS**

```
48408 #include <math.h>
48409
48409 double tan(double x);
48410 float tanf(float x);
48411 long double tanl(long double x);
```

48412 **DESCRIPTION**

48413 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 48414 conflict between the requirements described here and the ISO C standard is unintentional. This
 48415 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

48416 These functions shall compute the tangent of their argument x , measured in radians.

48417 An application wishing to check for error situations should set *errno* to zero and call
 48418 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 48419 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 48420 zero, an error has occurred.

48421 **RETURN VALUE**

48422 Upon successful completion, these functions shall return the tangent of x .

48423 If the correct value would cause underflow, and is not representable, a range error may occur,
 48424 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

48425 MX If x is NaN, a NaN shall be returned.

48426 If x is ± 0 , x shall be returned.

48427 If x is subnormal, a range error may occur and x should be returned.

48428 If x is $\pm\text{Inf}$, a domain error shall occur, and either a NaN (if supported), or an implementation-
 48429 defined value shall be returned.

48430 If the correct value would cause underflow, and is representable, a range error may occur and
 48431 the correct value shall be returned.

48432 XSI If the correct value would cause overflow, a range error shall occur and *tan()*, *tanf()*, and *tanl()*
 48433 shall return $\pm\text{HUGE_VAL}$, $\pm\text{HUGE_VALF}$, and $\pm\text{HUGE_VALL}$, respectively, with the same sign
 48434 as the correct value of the function.

48435 **ERRORS**

48436 These functions shall fail if:

48437 MX **Domain Error** The value of x is $\pm\text{Inf}$.

48438 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 48439 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 48440 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 48441 shall be raised.

48442 XSI **Range Error** The result overflows

48443 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 48444 then *errno* shall be set to [ERANGE]. If the integer expression
 48445 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
 48446 floating-point exception shall be raised.

48447 These functions may fail if:

48448 MX Range Error The result underflows, or the value of x is subnormal.

48449 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 48450 then *errno* shall be set to [ERANGE]. If the integer expression
 48451 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 48452 floating-point exception shall be raised.

48453 EXAMPLES

48454 Taking the Tangent of a 45-Degree Angle

```
48455 #include <math.h>
48456 ...
48457 double radians = 45.0 * M_PI / 180;
48458 double result;
48459 ...
48460 result = tan (radians);
```

48461 APPLICATION USAGE

48462 There are no known floating-point representations such that for a normal argument, $\tan(x)$ is
 48463 either overflow or underflow.

48464 These functions may lose accuracy when their argument is near a multiple of $\pi/2$ or is far from
 48465 0.0.

48466 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
 48467 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

48468 RATIONALE

48469 None.

48470 FUTURE DIRECTIONS

48471 None.

48472 SEE ALSO

48473 *atan()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-200x,
 48474 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

48475 CHANGE HISTORY

48476 First released in Issue 1. Derived from Issue 1 of the SVID.

48477 Issue 5

48478 The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes
 48479 in previous issues.

48480 Issue 6

48481 The *tanf()* and *tanl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

48482 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
 48483 revised to align with the ISO/IEC 9899:1999 standard.

48484 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
 48485 marked.

48486 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/64 is applied, correcting the last
 48487 paragraph in the RETURN VALUE section.

48488 **NAME**
 48489 tanh, tanhf, tanhl — hyperbolic tangent functions

48490 **SYNOPSIS**
 48491 #include <math.h>
 48492 double tanh(double x);
 48493 float tanhf(float x);
 48494 long double tanhl(long double x);

48495 **DESCRIPTION**
 48496 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 48497 conflict between the requirements described here and the ISO C standard is unintentional. This
 48498 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

48499 These functions shall compute the hyperbolic tangent of their argument x .

48500 An application wishing to check for error situations should set *errno* to zero and call
 48501 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 48502 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 48503 zero, an error has occurred.

48504 **RETURN VALUE**
 48505 Upon successful completion, these functions shall return the hyperbolic tangent of x .

48506 MX If x is NaN, a NaN shall be returned.
 48507 If x is ± 0 , x shall be returned.
 48508 If x is $\pm \text{Inf}$, ± 1 shall be returned.
 48509 If x is subnormal, a range error may occur and x should be returned.

48510 **ERRORS**
 48511 These functions may fail if:

48512 MX **Range Error** The value of x is subnormal.
 48513 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 48514 then *errno* shall be set to [ERANGE]. If the integer expression
 48515 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 48516 floating-point exception shall be raised.

48517 **EXAMPLES**
 48518 None.

48519 **APPLICATION USAGE**
 48520 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
 48521 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

48522 **RATIONALE**
 48523 None.

48524 **FUTURE DIRECTIONS**
 48525 None.

48526 **SEE ALSO**
 48527 *atanh()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, *tan()*, the Base Definitions volume of
 48528 IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
 48529 <math.h>

48530

CHANGE HISTORY

48531

First released in Issue 1. Derived from Issue 1 of the SVID.

48532

Issue 5

48533

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

48534

48535

Issue 6

48536

The *tanhf()* and *tanhl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

48537

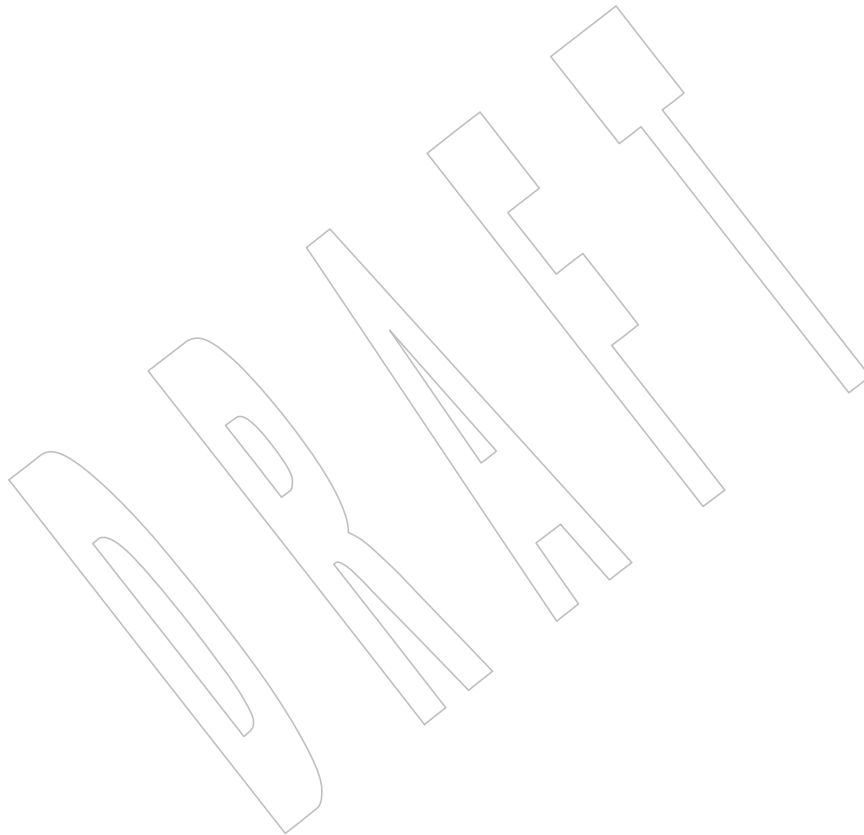
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

48538

48539

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

48540



48541 **NAME**
48542 tanl — tangent function

48543 **SYNOPSIS**
48544 #include <math.h>
48545 long double tanl(long double x);

48546 **DESCRIPTION**
48547 Refer to *tan()*.



48548 **NAME**
 48549 `tcdrain` — wait for transmission of output

48550 **SYNOPSIS**
 48551 `#include <termios.h>`
 48552 `int tcdrain(int fildev);`

48553 **DESCRIPTION**
 48554 The `tcdrain()` function shall block until all output written to the object referred to by *fildev* is
 48555 transmitted. The *fildev* argument is an open file descriptor associated with a terminal.

48556 Any attempts to use `tcdrain()` from a process which is a member of a background process group
 48557 on a *fildev* associated with its controlling terminal, shall cause the process group to be sent a
 48558 SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process
 48559 shall be allowed to perform the operation, and no signal is sent.

48560 **RETURN VALUE**
 48561 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
 48562 indicate the error.

48563 **ERRORS**
 48564 The `tcdrain()` function shall fail if:
 48565 [EBADF] The *fildev* argument is not a valid file descriptor.
 48566 [EINTR] A signal interrupted `tcdrain()`.
 48567 [ENOTTY] The file associated with *fildev* is not a terminal.
 48568 The `tcdrain()` function may fail if:
 48569 [EIO] The process group of the writing process is orphaned, and the writing process
 48570 is not ignoring or blocking SIGTTOU.

48571 **EXAMPLES**
 48572 None.

48573 **APPLICATION USAGE**
 48574 None.

48575 **RATIONALE**
 48576 None.

48577 **FUTURE DIRECTIONS**
 48578 None.

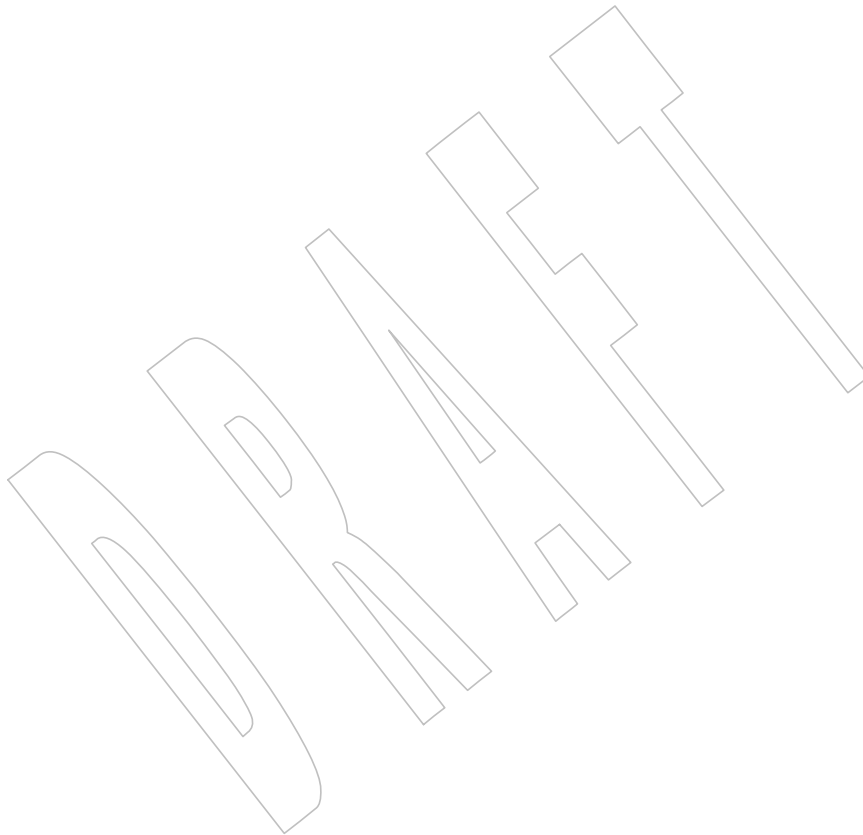
48579 **SEE ALSO**
 48580 `tcflush()`, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal
 48581 Interface, `<termios.h>`, `<unistd.h>`

48582 **CHANGE HISTORY**
 48583 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

48584
48585
48586
48587
48588
48589**Issue 6**

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the final paragraph is no longer conditional on `_POSIX_JOB_CONTROL`. This is a FIPS requirement.
- The [EIO] error is added.



48590 **NAME**
 48591 tcflow — suspend or restart the transmission or reception of data

48592 **SYNOPSIS**
 48593 #include <termios.h>
 48594 int tcflow(int *fildev*, int *action*);

48595 **DESCRIPTION**
 48596 The *tcflow()* function shall suspend or restart transmission or reception of data on the object
 48597 referred to by *fildev*, depending on the value of *action*. The *fildev* argument is an open file
 48598 descriptor associated with a terminal.

- 48599 • If *action* is TCOOFF, output shall be suspended.
- 48600 • If *action* is TCOON, suspended output shall be restarted.
- 48601 • If *action* is TCIOFF, the system shall transmit a STOP character, which is intended to cause
 48602 the terminal device to stop transmitting data to the system.
- 48603 • If *action* is TCION, the system shall transmit a START character, which is intended to cause
 48604 the terminal device to start transmitting data to the system.

48605 The default on the opening of a terminal file is that neither its input nor its output are
 48606 suspended.

48607 Attempts to use *tcflow()* from a process which is a member of a background process group on a
 48608 *fildev* associated with its controlling terminal, shall cause the process group to be sent a
 48609 SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process
 48610 shall be allowed to perform the operation, and no signal is sent.

48611 **RETURN VALUE**
 48612 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
 48613 indicate the error.

48614 **ERRORS**
 48615 The *tcflow()* function shall fail if:

48616 [EBADF]	The <i>fildev</i> argument is not a valid file descriptor.
48617 [EINVAL]	The <i>action</i> argument is not a supported value.
48618 [ENOTTY]	The file associated with <i>fildev</i> is not a terminal.

48619 The *tcflow()* function may fail if:

48620 [EIO]	The process group of the writing process is orphaned, and the writing process 48621 is not ignoring or blocking SIGTTOU.
-------------	---

48622 **EXAMPLES**
 48623 None.

48624 **APPLICATION USAGE**
 48625 None.

48626 **RATIONALE**
 48627 None.

48628

FUTURE DIRECTIONS

48629

None.

48630

SEE ALSO

48631

tcsendbreak(), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface, `<termios.h>`, `<unistd.h>`

48632

48633

CHANGE HISTORY

48634

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

48635

Issue 6

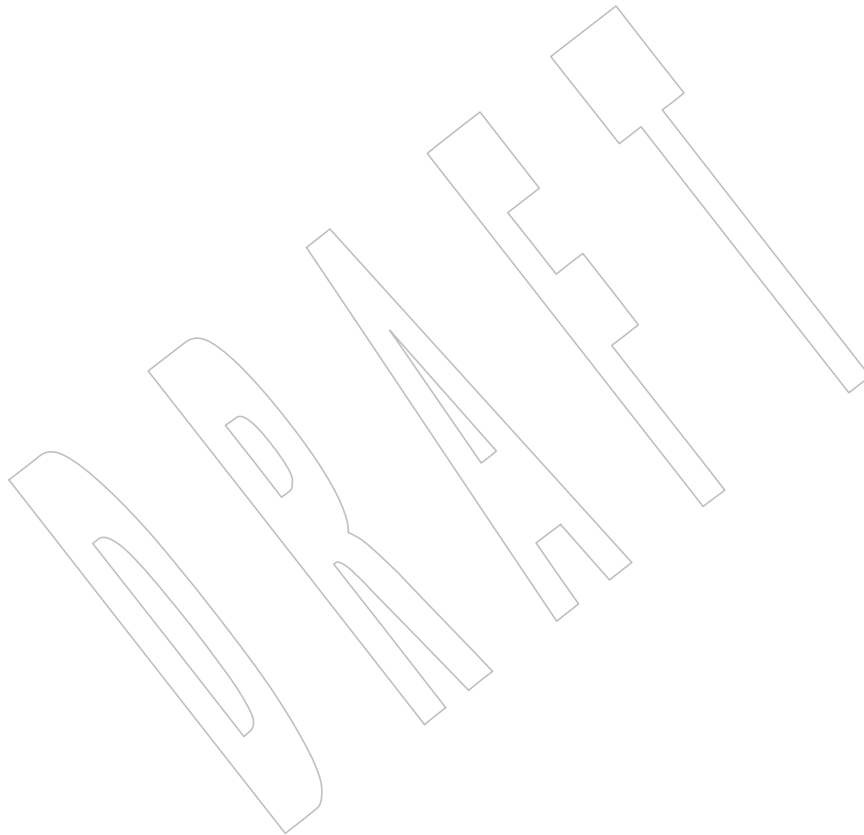
48636

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

48637

- The [EIO] error is added.

48638



48639 **NAME**

48640 tcflush — flush non-transmitted output data, non-read input data, or both

48641 **SYNOPSIS**

48642 #include <termios.h>

48643 int tcflush(int *fildev*, int *queue_selector*);48644 **DESCRIPTION**48645 Upon successful completion, *tcflush()* shall discard data written to the object referred to by *fildev*
48646 (an open file descriptor associated with a terminal) but not transmitted, or data received but not
48647 read, depending on the value of *queue_selector*:

- 48648
- If *queue_selector* is TCIFLUSH, it shall flush data received but not read.
 - If *queue_selector* is TCOFLUSH, it shall flush data written but not transmitted.
 - If *queue_selector* is TCIOFLUSH, it shall flush both data received but not read and data
48651 written but not transmitted.

48652 Attempts to use *tcflush()* from a process which is a member of a background process group on a
48653 *fildev* associated with its controlling terminal shall cause the process group to be sent a SIGTTOU
48654 signal. If the calling process is blocking or ignoring SIGTTOU signals, the process shall be
48655 allowed to perform the operation, and no signal is sent.48656 **RETURN VALUE**48657 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
48658 indicate the error.48659 **ERRORS**48660 The *tcflush()* function shall fail if:

- 48661 [EBADF] The
- fildev*
- argument is not a valid file descriptor.
-
- 48662 [EINVAL] The
- queue_selector*
- argument is not a supported value.
-
- 48663 [ENOTTY] The file associated with
- fildev*
- is not a terminal.

48664 The *tcflush()* function may fail if:

- 48665 [EIO] The process group of the writing process is orphaned, and the writing process
-
- 48666 is not ignoring or blocking SIGTTOU.

48667 **EXAMPLES**

48668 None.

48669 **APPLICATION USAGE**

48670 None.

48671 **RATIONALE**

48672 None.

48673 **FUTURE DIRECTIONS**

48674 None.

48675 **SEE ALSO**48676 *tcdrain()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal
48677 Interface, <termios.h>, <unistd.h>

48678

CHANGE HISTORY

48679

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

48680

Issue 6

48681

The Open Group Corrigendum U035/1 is applied. In the ERRORS and APPLICATION USAGE sections, references to *tcflow()* are replaced with *tcflush()*.

48682

48683

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

48684

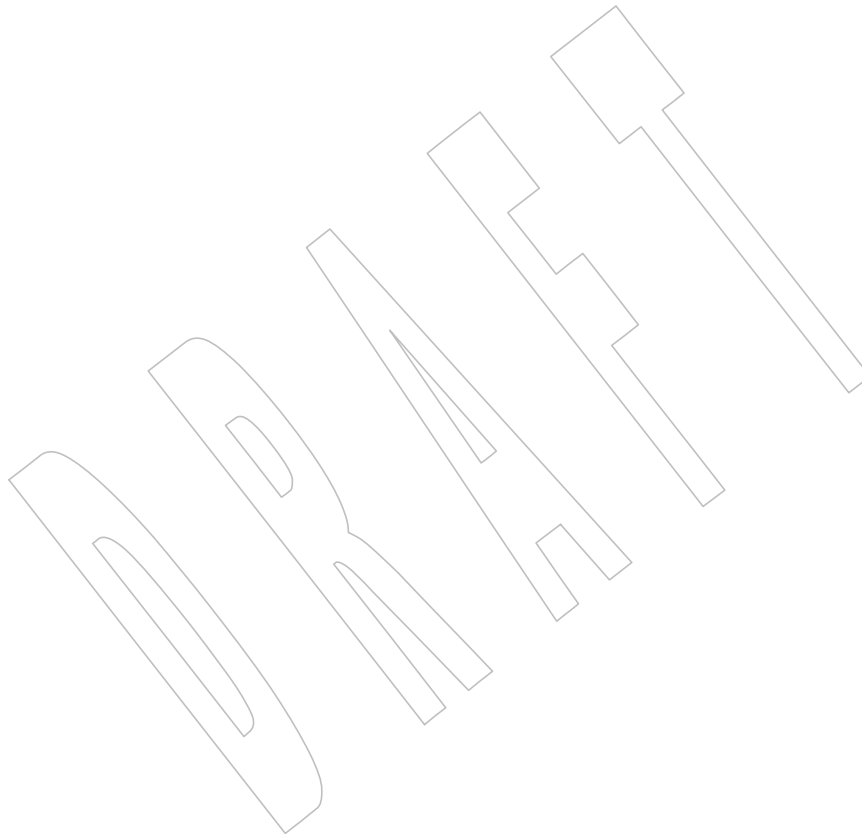
48685

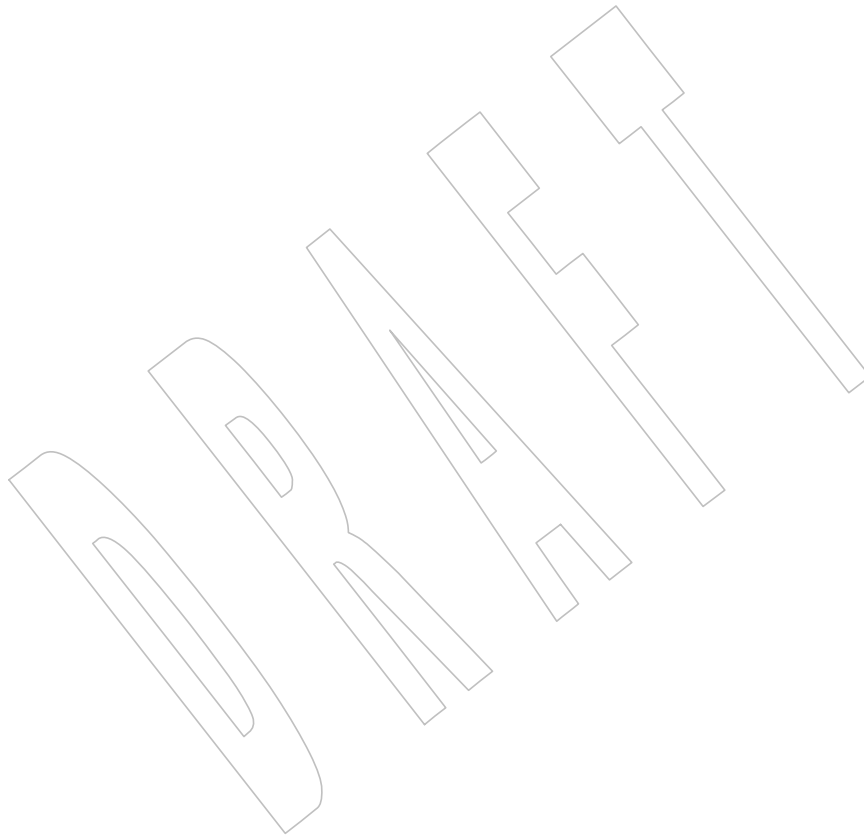
- In the DESCRIPTION, the final paragraph is no longer conditional on `_POSIX_JOB_CONTROL`. This is a FIPS requirement.

48686

- The [EIO] error is added.

48687





48732 succeeded even if some of the changes were not made. When `-1` is returned, it implies
48733 everything failed even though some of the changes were made.

48734 Applications that need all of the requested changes made to work properly should follow
48735 `tcsetattr()` with a call to `tcgetattr()` and compare the appropriate field values.

48736 FUTURE DIRECTIONS

48737 None.

48738 SEE ALSO

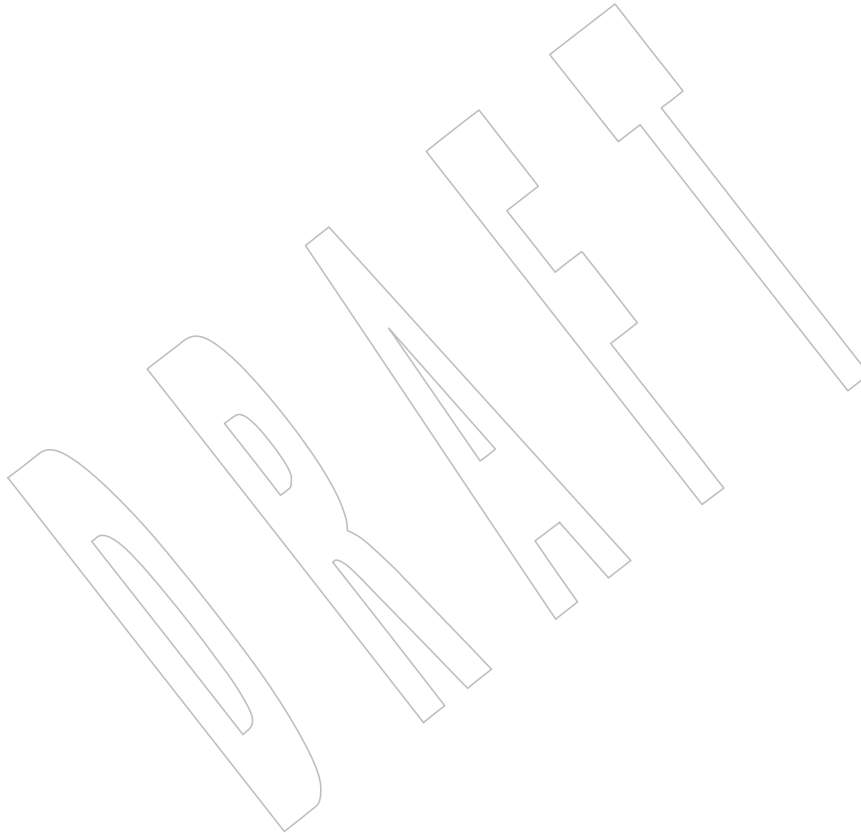
48739 `tcsetattr()`, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal
48740 Interface, `<termios.h>`

48741 CHANGE HISTORY

48742 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

48743 Issue 6

48744 In the DESCRIPTION, the rate returned as the input baud rate shall be the output rate.
48745 Previously, the number zero was also allowed but was obsolescent.



48746 **NAME**
 48747 `tcgetpgrp` — get the foreground process group ID

48748 **SYNOPSIS**
 48749 `#include <unistd.h>`
 48750 `pid_t tcgetpgrp(int fildev);`

48751 **DESCRIPTION**
 48752 The `tcgetpgrp()` function shall return the value of the process group ID of the foreground process
 48753 group associated with the terminal.

48754 If there is no foreground process group, `tcgetpgrp()` shall return a value greater than 1 that does
 48755 not match the process group ID of any existing process group.

48756 The `tcgetpgrp()` function is allowed from a process that is a member of a background process
 48757 group; however, the information may be subsequently changed by a process that is a member of
 48758 a foreground process group.

48759 **RETURN VALUE**
 48760 Upon successful completion, `tcgetpgrp()` shall return the value of the process group ID of the
 48761 foreground process associated with the terminal. Otherwise, `-1` shall be returned and `errno` set to
 48762 indicate the error.

48763 **ERRORS**
 48764 The `tcgetpgrp()` function shall fail if:
 48765 [EBADF] The *fildev* argument is not a valid file descriptor.
 48766 [ENOTTY] The calling process does not have a controlling terminal, or the file is not the
 48767 controlling terminal.

48768 **EXAMPLES**
 48769 None.

48770 **APPLICATION USAGE**
 48771 None.

48772 **RATIONALE**
 48773 None.

48774 **FUTURE DIRECTIONS**
 48775 None.

48776 **SEE ALSO**
 48777 `setsid()`, `setpgid()`, `tcsetpgrp()`, the Base Definitions volume of IEEE Std 1003.1-200x,
 48778 `<sys/types.h>`, `<unistd.h>`

48779 **CHANGE HISTORY**
 48780 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

48781 **Issue 6**
 48782 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

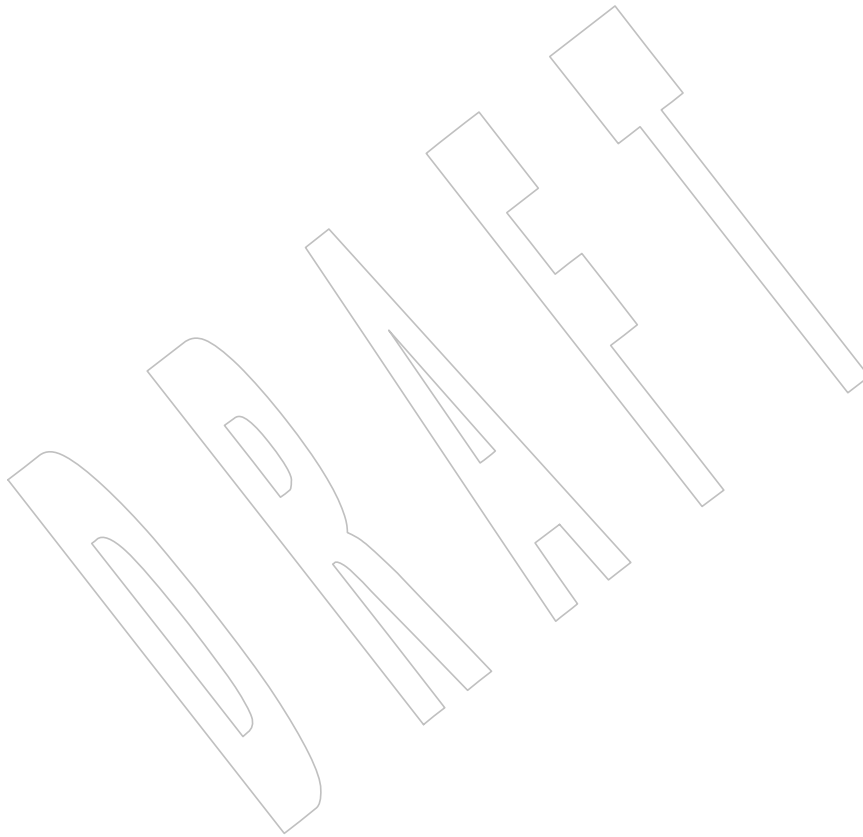
48783 The following new requirements on POSIX implementations derive from alignment with the
 48784 Single UNIX Specification:

- 48785 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
 48786 required for conforming implementations of previous POSIX specifications, it was not
 48787 required for UNIX applications.

48788

48789

- In the DESCRIPTION, text previously conditional on support for `_POSIX_JOB_CONTROL` is now mandatory. This is a FIPS requirement.



48790 **NAME**
 48791 `tcgetsid` — get the process group ID for the session leader for the controlling terminal

48792 **SYNOPSIS**
 48793 `#include <termios.h>`
 48794 `pid_t tcgetsid(int fildes);`

48795 **DESCRIPTION**
 48796 The `tcgetsid()` function shall obtain the process group ID of the session for which the terminal
 48797 specified by `fildes` is the controlling terminal.

48798 **RETURN VALUE**
 48799 Upon successful completion, `tcgetsid()` shall return the process group ID associated with the
 48800 terminal. Otherwise, a value of `(pid_t)-1` shall be returned and `errno` set to indicate the error.

48801 **ERRORS**
 48802 The `tcgetsid()` function shall fail if:

48803	[EBADF]	The <code>fildes</code> argument is not a valid file descriptor.
48804	[ENOTTY]	The calling process does not have a controlling terminal, or the file is not the 48805 controlling terminal.

48806 **EXAMPLES**
 48807 None.

48808 **APPLICATION USAGE**
 48809 None.

48810 **RATIONALE**
 48811 None.

48812 **FUTURE DIRECTIONS**
 48813 None.

48814 **SEE ALSO**
 48815 The Base Definitions volume of IEEE Std 1003.1-200x, `<termios.h>`

48816 **CHANGE HISTORY**
 48817 First released in Issue 4, Version 2.

48818 **Issue 5**
 48819 Moved from X/OPEN UNIX extension to BASE.

48820 The [EACCES] error has been removed from the list of mandatory errors, and the description of
 48821 [ENOTTY] has been reworded.

48822 **Issue 7**
 48823 The `tcgetsid()` function is moved from the XSI option to the Base.

NAME

tcsendbreak — send a break for a specific duration

SYNOPSIS

```
#include <termios.h>

int tcsendbreak(int fildev, int duration);
```

DESCRIPTION

If the terminal is using asynchronous serial data transmission, *tcsendbreak()* shall cause transmission of a continuous stream of zero-valued bits for a specific duration. If *duration* is 0, it shall cause transmission of zero-valued bits for at least 0.25 seconds, and not more than 0.5 seconds. If *duration* is not 0, it shall send zero-valued bits for an implementation-defined period of time.

The *fildev* argument is an open file descriptor associated with a terminal.

If the terminal is not using asynchronous serial data transmission, it is implementation-defined whether *tcsendbreak()* sends data to generate a break condition or returns without taking any action.

Attempts to use *tcsendbreak()* from a process which is a member of a background process group on a *fildev*

DRAFT

48864

CHANGE HISTORY

48865

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

48866

Issue 6

48867

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

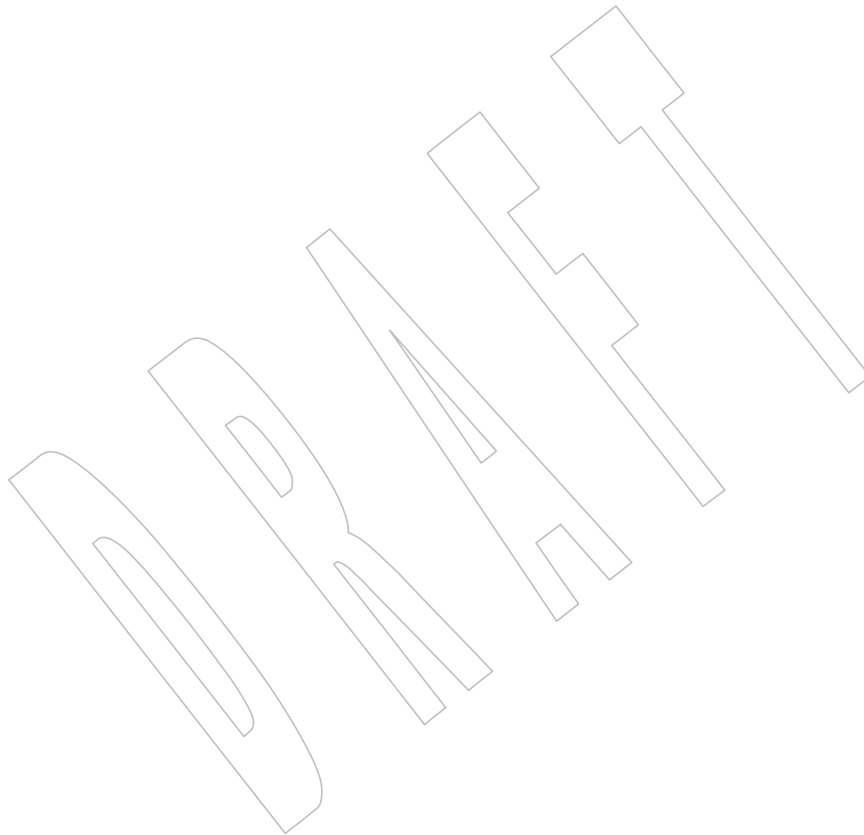
48868

- In the DESCRIPTION, text previously conditional on `_POSIX_JOB_CONTROL` is now mandated. This is a FIPS requirement.
- The [EIO] error is added.

48869

48870

48871



48872 **NAME**

48873 tcsetattr — set the parameters associated with the terminal

48874 **SYNOPSIS**

48875 #include <termios.h>

48876 int tcsetattr(int *fildev*, int *optional_actions*,
48877 const struct termios **termios_p*);48878 **DESCRIPTION**48879 The *tcsetattr()* function shall set the parameters associated with the terminal referred to by the
48880 open file descriptor *fildev* (an open file descriptor associated with a terminal) from the **termios**
48881 structure referenced by *termios_p* as follows:

- 48882
- If *optional_actions* is TCSANOW, the change shall occur immediately.
 - If *optional_actions* is TCSADRAIN, the change shall occur after all output written to *fildev* is
48883 transmitted. This function should be used when changing parameters that affect output.
 - If *optional_actions* is TCSAFLUSH, the change shall occur after all output written to *fildev* is
48884 transmitted, and all input so far received but not read shall be discarded before the change
48885 is made.

48886 If the output baud rate stored in the **termios** structure pointed to by *termios_p* is the zero baud
48887 rate, B0, the modem control lines shall no longer be asserted. Normally, this shall disconnect the
48888 line.48889 If the input baud rate stored in the **termios** structure pointed to by *termios_p* is 0, the input baud
48890 rate given to the hardware is the same as the output baud rate stored in the **termios** structure.48891 The *tcsetattr()* function shall return successfully if it was able to perform any of the requested
48892 actions, even if some of the requested actions could not be performed. It shall set all the
48893 attributes that the implementation supports as requested and leave all the attributes not
48894 supported by the implementation unchanged. If no part of the request can be honored, it shall
48895 return -1 and set *errno* to [EINVAL]. If the input and output baud rates differ and are a
48896 combination that is not supported, neither baud rate shall be changed. A subsequent call to
48897 *tcgetattr()* shall return the actual state of the terminal device (reflecting both the changes made
48898 and not made in the previous *tcsetattr()* call). The *tcsetattr()* function shall not change the values
48899 found in the **termios** structure under any circumstances.48900 The effect of *tcsetattr()* is undefined if the value of the **termios** structure pointed to by *termios_p*
48901 was not derived from the result of a call to *tcgetattr()* on *fildev*; an application should modify
48902 only fields and flags defined by this volume of IEEE Std 1003.1-200x between the call to
48903 *tcgetattr()* and *tcsetattr()*, leaving all other fields and flags unmodified.48904 No actions defined by this volume of IEEE Std 1003.1-200x, other than a call to *tcsetattr()* or a
48905 close of the last file descriptor in the system associated with this terminal device, shall cause any
48906 of the terminal attributes defined by this volume of IEEE Std 1003.1-200x to change.48907 If *tcsetattr()* is called from a process which is a member of a background process group on a *fildev*
48908 associated with its controlling terminal:

- 48909
- If the calling process is blocking or ignoring SIGTTOU signals, the operation completes
48910 normally and no signal is sent.
 - Otherwise, a SIGTTOU signal shall be sent to the process group.
- 48911
-
- 48912
-
- 48913

48914

RETURN VALUE

48915

Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to indicate the error.

48916

48917

ERRORS

48918

The *tcsetattr()* function shall fail if:

48919

[EBADF] The *fildev* argument is not a valid file descriptor.

48920

[EINTR] A signal interrupted *tcsetattr()*.

48921

[EINVAL] The *optional_actions* argument is not a supported value, or an attempt was made to change an attribute represented in the **termios** structure to an unsupported value.

48922

48923

48924

[ENOTTY] The file associated with *fildev* is not a terminal.

48925

The *tcsetattr()* function may fail if:

48926

[EIO] The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

48927

48928

EXAMPLES

48929

None.

48930

APPLICATION USAGE

48931

If trying to change baud rates, applications should call *tcsetattr()* then call *tcgetattr()* in order to determine what baud rates were actually selected.

48932

48933

RATIONALE

48934

The *tcsetattr()* function can be interrupted in the following situations:

48935

- It is interrupted while waiting for output to drain.

48936

- It is called from a process in a background process group and SIGTTOU is caught.

48937

See also the RATIONALE section in *tcgetattr()*.

48938

FUTURE DIRECTIONS

48939

Using an input baud rate of 0 to set the input rate equal to the output rate may not necessarily be supported in a future version of this volume of IEEE Std 1003.1-200x.

48940

48941

SEE ALSO

48942

cfgetispeed(), *tcgetattr()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface, **<termios.h>**, **<unistd.h>**

48943

48944

CHANGE HISTORY

48945

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

48946

Issue 6

48947

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

48948

48949

- In the DESCRIPTION, text previously conditional on **_POSIX_JOB_CONTROL** is now mandated. This is a FIPS requirement.

48950

48951

- The [EIO] error is added.

48952

48953

In the DESCRIPTION, the text describing use of *tcsetattr()* from a process which is a member of a background process group is clarified.



48954 **NAME**
 48955 tcsetpgrp — set the foreground process group ID

48956 **SYNOPSIS**
 48957 #include <unistd.h>
 48958 int tcsetpgrp(int *fildev*, pid_t *pgid_id*);

48959 **DESCRIPTION**
 48960 If the process has a controlling terminal, *tcsetpgrp()* shall set the foreground process group ID
 48961 associated with the terminal to *pgid_id*. The application shall ensure that the file associated with
 48962 *fildev* is the controlling terminal of the calling process and the controlling terminal is currently
 48963 associated with the session of the calling process. The application shall ensure that the value of
 48964 *pgid_id* matches a process group ID of a process in the same session as the calling process.

48965 Attempts to use *tcsetpgrp()* from a process which is a member of a background process group on
 48966 a *fildev* associated with its controlling terminal shall cause the process group to be sent a
 48967 SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process
 48968 shall be allowed to perform the operation, and no signal is sent.

48969 **RETURN VALUE**
 48970 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
 48971 indicate the error.

48972 **ERRORS**
 48973 The *tcsetpgrp()* function shall fail if:

48974 [EBADF]	The <i>fildev</i> argument is not a valid file descriptor.
48975 [EINVAL]	This implementation does not support the value in the <i>pgid_id</i> argument.
48976 [ENOTTY]	The calling process does not have a controlling terminal, or the file is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.
48977	
48978	
48979 [EPERM]	The value of <i>pgid_id</i> is a value supported by the implementation, but does not match the process group ID of a process in the same session as the calling process.
48980	
48981	

48982 **EXAMPLES**
 48983 None.

48984 **APPLICATION USAGE**
 48985 None.

48986 **RATIONALE**
 48987 None.

48988 **FUTURE DIRECTIONS**
 48989 None.

48990 **SEE ALSO**
 48991 *tcsetpgrp()*, the Base Definitions volume of IEEE Std 1003.1-200x, <sys/types.h>, <unistd.h>

48992 **CHANGE HISTORY**
 48993 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

48994
48995
48996
48997
48998
48999
49000
49001
49002
49003
49004

Issue 6

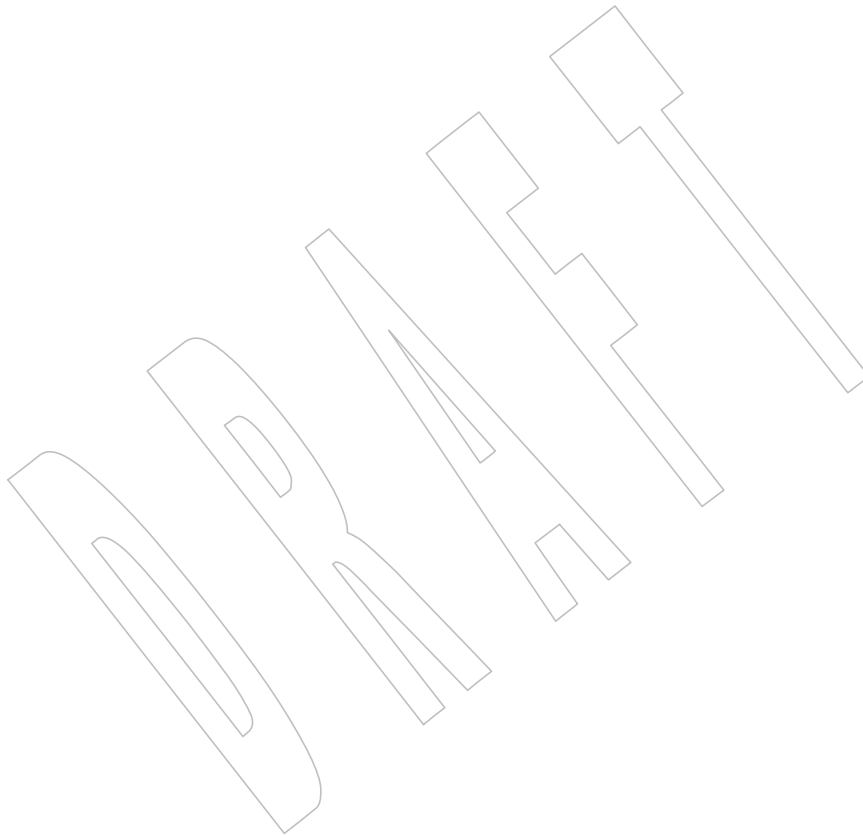
In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the DESCRIPTION and ERRORS sections, text previously conditional on `_POSIX_JOB_CONTROL` is now mandated. This is a FIPS requirement.

The normative text is updated to avoid use of the term “must” for application requirements.

The Open Group Corrigendum U047/4 is applied.



49005 **NAME**49006 `tdelete`, `tfind`, `tsearch`, `twalk` — manage a binary search tree49007 **SYNOPSIS**

```

49008 XSI      #include <search.h>
49009
49010 void *tdelete(const void *restrict key, void **restrict rootp,
49011             int(*compar)(const void *, const void *));
49012 void *tfind(const void *key, void *const *rootp,
49013            int(*compar)(const void *, const void *));
49014 void *tsearch(const void *key, void **rootp,
49015             int (*compar)(const void *, const void *));
49016 void twalk(const void *root,
49017           void (*action)(const void *, VISIT, int));

```

49017 **DESCRIPTION**

49018 The `tdelete()`, `tfind()`, `tsearch()`, and `twalk()` functions manipulate binary search trees.
 49019 Comparisons are made with a user-supplied routine, the address of which is passed as the
 49020 `compar` argument. This routine is called with two arguments, which are the pointers to the
 49021 elements being compared. The application shall ensure that the user-supplied routine returns an
 49022 integer less than, equal to, or greater than 0, according to whether the first argument is to be
 49023 considered less than, equal to, or greater than the second argument. The comparison function
 49024 need not compare every byte, so arbitrary data may be contained in the elements in addition to
 49025 the values being compared.

49026 The `tsearch()` function shall build and access the tree. The `key` argument is a pointer to an element
 49027 to be accessed or stored. If there is a node in the tree whose element is equal to the value pointed
 49028 to by `key`, a pointer to this found node shall be returned. Otherwise, the value pointed to by `key`
 49029 shall be inserted (that is, a new node is created and the value of `key` is copied to this node), and a
 49030 pointer to this node returned. Only pointers are copied, so the application shall ensure that the
 49031 calling routine stores the data. The `rootp` argument points to a variable that points to the root
 49032 node of the tree. A null pointer value for the variable pointed to by `rootp` denotes an empty tree;
 49033 in this case, the variable shall be set to point to the node which shall be at the root of the new
 49034 tree.

49035 Like `tsearch()`, `tfind()` shall search for a node in the tree, returning a pointer to it if found.
 49036 However, if it is not found, `tfind()` shall return a null pointer. The arguments for `tfind()` are the
 49037 same as for `tsearch()`.

49038 The `tdelete()` function shall delete a node from a binary search tree. The arguments are the same
 49039 as for `tsearch()`. The variable pointed to by `rootp` shall be changed if the deleted node was the
 49040 root of the tree. The `tdelete()` function shall return a pointer to the parent of the deleted node, or
 49041 a null pointer if the node is not found.

49042 The `twalk()` function shall traverse a binary search tree. The `root` argument is a pointer to the root
 49043 node of the tree to be traversed. (Any node in a tree may be used as the root for a walk below
 49044 that node.) The argument `action` is the name of a routine to be invoked at each node. This routine
 49045 is, in turn, called with three arguments. The first argument shall be the address of the node being
 49046 visited. The structure pointed to by this argument is unspecified and shall not be modified by
 49047 the application, but it shall be possible to cast a pointer-to-node into a pointer-to-pointer-to-
 49048 element to access the element stored in the node. The second argument shall be a value from an
 49049 enumeration data type:

```

49050 typedef enum { preorder, postorder, endorder, leaf } VISIT;

```

49051 (defined in `<search.h>`), depending on whether this is the first, second, or third time that the
 49052 node is visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a
 49053 leaf. The third argument shall be the level of the node in the tree, with the root being level 0.

49054 If the calling function alters the pointer to the root, the result is undefined.

49055 RETURN VALUE

49056 If the node is found, both `tsearch()` and `tfind()` shall return a pointer to it. If not, `tfind()` shall
 49057 return a null pointer, and `tsearch()` shall return a pointer to the inserted item.

49058 A null pointer shall be returned by `tsearch()` if there is not enough space available to create a new
 49059 node.

49060 A null pointer shall be returned by `tdelete()`, `tfind()`, and `tsearch()` if `rootp` is a null pointer on
 49061 entry.

49062 The `tdelete()` function shall return a pointer to the parent of the deleted node, or a null pointer if
 49063 the node is not found.

49064 The `twalk()` function shall not return a value.

49065 ERRORS

49066 No errors are defined.

49067 EXAMPLES

49068 The following code reads in strings and stores structures containing a pointer to each string and
 49069 a count of its length. It then walks the tree, printing out the stored strings and their lengths in
 49070 alphabetical order.

```

49071 #include <search.h>
49072 #include <string.h>
49073 #include <stdio.h>
49074 #define STRSZ 10000
49075 #define NODSZ 500
49076 struct node { /* Pointers to these are stored in the tree. */
49077     char *string;
49078     int length;
49079 };
49080 char string_space[STRSZ]; /* Space to store strings. */
49081 struct node nodes[NODSZ]; /* Nodes to store. */
49082 void *root = NULL; /* This points to the root. */
49083 int main(int argc, char *argv[])
49084 {
49085     char *strptr = string_space;
49086     struct node *nodeptr = nodes;
49087     void print_node(const void *, VISIT, int);
49088     int i = 0, node_compare(const void *, const void *);
49089     while (gets(strptr) != NULL && i++ < NODSZ) {
49090         /* Set node. */
49091         nodeptr->string = strptr;
49092         nodeptr->length = strlen(strptr);
49093         /* Put node into the tree. */
49094         (void) tsearch((void *)nodeptr, (void **)&root,
49095             node_compare);
49096         /* Adjust pointers, so we do not overwrite tree. */
49097         strptr += nodeptr->length + 1;

```



```

49098         nodeptr++;
49099     }
49100     twalk(root, print_node);
49101     return 0;
49102 }
49103 /*
49104  * This routine compares two nodes, based on an
49105  * alphabetical ordering of the string field.
49106  */
49107 int
49108 node_compare(const void *node1, const void *node2)
49109 {
49110     return strcmp(((const struct node *) node1)->string,
49111                 ((const struct node *) node2)->string);
49112 }
49113 /*
49114  * This routine prints out a node, the second time
49115  * twalk encounters it or if it is a leaf.
49116  */
49117 void
49118 print_node(const void *ptr, VISIT order, int level)
49119 {
49120     const struct node *p = *(const struct node **) ptr;
49121     if (order == postorder || order == leaf) {
49122         (void) printf("string = %s, length = %d\n",
49123                     p->string, p->length);
49124     }
49125 }

```

APPLICATION USAGE

The *root* argument to *twalk()* is one level of indirection less than the *rootp* arguments to *tdelete()* and *tsearch()*.

There are two nomenclatures used to refer to the order in which tree nodes are visited. The *tsearch()* function uses **preorder**, **postorder**, and **endorder** to refer respectively to visiting a node before any of its children, after its left child and before its right, and after both its children. The alternative nomenclature uses **preorder**, **inorder**, and **postorder** to refer to the same visits, which could result in some confusion over the meaning of **postorder**.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

hcreate(), *lsearch()*, the Base Definitions volume of IEEE Std 1003.1-200x, <[search.h](#)>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

49145

Issue 6

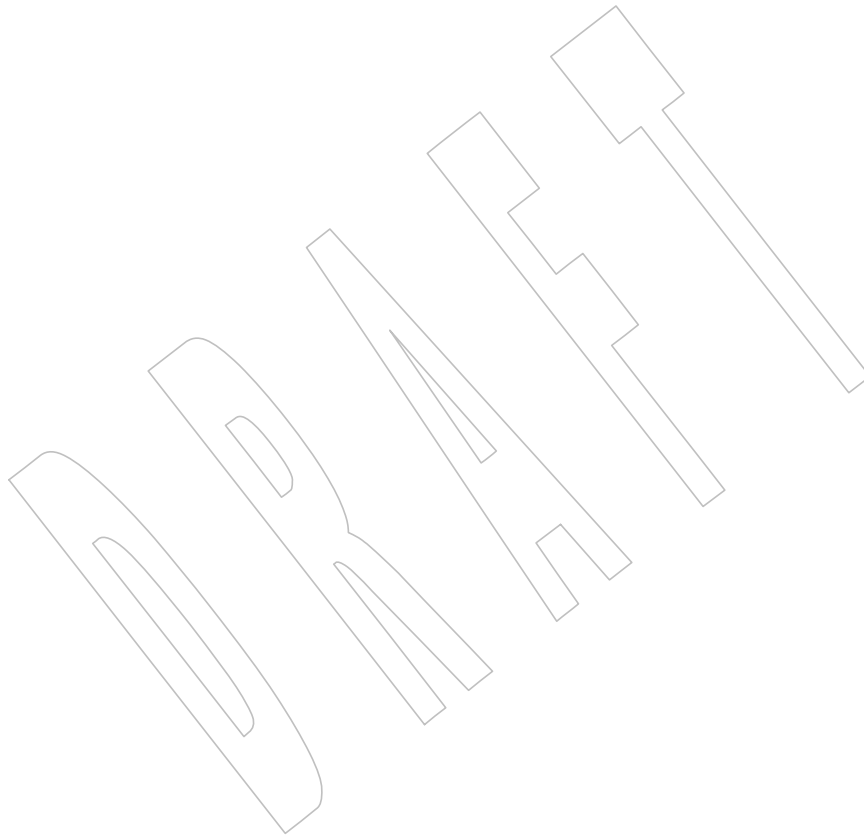
49146

The normative text is updated to avoid use of the term “must” for application requirements.

49147

The **restrict** keyword is added to the *tdelete()* prototype for alignment with the ISO/IEC 9899:1999 standard.

49148



49149 **NAME**
 49150 tellmdir — current location of a named directory stream

49151 **SYNOPSIS**

```
49152 XSI #include <dirent.h>
49153 long tellmdir(DIR *dirp);
```

49154 **DESCRIPTION**

49155 The *tellmdir()* function shall obtain the current location associated with the directory stream
 49156 specified by *dirp*.

49157 If the most recent operation on the directory stream was a *seekdir()*, the directory position
 49158 returned from the *tellmdir()* shall be the same as that supplied as a *loc* argument for *seekdir()*.

49159 **RETURN VALUE**

49160 Upon successful completion, *tellmdir()* shall return the current location of the specified directory
 49161 stream.

49162 **ERRORS**

49163 No errors are defined.

49164 **EXAMPLES**

49165 None.

49166 **APPLICATION USAGE**

49167 None.

49168 **RATIONALE**

49169 None.

49170 **FUTURE DIRECTIONS**

49171 None.

49172 **SEE ALSO**

49173 *fdopendir()*, *readdir()*, *seekdir()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**

49174 **CHANGE HISTORY**

49175 First released in Issue 2.

49176 **NAME**

49177 tempnam — create a name for a temporary file

49178 **SYNOPSIS**

49179 OB XSI #include <stdio.h>

49180 char *tempnam(const char *dir, const char *pfx);

49181 **DESCRIPTION**49182 The *tempnam()* function shall generate a pathname that may be used for a temporary file.

49183 The *tempnam()* function allows the user to control the choice of a directory. The *dir* argument
 49184 points to the name of the directory in which the file is to be created. If *dir* is a null pointer or
 49185 points to a string which is not a name for an appropriate directory, the path prefix defined as
 49186 P_tmpdir in the <stdio.h> header shall be used. If that directory is not accessible, an
 49187 implementation-defined directory may be used.

49188 Many applications prefer their temporary files to have certain initial letter sequences in their
 49189 names. The *pfx* argument should be used for this. This argument may be a null pointer or point
 49190 to a string of up to five bytes to be used as the beginning of the filename.

49191 Some implementations of *tempnam()* may use *tmpnam()* internally. On such implementations, if
 49192 called more than {TMP_MAX} times in a single process, the behavior is implementation-defined.

49193 **RETURN VALUE**

49194 Upon successful completion, *tempnam()* shall allocate space for a string, put the generated
 49195 pathname in that space, and return a pointer to it. The pointer shall be suitable for use in a
 49196 subsequent call to *free()*. Otherwise, it shall return a null pointer and set *errno* to indicate the
 49197 error.

49198 **ERRORS**49199 The *tempnam()* function shall fail if:

49200 [ENOMEM] Insufficient storage space is available.

49201 **EXAMPLES**49202 **Generating a Pathname**

49203 The following example generates a pathname for a temporary file in directory */tmp*, with the
 49204 prefix *file*. After the filename has been created, the call to *free()* deallocates the space used to
 49205 store the filename.

```
49206 #include <stdio.h>
49207 #include <stdlib.h>
49208 ...
49209 char *directory = "/tmp";
49210 char *fileprefix = "file";
49211 char *file;
49212
49213 file = tempnam(directory, fileprefix);
49214 free(file);
```

49214 **APPLICATION USAGE**

49215 This function only creates pathnames. It is the application's responsibility to create and remove
 49216 the files. Between the time a pathname is created and the file is opened, it is possible for some
 49217 other process to create a file with the same name. Applications may find *tmpfile()* more useful.

tempnam()*System Interfaces*

49218 Applications should use the *tmpfile()*, *mkdtemp()*, or *mkstemp()* functions instead of the
49219 obsolescent *tempnam()* function.

RATIONALE

49220 None.

FUTURE DIRECTIONS

49222 The *tempnam()* function may be removed in a future version.

SEE ALSO

49224 *fopen()*, *free()*, *open()*, *tmpfile()*, *tmpnam()*, *unlink()*, the Base Definitions volume of
49225 IEEE Std 1003.1-200x, **<stdio.h>**
49226

CHANGE HISTORY

49227 First released in Issue 1. Derived from Issue 1 of the SVID.
49228

Issue 5

49229 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
49230 previous issues.
49231

Issue 7

49232 The *tempnam()* function is marked obsolescent.
49233

DRAFT

49234 **NAME**
49235 tfind — search binary search tree

49236 **SYNOPSIS**

```
49237 XSI #include <search.h>  
49238 void *tfind(const void *key, void *const *rootp,  
49239 int (*compar)(const void *, const void *));
```

49240 **DESCRIPTION**

49241 Refer to *tdelete()*.

NAME

tgamma, tgammaf, tgamma1 — compute gamma() function

SYNOPSIS

```
#include <math.h>

double tgamma(double x);
float tgammaf(float x);
long double tgamma1(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the *gamma()* function of *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return *Gamma(x)*.

CX If *x* is a negative integer, a domain error may occur and either a NaN (if supported) or an implementation-defined value shall be returned. On systems that support the IEC 60559 Floating-Point option, a domain error shall occur and a NaN shall be returned.

MX If *x* is ± 0 , *tgamma()*, *tgammaf()*, and *tgamma1()* shall return \pm HUGE_VAL, \pm HUGE_VALF, and \pm HUGE_VALL, respectively. On systems that support the IEC 60559 Floating-Point option, a pole error shall occur;

CX otherwise, a pole error may occur.

If the correct value would cause overflow, a range error shall occur and *tgamma()*, *tgammaf()*, and *tgamma1()* shall return \pm HUGE_VAL, \pm HUGE_VALF, or \pm HUGE_VALL, respectively, with the same sign as the correct value of the function.

MX If *x* is NaN, a NaN shall be returned.

If *x* is +Inf, *x* shall be returned.

If *x* is -Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

ERRORS

These functions shall fail if:

MX **Domain Error** The value of *x* is a negative integer, or *x* is -Inf.
If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

MX **Pole Error** The value of *x* is zero.
If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by-zero

49285		floating-point exception shall be raised.
49286	Range Error	The value overflows.
49287		If the integer expression (<i>math_errhandling</i> & MATH_ERRNO) is non-zero,
49288		then <i>errno</i> shall be set to [ERANGE]. If the integer expression
49289		(<i>math_errhandling</i> & MATH_ERREXCEPT) is non-zero, then the overflow
49290		floating-point exception shall be raised.
49291		These functions may fail if:
49292	Domain Error	The value of <i>x</i> is a negative integer.
49293		If the integer expression (<i>math_errhandling</i> & MATH_ERRNO) is non-zero,
49294		then <i>errno</i> shall be set to [EDOM]. If the integer expression (<i>math_errhandling</i>
49295		& MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
49296		shall be raised.
49297	Pole Error	The value of <i>x</i> is zero.
49298		If the integer expression (<i>math_errhandling</i> & MATH_ERRNO) is non-zero,
49299		then <i>errno</i> shall be set to [ERANGE]. If the integer expression
49300		(<i>math_errhandling</i> & MATH_ERREXCEPT) is non-zero, then the divide-by-zero
49301		floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

For IEEE Std 754-1985 **double**, overflow happens when $0 < x < 1/\text{DBL_MAX}$, and $171.7 < x$.

On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

This function is named *tgamma()* in order to avoid conflicts with the historical *gamma()* and *lgamma()* functions.

FUTURE DIRECTIONS

It is possible that the error response for a negative integer argument may be changed to a pole error and a return value of $\pm\text{Inf}$.

SEE ALSO

feclearexcept(), *fetestexcept()*, *lgamma()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/65 is applied, correcting the third paragraph in the RETURN VALUE section.

Issue 7

ISO/IEC 9899:1999 standard, Technical Corrigendum 2 #52 (SD5-XSH-ERN-85) is applied.

49323 **NAME**
 49324 `time` — get time

49325 **SYNOPSIS**
 49326 `#include <time.h>`
 49327 `time_t time(time_t *tloc);`

49328 **DESCRIPTION**

49329 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 49330 conflict between the requirements described here and the ISO C standard is unintentional. This
 49331 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

49332 CX The `time()` function shall return the value of time in seconds since the Epoch.
 49333 The `tloc` argument points to an area where the return value is also stored. If `tloc` is a null pointer,
 49334 no value is stored.

49335 **RETURN VALUE**

49336 Upon successful completion, `time()` shall return the value of time. Otherwise, `(time_t)-1` shall be
 49337 returned.

49338 **ERRORS**

49339 No errors are defined.

49340 **EXAMPLES**

49341 **Getting the Current Time**

49342 The following example uses the `time()` function to calculate the time elapsed, in seconds, since
 49343 the Epoch, `localtime()` to convert that value to a broken-down time, and `asctime()` to convert the
 49344 broken-down time values into a printable string.

```
49345 #include <stdio.h>
49346 #include <time.h>
49347
49348 int main(void)
49349 {
49350     time_t result;
49351
49352     result = time(NULL);
49353     printf("%s%ju secs since the Epoch\n",
49354           asctime(localtime(&result)),
49355           (uintmax_t)result);
49356     return(0);
49357 }
```

49356 This example writes the current time to `stdout` in a form like this:

```
49357 Wed Jun 26 10:32:15 1996
49358 835810335 secs since the Epoch
```

49359
49360
49361
49362
49363
49364
49365
49366
49367
49368
49369
49370
49371
49372
49373
49374**Timing an Event**

The following example gets the current time, prints it out in the user's format, and prints the number of minutes to an event being timed.

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
minutes_to_event = ...;
printf("The time is ");
puts(asctime(localtime(&now)));
printf("There are %d minutes to the event.\n",
      minutes_to_event);
...
```

49375
49376**APPLICATION USAGE**

None.

49377
49378
49379
49380
49381**RATIONALE**

The *time()* function returns a value in seconds (type **time_t**) while *times()* returns a set of values in clock ticks (type **clock_t**). Some historical implementations, such as 4.3 BSD, have mechanisms capable of returning more precise times (see below). A generalized timing scheme to unify these various timing mechanisms has been proposed but not adopted.

49382
49383
49384

Implementations in which **time_t** is a 32-bit signed integer (many historical implementations) fail in the year 2038. IEEE Std 1003.1-200x does not address this problem. However, the use of the **time_t** type is mandated in order to ease the eventual fix.

49385
49386

The use of the **<time.h>** header instead of **<sys/types.h>** allows compatibility with the ISO C standard.

49387
49388
49389

Many historical implementations (including Version 7) and the 1984 /usr/group standard use **long** instead of **time_t**. This volume of IEEE Std 1003.1-200x uses the latter type in order to agree with the ISO C standard.

49390

4.3 BSD includes *time()* only as an alternate function to the more flexible *gettimeofday()* function.

49391
49392
49393
49394
49395
49396**FUTURE DIRECTIONS**

In a future version of this volume of IEEE Std 1003.1-200x, **time_t** is likely to be required to be capable of representing times far in the future. Whether this will be mandated as a 64-bit type or a requirement that a specific date in the future be representable (for example, 10000 AD) is not yet determined. Systems purchased after the approval of this volume of IEEE Std 1003.1-200x should be evaluated to determine whether their lifetime will extend past 2038.

49397
49398
49399**SEE ALSO**

asctime(), *clock()*, *ctime()*, *difftime()*, *gettimeofday()*, *gmtime()*, *localtime()*, *mktime()*, *strptime()*, *strptime()*, *utime()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

49400
49401**CHANGE HISTORY**

First released in Issue 1. Derived from Issue 1 of the SVID.

49402
49403
49404**Issue 6**

Extensions beyond the ISO C standard are marked.

The EXAMPLES, RATIONALE, and FUTURE DIRECTIONS sections are added.

49405 NAME

49406 timer_create — create a per-process timer

49407 SYNOPSIS

```

49408 CX #include <signal.h>
49409 #include <time.h>
49410 int timer_create(clockid_t clockid, struct sigevent *restrict evp,
49411 timer_t *restrict timerid);

```

49412 DESCRIPTION

49413 The *timer_create()* function shall create a per-process timer using the specified clock, *clock_id*, as
 49414 the timing base. The *timer_create()* function shall return, in the location referenced by *timerid*, a
 49415 timer ID of type **timer_t** used to identify the timer in timer requests. This timer ID shall be
 49416 unique within the calling process until the timer is deleted. The particular clock, *clock_id*, is
 49417 defined in **<time.h>**. The timer whose ID is returned shall be in a disarmed state upon return
 49418 from *timer_create()*.

49419 The *evp* argument, if non-NULL, points to a **sigevent** structure. This structure, allocated by the
 49420 application, defines the asynchronous notification to occur as specified in [Section 2.4.1](#) when the
 49421 timer expires. If the *evp* argument is NULL, the effect is as if the *evp* argument pointed to a
 49422 **sigevent** structure with the *sigev_notify* member having the value SIGEV_SIGNAL, the
 49423 *sigev_signo* having a default signal number, and the *sigev_value* member having the value of the
 49424 timer ID.

49425 Each implementation shall define a set of clocks that can be used as timing bases for per-process
 49426 timers. All implementations shall support a *clock_id* of CLOCK_REALTIME. If the Monotonic
 49427 Clock option is supported, implementations shall support a *clock_id* of CLOCK_MONOTONIC.

49428 Per-process timers shall not be inherited by a child process across a *fork()* and shall be disarmed
 49429 and deleted by an *exec*.

49430 CPT If **_POSIX_CPUTIME** is defined, implementations shall support *clock_id* values representing the
 49431 CPU-time clock of the calling process.

49432 TCT If **_POSIX_THREAD_CPUTIME** is defined, implementations shall support *clock_id* values
 49433 representing the CPU-time clock of the calling thread.

49434 CPT|TCT It is implementation-defined whether a *timer_create()* function will succeed if the value defined
 49435 by *clock_id* corresponds to the CPU-time clock of a process or thread different from the process
 49436 or thread invoking the function.

49437 TSA If *evp->sigev_notify* is SIGEV_THREAD and *sev->sigev_notify_attributes* is not NULL, if the
 49438 attribute pointed to by *sev->sigev_notify_attributes* has a thread stack address specified by a call
 49439 to *pthread_attr_setstack()*, the results are unspecified if the signal is generated more than once.

49440 RETURN VALUE

49441 If the call succeeds, *timer_create()* shall return zero and update the location referenced by *timerid*
 49442 to a **timer_t**, which can be passed to the per-process timer calls. If an error occurs, the function
 49443 shall return a value of -1 and set *errno* to indicate the error. The value of *timerid* is undefined if
 49444 an error occurs.

49445 ERRORS

49446 The *timer_create()* function shall fail if:

49447	[EAGAIN]	The system lacks sufficient signal queuing resources to honor the request.
49448	[EAGAIN]	The calling process has already created all of the timers it is allowed by this implementation.
49449		
49450	[EINVAL]	The specified clock ID is not defined.
49451	CPT TCT [ENOTSUP]	The implementation does not support the creation of a timer attached to the CPU-time clock that is specified by <i>clock_id</i> and associated with a process or thread different from the process or thread invoking <i>timer_create()</i> .
49452		
49453		

EXAMPLES

None.

APPLICATION USAGE

If a timer is created which has *evp->sigev_sigev_notify* set to `SIGEV_THREAD` and the attribute pointed to by *evp->sigev_notify_attributes* has a thread stack address specified by a call to *pthread_attr_setstack()*, the memory dedicated as a thread stack cannot be recovered. The reason for this is that the threads created in response to a timer expiration are created detached, or in an unspecified way if the thread attribute's *detachstate* is `PTHREAD_CREATE_JOINABLE`. In neither case is it valid to call *pthread_join()*, which makes it impossible to determine the lifetime of the created thread which thus means the stack memory cannot be reused.

RATIONALE**Periodic Timer Overrun and Resource Allocation**

The specified timer facilities may deliver realtime signals (that is, queued signals) on implementations that support this option. Since realtime applications cannot afford to lose notifications of asynchronous events, like timer expirations or asynchronous I/O completions, it must be possible to ensure that sufficient resources exist to deliver the signal when the event occurs. In general, this is not a difficulty because there is a one-to-one correspondence between a request and a subsequent signal generation. If the request cannot allocate the signal delivery resources, it can fail the call with an [EAGAIN] error.

Periodic timers are a special case. A single request can generate an unspecified number of signals. This is not a problem if the requesting process can service the signals as fast as they are generated, thus making the signal delivery resources available for delivery of subsequent periodic timer expiration signals. But, in general, this cannot be assured—processing of periodic timer signals may “overrun”; that is, subsequent periodic timer expirations may occur before the currently pending signal has been delivered.

Also, for signals, according to the POSIX.1-1990 standard, if subsequent occurrences of a pending signal are generated, it is implementation-defined whether a signal is delivered for each occurrence. This is not adequate for some realtime applications. So a mechanism is required to allow applications to detect how many timer expirations were delayed without requiring an indefinite amount of system resources to store the delayed expirations.

The specified facilities provide for an overrun count. The overrun count is defined as the number of extra timer expirations that occurred between the time a timer expiration signal is generated and the time the signal is delivered. The signal-catching function, if it is concerned with overruns, can retrieve this count on entry. With this method, a periodic timer only needs one “signal queuing resource” that can be allocated at the time of the *timer_create()* function call.

A function is defined to retrieve the overrun count so that an application need not allocate static storage to contain the count, and an implementation need not update this storage asynchronously on timer expirations. But, for some high-frequency periodic applications, the overhead of an additional system call on each timer expiration may be prohibitive. The functions, as defined, permit an implementation to maintain the overrun count in user space, associated with the *timerid*. The *timer_getoverrun()* function can then be implemented as a macro

49495 that uses the *timerid* argument (which may just be a pointer to a user space structure containing
 49496 the counter) to locate the overrun count with no system call overhead. Other implementations,
 49497 less concerned with this class of applications, can avoid the asynchronous update of user space
 49498 by maintaining the count in a system structure at the cost of the extra system call to obtain it.

49499 **Timer Expiration Signal Parameters**

49500 The Realtime Signals Extension option supports an application-specific datum that is delivered
 49501 to the extended signal handler. This value is explicitly specified by the application, along with
 49502 the signal number to be delivered, in a **sigevent** structure. The type of the application-defined
 49503 value can be either an integer constant or a pointer. This explicit specification of the value, as
 49504 opposed to always sending the timer ID, was selected based on existing practice.

49505 It is common practice for realtime applications (on non-POSIX systems or realtime extended
 49506 POSIX systems) to use the parameters of event handlers as the case label of a switch statement or
 49507 as a pointer to an application-defined data structure. Since *timer_ids* are dynamically allocated
 49508 by the *timer_create()* function, they can be used for neither of these functions without additional
 49509 application overhead in the signal handler; for example, to search an array of saved timer IDs to
 49510 associate the ID with a constant or application data structure.

49511 **FUTURE DIRECTIONS**

49512 None.

49513 **SEE ALSO**

49514 [clock_getres\(\)](#), [timer_delete\(\)](#), [timer_getoverrun\(\)](#), the Base Definitions volume of
 49515 IEEE Std 1003.1-200x, **<time.h>**

49516 **CHANGE HISTORY**

49517 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

49518 **Issue 6**

49519 The *timer_create()* function is marked as part of the Timers option.

49520 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 49521 implementation does not support the Timers option.

49522 CPU-time clocks are added for alignment with IEEE Std 1003.1d-1999.

49523 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding the
 49524 requirement for the CLOCK_MONOTONIC clock under the Monotonic Clock option.

49525 The **restrict** keyword is added to the *timer_create()* prototype for alignment with the
 49526 ISO/IEC 9899:1999 standard.

49527 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/138 is applied, updating the
 49528 DESCRIPTION and APPLICATION USAGE sections to describe the case when a timer is created
 49529 with the notification method set to SIGEV_THREAD.

49530 **Issue 7**

49531 The *timer_create()* function is moved from the Timers option to the Base.

49532 **NAME**

49533 timer_delete — delete a per-process timer

49534 **SYNOPSIS**

```
49535 CX #include <time.h>
49536 int timer_delete(timer_t timerid);
```

49537 **DESCRIPTION**

49538 The *timer_delete()* function deletes the specified timer, *timerid*, previously created by the
 49539 *timer_create()* function. If the timer is armed when *timer_delete()* is called, the behavior shall be
 49540 as if the timer is automatically disarmed before removal. The disposition of pending signals for
 49541 the deleted timer is unspecified.

49542 **RETURN VALUE**

49543 If successful, the *timer_delete()* function shall return a value of zero. Otherwise, the function shall
 49544 return a value of -1 and set *errno* to indicate the error.

49545 **ERRORS**49546 The *timer_delete()* function may fail if:49547 [EINVAL] The timer ID specified by *timerid* is not a valid timer ID.49548 **EXAMPLES**

49549 None.

49550 **APPLICATION USAGE**

49551 None.

49552 **RATIONALE**

49553 None.

49554 **FUTURE DIRECTIONS**

49555 None.

49556 **SEE ALSO**49557 *timer_create()*, the Base Definitions volume of IEEE Std 1003.1-200x, <time.h>49558 **CHANGE HISTORY**

49559 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

49560 **Issue 6**49561 The *timer_delete()* function is marked as part of the Timers option.

49562 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 49563 implementation does not support the Timers option.

49564 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/139 is applied, updating the ERRORS
 49565 section so that the [EINVAL] error becomes optional.

49566 **Issue 7**49567 The *timer_delete()* function is moved from the Timers option to the Base.

49568 **NAME**

49569 timer_getoverrun, timer_gettime, timer_settime — per-process timers

49570 **SYNOPSIS**

```

49571 CX    #include <time.h>
49572
49572     int timer_getoverrun(timer_t timerid);
49573     int timer_gettime(timer_t timerid, struct itimerspec *value);
49574     int timer_settime(timer_t timerid, int flags,
49575         const struct itimerspec *restrict value,
49576         struct itimerspec *restrict ovalue);

```

49577 **DESCRIPTION**

49578 The *timer_gettime()* function shall store the amount of time until the specified timer, *timerid*,
 49579 expires and the reload value of the timer into the space pointed to by the *value* argument. The
 49580 *it_value* member of this structure shall contain the amount of time before the timer expires, or
 49581 zero if the timer is disarmed. This value is returned as the interval until timer expiration, even if
 49582 the timer was armed with absolute time. The *it_interval* member of *value* shall contain the reload
 49583 value last set by *timer_settime()*.

49584 The *timer_settime()* function shall set the time until the next expiration of the timer specified by
 49585 *timerid* from the *it_value* member of the *value* argument and arm the timer if the *it_value* member
 49586 of *value* is non-zero. If the specified timer was already armed when *timer_settime()* is called, this
 49587 call shall reset the time until next expiration to the *value* specified. If the *it_value* member of *value*
 49588 is zero, the timer shall be disarmed. The effect of disarming or resetting a timer with pending
 49589 expiration notifications is unspecified.

49590 If the flag `TIMER_ABSTIME` is not set in the argument *flags*, *timer_settime()* shall behave as if the
 49591 time until next expiration is set to be equal to the interval specified by the *it_value* member of
 49592 *value*. That is, the timer shall expire in *it_value* nanoseconds from when the call is made. If the
 49593 flag `TIMER_ABSTIME` is set in the argument *flags*, *timer_settime()* shall behave as if the time
 49594 until next expiration is set to be equal to the difference between the absolute time specified by
 49595 the *it_value* member of *value* and the current value of the clock associated with *timerid*. That is,
 49596 the timer shall expire when the clock reaches the value specified by the *it_value* member of *value*.
 49597 If the specified time has already passed, the function shall succeed and the expiration
 49598 notification shall be made.

49599 The reload value of the timer shall be set to the value specified by the *it_interval* member of
 49600 *value*. When a timer is armed with a non-zero *it_interval*, a periodic (or repetitive) timer is
 49601 specified.

49602 Time values that are between two consecutive non-negative integer multiples of the resolution of
 49603 the specified timer shall be rounded up to the larger multiple of the resolution. Quantization
 49604 error shall not cause the timer to expire earlier than the rounded time value.

49605 If the argument *ovalue* is not `NULL`, the *timer_settime()* function shall store, in the location
 49606 referenced by *ovalue*, a value representing the previous amount of time before the timer would
 49607 have expired, or zero if the timer was disarmed, together with the previous timer reload value.
 49608 Timers shall not expire before their scheduled time.

49609 Only a single signal shall be queued to the process for a given timer at any point in time. When a
 49610 timer for which a signal is still pending expires, no signal shall be queued, and a timer overrun
 49611 shall occur. When a timer expiration signal is delivered to or accepted by a process, the
 49612 *timer_getoverrun()* function shall return the timer expiration overrun count for the specified
 49613 timer. The overrun count returned contains the number of extra timer expirations that occurred

49614 between the time the signal was generated (queued) and when it was delivered or accepted, up
 49615 to but not including an implementation-defined maximum of {DELAYTIMER_MAX}. If the
 49616 number of such extra expirations is greater than or equal to {DELAYTIMER_MAX}, then the
 49617 overrun count shall be set to {DELAYTIMER_MAX}. The value returned by *timer_getoverrun()*
 49618 shall apply to the most recent expiration signal delivery or acceptance for the timer. If no
 49619 expiration signal has been delivered for the timer, the return value of *timer_getoverrun()* is
 49620 unspecified.

RETURN VALUE

49621 If the *timer_getoverrun()* function succeeds, it shall return the timer expiration overrun count as
 49622 explained above.
 49623

49624 If the *timer_gettime()* or *timer_settime()* functions succeed, a value of 0 shall be returned.

49625 If an error occurs for any of these functions, the value -1 shall be returned, and *errno* set to
 49626 indicate the error.

ERRORS

49627 The *timer_settime()* function shall fail if:

49628 [EINVAL] A *value* structure specified a nanosecond value less than zero or greater than
 49629 or equal to 1000 million, and the *it_value* member of that structure did not
 49630 specify zero seconds and nanoseconds.
 49631

49632 These functions may fail if:

49633 [EINVAL] The *timerid* argument does not correspond to an ID returned by *timer_create()*
 49634 but not yet deleted by *timer_delete()*.

49635 The *timer_settime()* function may fail if:

49636 [EINVAL] The *it_interval* member of *value* is not zero and the timer was created with
 49637 notification by creation of a new thread (*sigev_sigev_notify* was
 49638 SIGEV_THREAD) and a fixed stack address has been set in the thread
 49639 attribute pointed to by *sigev_notify_attributes*.

EXAMPLES

49640 None.
 49641

APPLICATION USAGE

49642 Using fixed stack addresses is problematic when timer expiration is signalled by the creation of a
 49643 new thread. Since it cannot be assumed that the thread created for one expiration is finished
 49644 before the next expiration of the timer, it could happen that two threads use the same memory as
 49645 a stack at the same time. This is invalid and produces undefined results.
 49646

RATIONALE

49647 Practical clocks tick at a finite rate, with rates of 100 hertz and 1 000 hertz being common. The
 49648 inverse of this tick rate is the clock resolution, also called the clock granularity, which in either
 49649 case is expressed as a time duration, being 10 milliseconds and 1 millisecond respectively for
 49650 these common rates. The granularity of practical clocks implies that if one reads a given clock
 49651 twice in rapid succession, one may get the same time value twice; and that timers must wait for
 49652 the next clock tick after the theoretical expiration time, to ensure that a timer never returns too
 49653 soon. Note also that the granularity of the clock may be significantly coarser than the resolution
 49654 of the data format used to set and get time and interval values. Also note that some
 49655 implementations may choose to adjust time and/or interval values to exactly match the ticks of
 49656 the underlying clock.
 49657

49658 This volume of IEEE Std 1003.1-200x defines functions that allow an application to determine the
 49659 implementation-supported resolution for the clocks and requires an implementation to
 49660 document the resolution supported for timers and *nanosleep()* if they differ from the supported
 49661 clock resolution. This is more of a procurement issue than a runtime application issue.

49662

FUTURE DIRECTIONS

49663

None.

49664

SEE ALSO

49665

clock_getres(), *timer_create()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**time.h**>

49666

CHANGE HISTORY

49667

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

49668

Issue 6

49669

The *timer_getoverrun()*, *timer_gettime()*, and *timer_settime()* functions are marked as part of the Timers option.

49670

49671

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

49672

49673

The [EINVAL] error condition is updated to include the following: “and the *it_value* member of that structure did not specify zero seconds and nanoseconds.” This change is for IEEE PASC Interpretation 1003.1 #89.

49674

49675

49676

The DESCRIPTION for *timer_getoverrun()* is updated to clarify that “If no expiration signal has been delivered for the timer, or if the Realtime Signals Extension is not supported, the return value of *timer_getoverrun()* is unspecified”.

49677

49678

49679

The **restrict** keyword is added to the *timer_settime()* prototype for alignment with the ISO/IEC 9899:1999 standard.

49680

49681

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/140 is applied, updating the ERRORS section so that the mandatory [EINVAL] error (“The *timerid* argument does not correspond to an ID returned by *timer_create()* but not yet deleted by *timer_delete()*”) becomes optional.

49682

49683

49684

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/141 is applied, updating the ERRORS section to include an optional [EINVAL] error for the case when a timer is created with the notification method set to SIGEV_THREAD. APPLICATION USAGE text is also added.

49685

49686

49687

Issue 7

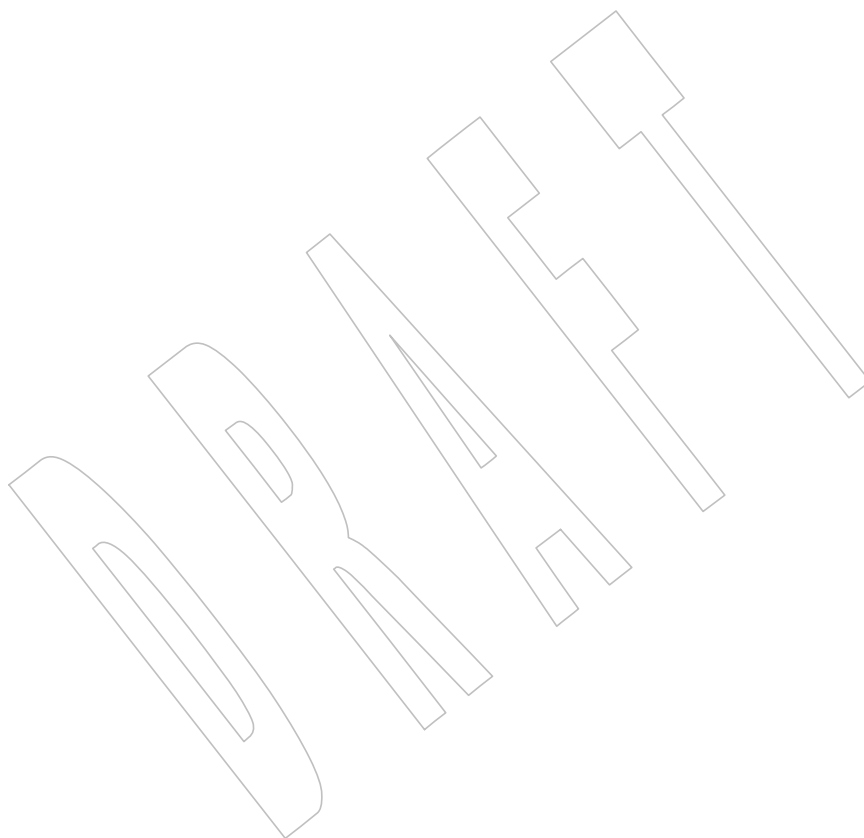
49688

The *timer_getoverrun()*, *timer_gettime()*, and *timer_settime()* functions are moved from the Timers option to the Base.

49689

49690

Functionality relating to the Realtime Signals Extension option is moved to the Base.



```

49735     ...
49736     void
49737     start_clock()
49738     {
49739         st_time = times(&st_cpu);
49740     }
49741
49742     /* This example assumes that the result of each subtraction
49743        is within the range of values that can be represented in
49744        an integer type. */
49745     void
49746     end_clock(char *msg)
49747     {
49748         en_time = times(&en_cpu);
49749
49750         fputs(msg, stdout);
49751         printf("Real Time: %jd, User Time %jd, System Time %jd\n",
49752             (intmax_t)(en_time - st_time),
49753             (intmax_t)(en_cpu.tms_utime - st_cpu.tms_utime),
49754             (intmax_t)(en_cpu.tms_stime - st_cpu.tms_stime));
49755     }

```

APPLICATION USAGE

Applications should use `sysconf(SC_CLK_TCK)` to determine the number of clock ticks per second as it may vary from system to system.

RATIONALE

The accuracy of the times reported is intentionally left unspecified to allow implementations flexibility in design, from uniprocessor to multi-processor networks.

The inclusion of times of child processes is recursive, so that a parent process may collect the total times of all of its descendants. But the times of a child are only added to those of its parent when its parent successfully waits on the child. Thus, it is not guaranteed that a parent process can always see the total times of all its descendants; see also the discussion of the term “realtime” in [alarm\(\)](#).

If the type `clock_t` is defined to be a signed 32-bit integer, it overflows in somewhat more than a year if there are 60 clock ticks per second, or less than a year if there are 100. There are individual systems that run continuously for longer than that. This volume of IEEE Std 1003.1-200x permits an implementation to make the reference point for the returned value be the start-up time of the process, rather than system start-up time.

The term “charge” in this context has nothing to do with billing for services. The operating system accounts for time used in this way. That information must be correct, regardless of how that information is used.

FUTURE DIRECTIONS

None.

SEE ALSO

[alarm\(\)](#), [exec](#), [fork\(\)](#), [sysconf\(\)](#), [time\(\)](#), [wait\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, [<sys/times.h>](#)

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

49780 **NAME**
49781 `timezone` — difference from UTC and local standard time

49782 **SYNOPSIS**

49783 XSI `#include <time.h>`
49784 `extern long timezone;`

49785 **DESCRIPTION**

49786 Refer to [tzset\(\)](#).

49787 **NAME**

49788 tmpfile — create a temporary file

49789 **SYNOPSIS**

49790 #include <stdio.h>

49791 FILE *tmpfile(void);

49792 **DESCRIPTION**

49793 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 49794 conflict between the requirements described here and the ISO C standard is unintentional. This
 49795 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

49796 The *tmpfile()* function shall create a temporary file and open a corresponding stream. The file
 49797 shall be automatically deleted when all references to the file are closed. The file is opened as in
 49798 *fopen()* for update (*w+*), except that implementations may restrict the permissions, either by
 49799 clearing the file mode bits or setting them to the value `S_IRUSR | S_IWUSR`.

49800 CX In some implementations, a permanent file may be left behind if the process calling *tmpfile()* is
 49801 killed while it is processing a call to *tmpfile()*.

49802 An error message may be written to standard error if the stream cannot be opened.

49803 **RETURN VALUE**

49804 Upon successful completion, *tmpfile()* shall return a pointer to the stream of the file that is
 49805 CX created. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

49806 **ERRORS**49807 The *tmpfile()* function shall fail if:

49808 CX **[EINTR]** A signal was caught during *tmpfile()*.

49809 CX **[EMFILE]** All file descriptors available to the process are currently open.

49810 CX **[ENFILE]** The maximum allowable number of files is currently open in the system.

49811 CX **[ENOSPC]** The directory or file system which would contain the new file cannot be
 49812 expanded.

49813 CX **[EOVERFLOW]** The file is a regular file and the size of the file cannot be represented correctly
 49814 in an object of type **off_t**.

49815 The *tmpfile()* function may fail if:

49816 CX **[EMFILE]** {`FOPEN_MAX`} streams are currently open in the calling process.

49817 CX **[ENOMEM]** Insufficient storage space is available.

49818 **EXAMPLES**49819 **Creating a Temporary File**

49820 The following example creates a temporary file for update, and returns a pointer to a stream for
 49821 the created file in the *fp* variable.

```
49822 #include <stdio.h>
49823 ...
49824 FILE *fp;
49825 fp = tmpfile ();
```

49826
49827
49828
49829
49830

49831
49832

49833
49834

49835
49836
49837

49838
49839

49840
49841

49842
49843

49844
49845

49846
49847

49848
49849

49850

49851
49852

49853
49854

49855
49856

APPLICATION USAGE

It should be possible to open at least {TMP_MAX} temporary files during the lifetime of the program (this limit may be shared with *tmpnam()*) and there should be no limit on the number simultaneously open other than this limit and any limit on the number of open files ({FOPEN_MAX}).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fopen(), *mkdtemp()*, *tmpnam()*, *unlink()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdio.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Large File Summit extensions are added.

The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes in previous issues.

Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the ERRORS section, the [Eoverflow] condition is added. This change is to support large files.
- The [EMFILE] optional error condition is added.

The APPLICATION USAGE section is added for alignment with the ISO/IEC 9899:1999 standard.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

Austin Group Interpretation 1003.1-2001 #025 is applied, clarifying that implementations may restrict the permissions of the file created.

49857 **NAME**

49858 tmpnam — create a name for a temporary file

49859 **SYNOPSIS**

```
49860 OB #include <stdio.h>
49861 char *tmpnam(char *s);
```

49862 **DESCRIPTION**

49863 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 49864 conflict between the requirements described here and the ISO C standard is unintentional. This
 49865 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

49866 The *tmpnam()* function shall generate a string that is a valid filename and that is not the same as
 49867 the name of an existing file. The function is potentially capable of generating {TMP_MAX}
 49868 different strings, but any or all of them may already be in use by existing files and thus not be
 49869 suitable return values.

49870 The *tmpnam()* function generates a different string each time it is called from the same process,
 49871 up to {TMP_MAX} times. If it is called more than {TMP_MAX} times, the behavior is
 49872 implementation-defined.

49873 The implementation shall behave as if no function defined in this volume of
 49874 IEEE Std 1003.1-200x, except *tmpnam()*, calls *tmpnam()*.

49875 CX If the application uses any of the POSIX threads functions, the application shall ensure that the
 49876 *tmpnam()* function is called with a non-NULL parameter.

49877 **RETURN VALUE**

49878 Upon successful completion, *tmpnam()* shall return a pointer to a string. If no suitable string can
 49879 be generated, the *tmpnam()* function shall return a null pointer.

49880 If the argument *s* is a null pointer, *tmpnam()* shall leave its result in an internal static object and
 49881 return a pointer to that object. Subsequent calls to *tmpnam()* may modify the same object. If the
 49882 argument *s* is not a null pointer, it is presumed to point to an array of at least `L_tmpnam` **chars**;
 49883 *tmpnam()* shall write its result in that array and shall return the argument as its value.

49884 **ERRORS**

49885 No errors are defined.

49886 **EXAMPLES**49887 **Generating a Filename**49888 The following example generates a unique filename and stores it in the array pointed to by *ptr*.

```
49889 #include <stdio.h>
49890 ...
49891 char filename[L_tmpnam+1];
49892 char *ptr;
49893 ptr = tmpnam(filename);
```

49894 **APPLICATION USAGE**49895 This function only creates filenames. It is the application's responsibility to create and remove
49896 the files.49897 Between the time a pathname is created and the file is opened, it is possible for some other
49898 process to create a file with the same name. Applications may find *tmpfile()* more useful.

49899 Applications should use the *tmpfile()*, *mkstemp()*, or *mkdtemp()* functions instead of the
 49900 obsolescent *tmpnam()* function.

49901 **RATIONALE**

49902 None.

49903 **FUTURE DIRECTIONS**

49904 The *tmpnam()* function may be removed in a future version.

49905 **SEE ALSO**

49906 *fopen()*, *open()*, *mkdtemp()*, *tmpnam()*, *tmpfile()*, *unlink()*, the Base Definitions volume of
 49907 IEEE Std 1003.1-200x, <stdio.h>

49908 **CHANGE HISTORY**

49909 First released in Issue 1. Derived from Issue 1 of the SVID.

49910 **Issue 5**

49911 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

49912 **Issue 6**

49913 Extensions beyond the ISO C standard are marked.

49914 The normative text is updated to avoid use of the term “must” for application requirements.

49915 The DESCRIPTION is expanded for alignment with the ISO/IEC 9899:1999 standard.

49916 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/142 is applied, updating the
 49917 DESCRIPTION to allow implementations of the *tmpnam()* function to call *tmpnam()*.

49918 **Issue 7**

49919 The *tmpnam()* function is marked obsolescent.

49920 **NAME**
49921 toascii — translate an integer to a 7-bit ASCII character

49922 **SYNOPSIS**
49923 OB XSI `#include <ctype.h>`
49924 `int toascii(int c);`

49925 **DESCRIPTION**
49926 The *toascii()* function shall convert its argument into a 7-bit ASCII character.

49927 **RETURN VALUE**
49928 The *toascii()* function shall return the value (*c* & 0x7f).

49929 **ERRORS**
49930 No errors are returned.

49931 **EXAMPLES**
49932 None.

49933 **APPLICATION USAGE**
49934 The *toascii()* function cannot be used portably in a localized application.

49935 **RATIONALE**
49936 None.

49937 **FUTURE DIRECTIONS**
49938 The *toascii()* function may be removed in a future version.

49939 **SEE ALSO**
49940 *isascii()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<ctype.h>`

49941 **CHANGE HISTORY**
49942 First released in Issue 1. Derived from Issue 1 of the SVID.

49943 **Issue 7**
49944 The *toascii()* function is marked obsolescent.

49945 **NAME**
 49946 tolower, tolower_l — transliterate uppercase characters to lowercase

49947 **SYNOPSIS**

49948 #include <ctype.h>
 49949 int tolower(int c);
 49950 CX int tolower_l(int c, locale_t locale);

49951 **DESCRIPTION**

49952 CX For *tolower()*: The functionality described on this reference page is aligned with the ISO C
 49953 standard. Any conflict between the requirements described here and the ISO C standard is
 49954 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

49955 CX The *tolower()* and *tolower_l()* functions have as a domain a type **int**, the value of which is
 49956 representable as an **unsigned char** or the value of EOF. If the argument has any other value, the
 49957 behavior is undefined. If the argument of *tolower()* or *tolower_l()* represents an uppercase letter,
 49958 and there exists a corresponding lowercase letter as defined by character type information in the
 49959 program locale or in the locale represented by *locale*, respectively (category *LC_CTYPE*), the
 49960 result shall be the corresponding lowercase letter. All other arguments in the domain are
 49961 returned unchanged.

49962 **RETURN VALUE**

49963 CX Upon successful completion, the *tolower()* and *tolower_l()* functions shall return the lowercase
 49964 letter corresponding to the argument passed; otherwise, they shall return the argument
 49965 unchanged.

49966 **ERRORS**

49967 The *tolower_l()* function may fail if:

49968 CX [EINVAL] *locale* is not a valid locale object handle.

49969 **EXAMPLES**

49970 None.

49971 **APPLICATION USAGE**

49972 None.

49973 **RATIONALE**

49974 None.

49975 **FUTURE DIRECTIONS**

49976 None.

49977 **SEE ALSO**

49978 *setlocale()*, *uselocale()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale,
 49979 <ctype.h>, <locale.h>

49980 **CHANGE HISTORY**

49981 First released in Issue 1. Derived from Issue 1 of the SVID.

49982 **Issue 6**

49983 Extensions beyond the ISO C standard are marked.

tolower()

49984

Issue 7

49985

The *tolower_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

49986



49987 **NAME**

49988 toupper, toupper_l — transliterate lowercase characters to uppercase

49989 **SYNOPSIS**

49990 #include <ctype.h>

49991 int toupper(int c);

49992 CX int toupper_l(int c, locale_t locale);

49993 **DESCRIPTION**49994 CX For *toupper()*: The functionality described on this reference page is aligned with the ISO C
49995 standard. Any conflict between the requirements described here and the ISO C standard is
49996 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.49997 CX The *toupper()* and *toupper_l()* functions have as a domain a type **int**, the value of which is
49998 representable as an **unsigned char** or the value of EOF. If the argument has any other value, the
49999 behavior is undefined. If the argument of *toupper()* or *toupper_l()* represents a lowercase letter,
50000 and there exists a corresponding uppercase letter as defined by character type information in the
50001 program locale or in the locale represented by *locale*, respectively (category *LC_CTYPE*), the
50002 result shall be the corresponding uppercase letter. All other arguments in the domain are
50003 returned unchanged.50004 **RETURN VALUE**50005 CX Upon successful completion, *toupper()* and *toupper_l()* shall return the uppercase letter
50006 corresponding to the argument passed.50007 **ERRORS**50008 The *toupper_l()* function may fail if:50009 CX [EINVAL] *locale* is not a valid locale object handle.50010 **EXAMPLES**

50011 None.

50012 **APPLICATION USAGE**

50013 None.

50014 **RATIONALE**

50015 None.

50016 **FUTURE DIRECTIONS**

50017 None.

50018 **SEE ALSO**50019 *setlocale()*, *uselocale()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale,
50020 <ctype.h>, <locale.h>50021 **CHANGE HISTORY**

50022 First released in Issue 1. Derived from Issue 1 of the SVID.

50023 **Issue 6**

50024 Extensions beyond the ISO C standard are marked.

50025 **Issue 7**50026 The *toupper_l()* function is added from The Open Group Technical Standard, 2006, Extended API
50027 Set Part 4.

50028 **NAME**
 50029 towctrans, towctrans_l — wide-character transliteration

50030 **SYNOPSIS**

```
50031 #include <wctype.h>
50032
50032 wint_t towctrans(wint_t wc, wctrans_t desc);
50033 CX wint_t towctrans_l(wint_t wc, wctrans_t desc,
50034 locale_t locale);
```

50035 **DESCRIPTION**

50036 CX For *towctrans()*: The functionality described on this reference page is aligned with the ISO C
 50037 standard. Any conflict between the requirements described here and the ISO C standard is
 50038 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

50039 CX The *towctrans()* and *towctrans_l()* functions shall transliterate the wide-character code *wc* using
 50040 the mapping described by *desc*.

50041 CX The current setting of the *LC_CTYPE* category in the current locale of the process or in the locale
 50042 CX represented by *locale*, respectively, should be the same as during the call to *wctrans()* or
 50043 *wctrans_l()* that returned the value *desc*.

50044 If the value of *desc* is invalid (that is, not obtained by a call to *wctrans()* or *desc* is invalidated by a
 50045 subsequent call to *setlocale()* that has affected category *LC_CTYPE*), the result is unspecified.

50046 CX If the value of *desc* is invalid (that is, not obtained by a call to *wctrans_l()* with the same locale
 50047 object *locale*) the result is unspecified.

50048 CX An application wishing to check for error situations should set *errno* to 0 before calling
 50049 *towctrans()* or *towctrans_l()*.

50050 If *errno* is non-zero on return, an error has occurred.

50051 **RETURN VALUE**

50052 CX If successful, the *towctrans()* and *towctrans_l()* functions shall return the mapped value of *wc*
 50053 using the mapping described by *desc*. Otherwise, they shall return *wc* unchanged.

50054 **ERRORS**

50055 These functions may fail if:

50056 CX [EINVAL] *desc* contains an invalid transliteration descriptor.

50057 The *towctrans_l()* function may fail if:

50058 CX [EINVAL] *locale* is not a valid locale object handle.

50059 **EXAMPLES**

50060 None.

50061 **APPLICATION USAGE**

50062 The strings "tolower" and "toupper" are reserved for the standard mapping names. In the
 50063 table below, the functions in the left column are equivalent to the functions in the right column.

```
50064 tolower(wc)          towctrans(wc, wctrans("tolower"))
50065 tolower_l(wc, locale) towctrans_l(wc, wctrans("tolower"), locale)
50066 toupper(wc)         towctrans(wc, wctrans("toupper"))
50067 toupper_l(wc, locale) towctrans_l(wc, wctrans("toupper"), locale)
```

50068
50069
50070
50071
50072
50073
50074
50075
50076
50077
50078
50079
50080
50081

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

tolower(), *toupper()*, *wctrans()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<wctype.h>`

CHANGE HISTORY

First released in Issue 5. Derived from ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

Extensions beyond the ISO C standard are marked.

Issue 7

The *towctrans_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



50082 **NAME**

50083 towlower, towlower_l — transliterate uppercase wide-character code to lowercase

50084 **SYNOPSIS**

50085 #include <wctype.h>

50086 wint_t towlower(wint_t wc);

50087 CX wint_t towlower_l(wint_t wc, locale_t locale);

50088 **DESCRIPTION**50089 CX For *towlower()*: The functionality described on this reference page is aligned with the ISO C
50090 standard. Any conflict between the requirements described here and the ISO C standard is
50091 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.50092 CX The *towlower()* and *towlower_l()* functions have as a domain a type **wint_t**, the value of which
50093 the application shall ensure is a character representable as a **wchar_t**, and a wide-character code
50094 corresponding to a valid character in the current locale or the value of WEOF. If the argument
50095 CX has any other value, the behavior is undefined. If the argument of *towlower()* or *towlower_l()*
50096 represents an uppercase wide-character code, and there exists a corresponding lowercase wide-
50097 CX character code as defined by character type information in the locale of the process or in the
50098 locale represented by *locale*, respectively (category *LC_CTYPE*), the result shall be the
50099 corresponding lowercase wide-character code. All other arguments in the domain are returned
50100 unchanged.50101 **RETURN VALUE**50102 CX Upon successful completion, the *towlower()* and *towlower_l()* functions shall return the
50103 lowercase letter corresponding to the argument passed; otherwise, they shall return the
50104 argument unchanged.50105 **ERRORS**50106 The *towlower_l()* function may fail if:50107 CX [EINVAL] *locale* is not a valid locale object handle.50108 **EXAMPLES**

50109 None.

50110 **APPLICATION USAGE**

50111 None.

50112 **RATIONALE**

50113 None.

50114 **FUTURE DIRECTIONS**

50115 None.

50116 **SEE ALSO**50117 *setlocale()*, *uselocale()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale,
50118 <locale.h>, <wctype.h>50119 **CHANGE HISTORY**

50120 First released in Issue 4.

50121 **Issue 5**50122 The following change has been made in this issue for alignment with
50123 ISO/IEC 9899:1990/Amendment 1:1995 (E):

50124

50125

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the `<wctype.h>` header rather than `<wchar.h>`.

50126

Issue 6

50127

The normative text is updated to avoid use of the term “must” for application requirements.

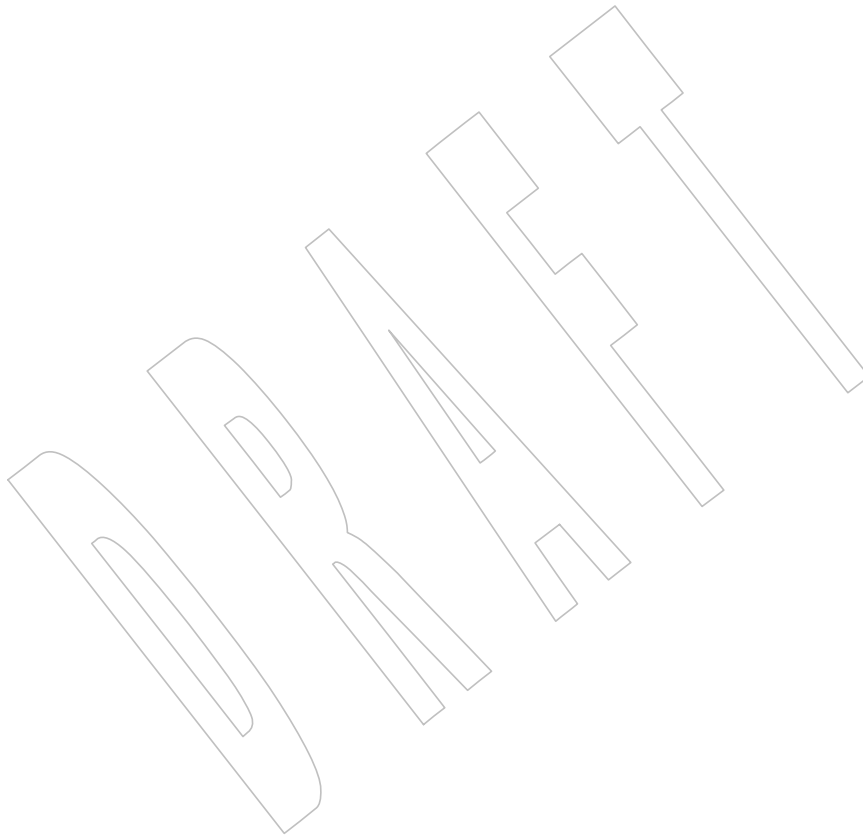
50128

Issue 7

50129

50130

The `tolower_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



50131 **NAME**

50132 towupper, towupper_l — transliterate lowercase wide-character code to uppercase

50133 **SYNOPSIS**

50134 #include <wctype.h>

50135 wint_t towupper(wint_t wc);

50136 CX wint_t towupper_l(wint_t wc, locale_t locale);

50137 **DESCRIPTION**50138 CX For *towupper()*: The functionality described on this reference page is aligned with the ISO C
50139 standard. Any conflict between the requirements described here and the ISO C standard is
50140 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.50141 CX The *towupper()* and *towupper_l()* functions have as a domain a type **wint_t**, the value of which
50142 the application shall ensure is a character representable as a **wchar_t**, and a wide-character code
50143 corresponding to a valid character in the current locale or the value of WEOF. If the argument
50144 has any other value, the behavior is undefined. If the argument of *towupper()* or *towupper_l()*
50145 represents a lowercase wide-character code, and there exists a corresponding uppercase wide-
50146 character code as defined by character type information in the locale of the process or in the
50147 locale represented by *locale*, respectively (category *LC_CTYPE*), the result shall be the
50148 corresponding uppercase wide-character code. All other arguments in the domain are returned
50149 unchanged.50150 **RETURN VALUE**50151 CX Upon successful completion, the *towupper()* and *towupper_l()* functions shall return the
50152 uppercase letter corresponding to the argument passed. Otherwise, they shall return the
50153 argument unchanged.50154 **ERRORS**50155 The *towupper_l()* function may fail if:50156 CX [EINVAL] *locale* is not a valid locale object handle.50157 **EXAMPLES**

50158 None.

50159 **APPLICATION USAGE**

50160 None.

50161 **RATIONALE**

50162 None.

50163 **FUTURE DIRECTIONS**

50164 None.

50165 **SEE ALSO**50166 *setlocale()*, *uselocale()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale,
50167 <locale.h>, <wctype.h>50168 **CHANGE HISTORY**

50169 First released in Issue 4.

50170 **Issue 5**50171 The following change has been made in this issue for alignment with
50172 ISO/IEC 9899:1990/Amendment 1:1995 (E):

50173

50174

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the `<wctype.h>` header rather than `<wchar.h>`.

50175

Issue 6

50176

The normative text is updated to avoid use of the term “must” for application requirements.

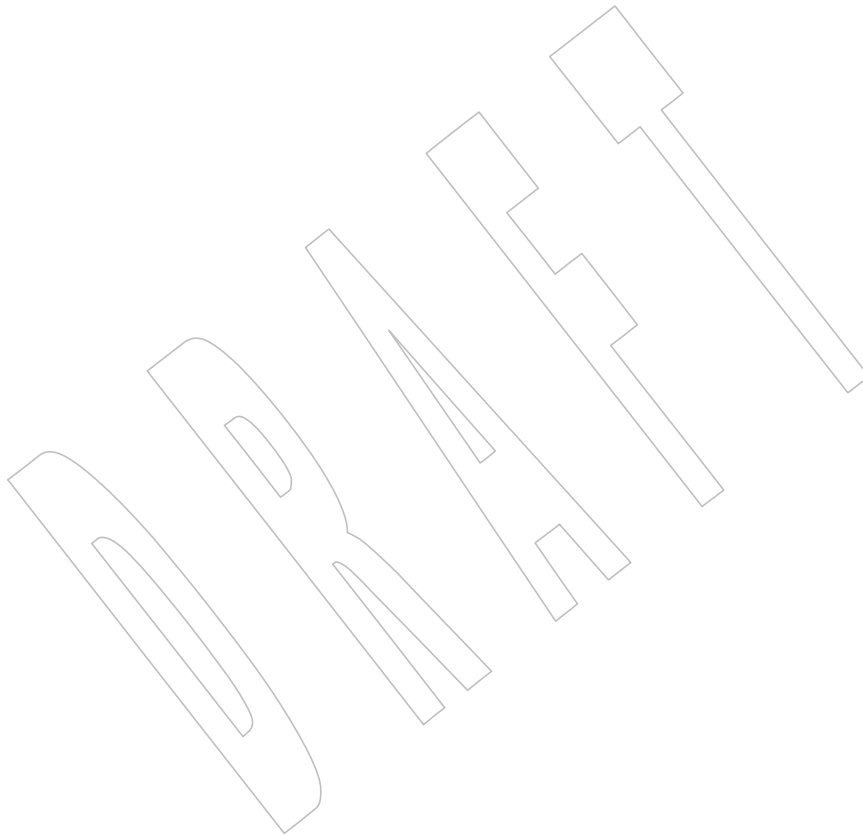
50177

Issue 7

50178

50179

The `towupper_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



50180 **NAME**
 50181 trunc, truncf, trunc1 — round to truncated integer value

50182 **SYNOPSIS**
 50183 #include <math.h>
 50184 double trunc(double x);
 50185 float truncf(float x);
 50186 long double trunc1(long double x);

50187 **DESCRIPTION**
 50188 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 50189 conflict between the requirements described here and the ISO C standard is unintentional. This
 50190 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

50191 These functions shall round their argument to the integer value, in floating format, nearest to but
 50192 no larger in magnitude than the argument.

50193 **RETURN VALUE**
 50194 Upon successful completion, these functions shall return the truncated integer value.

50195 MX If x is NaN, a NaN shall be returned.
 50196 If x is ± 0 or $\pm \text{Inf}$, x shall be returned.

50197 **ERRORS**
 50198 No errors are defined.

50199 **EXAMPLES**
 50200 None.

50201 **APPLICATION USAGE**
 50202 None.

50203 **RATIONALE**
 50204 None.

50205 **FUTURE DIRECTIONS**
 50206 None.

50207 **SEE ALSO**
 50208 The Base Definitions volume of IEEE Std 1003.1-200x, <math.h>

50209 **CHANGE HISTORY**
 50210 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

50211 **NAME**

50212 truncate — truncate a file to a specified length

50213 **SYNOPSIS**

50214 #include <unistd.h>

50215 int truncate(const char *path, off_t length);

50216 **DESCRIPTION**50217 The *truncate()* function shall cause the regular file named by *path* to have a size which shall be
50218 equal to *length* bytes.50219 If the file previously was larger than *length*, the extra data is discarded. If the file was previously
50220 shorter than *length*, its size is increased, and the extended area appears as if it were zero-filled.

50221 The application shall ensure that the process has write permission for the file.

50222 If the request would cause the file size to exceed the soft file size limit for the process, the
50223 request shall fail and the implementation shall generate the SIGXFSZ signal for the process.50224 This function shall not modify the file offset for any open file descriptions associated with the
50225 file. Upon successful completion, if the file size is changed, this function shall mark for update
50226 the *st_ctime* and *st_mtime* fields of the file, and the S_ISUID and S_ISGID bits of the file mode
50227 may be cleared.50228 **RETURN VALUE**50229 Upon successful completion, *truncate()* shall return 0. Otherwise, -1 shall be returned, and *errno*
50230 set to indicate the error.50231 **ERRORS**50232 The *truncate()* function shall fail if:

50233 [EINTR] A signal was caught during execution.

50234 [EINVAL] The *length* argument was less than 0.

50235 [EFBIG] or [EINVAL]

50236 The *length* argument was greater than the maximum file size.

50237 [EIO] An I/O error occurred while reading from or writing to a file system.

50238 [EACCES] A component of the path prefix denies search permission, or write permission
50239 is denied on the file.

50240 [EISDIR] The named file is a directory.

50241 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
50242 argument.

50243 [ENAMETOOLONG]

50244 The length of the *path* argument exceeds {PATH_MAX} or a pathname
50245 component is longer than {NAME_MAX}.50246 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.50247 [ENOTDIR] A component of the path prefix of *path* is not a directory.

50248 [EROFS] The named file resides on a read-only file system.

truncate()

50249 The *truncate()* function may fail if:

50250 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
50251 resolution of the *path* argument.

50252 [ENAMETOOLONG]
50253 Pathname resolution of a symbolic link produced an intermediate result
50254 whose length exceeds {PATH_MAX}.

EXAMPLES

50255 None.
50256

APPLICATION USAGE

50257 None.
50258

RATIONALE

50259 None.
50260

FUTURE DIRECTIONS

50261 None.
50262

SEE ALSO

50263 *open()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**unistd.h**>
50264

CHANGE HISTORY

50265 First released in Issue 4, Version 2.
50266

Issue 5

50267 Moved from X/OPEN UNIX extension to BASE.
50268

50269 Large File Summit extensions are added.

Issue 6

50270 This reference page is split out from the *ftruncate()* reference page.
50271

50272 The normative text is updated to avoid use of the term “must” for application requirements.

50273 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
50274 [ELOOP] error condition is added.

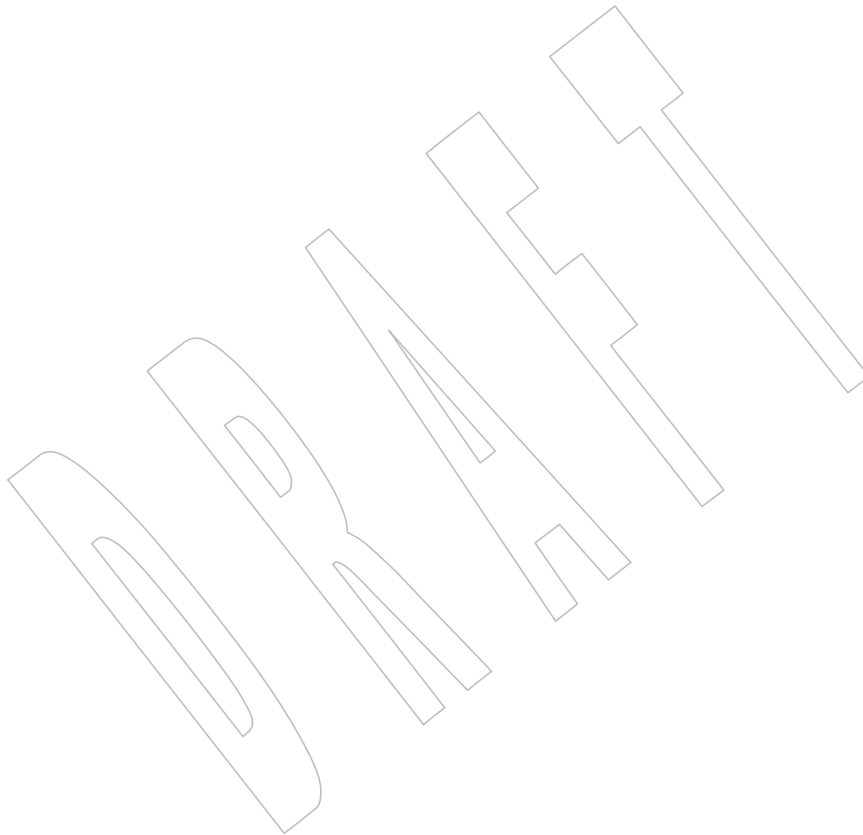
Issue 7

50275 The *truncate()* function is moved from the XSI option to the Base.
50276

50277 **NAME**
50278 `truncf, trunc1` — round to truncated integer value

50279 **SYNOPSIS**
50280 `#include <math.h>`
50281 `float truncf(float x);`
50282 `long double trunc1(long double x);`

50283 **DESCRIPTION**
50284 Refer to *trunc()*.



tsearch()50285 **NAME**50286 `tsearch` — search a binary search tree50287 **SYNOPSIS**

```
50288 XSI #include <search.h>
50289 void *tsearch(const void *key, void **rootp,
50290             int (*compar)(const void *, const void *));
```

50291 **DESCRIPTION**50292 Refer to *tdelete()*.

50293 **NAME**
 50294 `ttyname, ttyname_r` — find the pathname of a terminal

50295 **SYNOPSIS**
 50296 `#include <unistd.h>`
 50297 `char *ttyname(int fd);`
 50298 `int ttyname_r(int fd, char *name, size_t namesize);`

50299 **DESCRIPTION**
 50300 The `ttyname()` function shall return a pointer to a string containing a null-terminated pathname
 50301 of the terminal associated with file descriptor *fd*. The return value may point to static data
 50302 whose content is overwritten by each call.

50303 The `ttyname()` function need not be thread-safe. A function that is not required to be thread-safe
 50304 is not required to be reentrant.

50305 The `ttyname_r()` function shall store the null-terminated pathname of the terminal associated
 50306 with the file descriptor *fd* in the character array referenced by *name*. The array is *namesize*
 50307 characters long and should have space for the name and the terminating null character. The
 50308 maximum length of the terminal name shall be {TTY_NAME_MAX}.

50309 **RETURN VALUE**
 50310 Upon successful completion, `ttyname()` shall return a pointer to a string. Otherwise, a null
 50311 pointer shall be returned and *errno* set to indicate the error.

50312 If successful, the `ttyname_r()` function shall return zero. Otherwise, an error number shall be
 50313 returned to indicate the error.

50314 **ERRORS**
 50315 The `ttyname()` function may fail if:
 50316 [EBADF] The *fd* argument is not a valid file descriptor.
 50317 [ENOTTY] The file associated with the *fd* argument is not a terminal.
 50318 The `ttyname_r()` function may fail if:
 50319 [EBADF] The *fd* argument is not a valid file descriptor.
 50320 [ENOTTY] The file associated with the *fd* argument is not a terminal.
 50321 [ERANGE] The value of *namesize* is smaller than the length of the string to be returned
 50322 including the terminating null character.

50323 **EXAMPLES**
 50324 None.

50325 **APPLICATION USAGE**
 50326 None.

50327 **RATIONALE**
 50328 The term “terminal” is used instead of the historical term “terminal device” in order to avoid a
 50329 reference to an undefined term.

50330 The thread-safe version places the terminal name in a user-supplied buffer and returns a non-
 50331 zero value if it fails. The non-thread-safe version may return the name in a static data area that
 50332 may be overwritten by each call.

50333

FUTURE DIRECTIONS

50334

None.

50335

SEE ALSO

50336

The Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

50337

CHANGE HISTORY

50338

First released in Issue 1. Derived from Issue 1 of the SVID.

50339

Issue 5

50340

The *ttyname_r()* function is included for alignment with the POSIX Threads Extension.

50341

A note indicating that the *ttyname()* function need not be reentrant is added to the DESCRIPTION.

50342

50343

Issue 6

50344

The *ttyname_r()* function is marked as part of the Thread-Safe Functions option.

50345

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

50346

50347

- The statement that *errno* is set on error is added.

50348

- The [EBADF] and [ENOTTY] optional error conditions are added.

50349

Issue 7

50350

SD5-XSH-ERN-100 is applied, correcting the definition of the [ENOTTY] error condition.

50351

The *ttyname_r()* function is moved from the Thread-Safe Functions option to the Base.

50352 **NAME**
50353 twalk — traverse a binary search tree

50354 **SYNOPSIS**
50355 XSI

```
#include <search.h>
```


50356

```
void twalk(const void *root,
```


50357

```
           void (*action)(const void *, VISIT, int ));
```

50358 **DESCRIPTION**
50359 Refer to *tdelete()*.

50360 **NAME**
 50361 daylight, timezone, tzname, tzset — set timezone conversion information

50362 **SYNOPSIS**
 50363 #include <time.h>

50364 XSI extern int daylight;
 50365 extern long timezone;
 50366 CX extern char *tzname[2];
 50367 void tzset(void);

50368 **DESCRIPTION**
 50369 The *tzset()* function shall use the value of the environment variable *TZ* to set time conversion
 50370 information used by *ctime()*, *localtime()*, *mktime()*, and *strftime()*. If *TZ* is absent from the
 50371 environment, implementation-defined default timezone information shall be used.

50372 The *tzset()* function shall set the external variable *tzname* as follows:

50373 tzname[0] = "std";
 50374 tzname[1] = "dst";

50375 where *std* and *dst* are as described in the Base Definitions volume of IEEE Std 1003.1-200x,
 50376 Chapter 8, Environment Variables.

50377 XSI The *tzset()* function also shall set the external variable *daylight* to 0 if Daylight Savings Time
 50378 conversions should never be applied for the timezone in use; otherwise, non-zero. The external
 50379 variable *timezone* shall be set to the difference, in seconds, between Coordinated Universal Time
 50380 (UTC) and local standard time.

50381 **RETURN VALUE**
 50382 The *tzset()* function shall not return a value.

50383 **ERRORS**
 50384 No errors are defined.

50385 **EXAMPLES**
 50386 Example *TZ* variables and their timezone differences are given in the table below:

<i>TZ</i>	<i>timezone</i>
EST5EDT	5*60*60
GMT0	0*60*60
JST-9	-9*60*60
MET-1MEST	-1*60*60
MST7MDT	7*60*60
PST8PDT	8*60*60

50394 **APPLICATION USAGE**
 50395 None.

50396 **RATIONALE**
 50397 None.

50398 **FUTURE DIRECTIONS**
 50399 None.

50400
50401
50402
50403
50404
50405
50406**SEE ALSO**

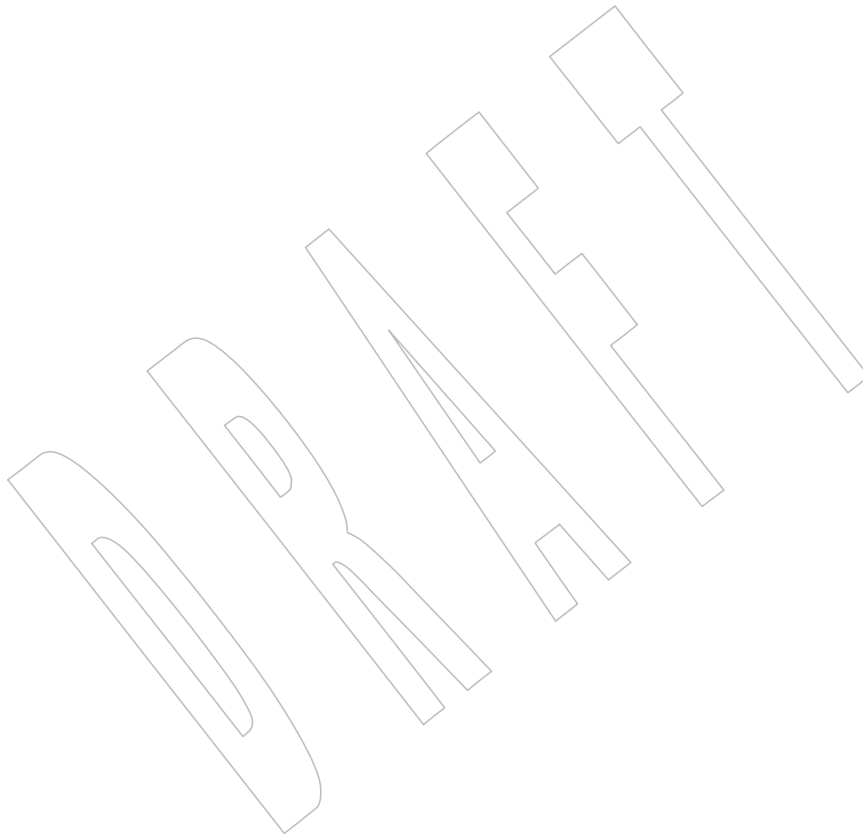
ctime(), *localtime()*, *mktime()*, *strftime()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables, **<time.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The example is corrected.



50407 **NAME**50408 `ulimit` — get and set process limits50409 **SYNOPSIS**

```
50410 OB XSI #include <ulimit.h>
50411 long ulimit(int cmd, ...);
```

50412 **DESCRIPTION**

50413 The `ulimit()` function shall control process limits. The process limits that can be controlled by
 50414 this function include the maximum size of a single file that can be written (this is equivalent to
 50415 using `setrlimit()` with `RLIMIT_FSIZE`). The `cmd` values, defined in `<ulimit.h>`, include:

50416 `UL_GETFSIZE` Return the file size limit (`RLIMIT_FSIZE`) of the process. The limit shall be in
 50417 units of 512-byte blocks and shall be inherited by child processes. Files of any
 50418 size can be read. The return value shall be the integer part of the soft file size
 50419 limit divided by 512. If the result cannot be represented as a **long**, the result is
 50420 unspecified.

50421 `UL_SETFSIZE` Set the file size limit for output operations of the process to the value of the
 50422 second argument, taken as a **long**, multiplied by 512. If the result would
 50423 overflow an `rlim_t`, the actual value set is unspecified. Any process may
 50424 decrease its own limit, but only a process with appropriate privileges may
 50425 increase the limit. The return value shall be the integer part of the new file size
 50426 limit divided by 512.

50427 The `ulimit()` function shall not change the setting of `errno` if successful.

50428 As all return values are permissible in a successful situation, an application wishing to check for
 50429 error situations should set `errno` to 0, then call `ulimit()`, and, if it returns `-1`, check to see if `errno` is
 50430 non-zero.

50431 **RETURN VALUE**

50432 Upon successful completion, `ulimit()` shall return the value of the requested limit. Otherwise, `-1`
 50433 shall be returned and `errno` set to indicate the error.

50434 **ERRORS**

50435 The `ulimit()` function shall fail and the limit shall be unchanged if:

50436 `[EINVAL]` The `cmd` argument is not valid.

50437 `[EPERM]` A process not having appropriate privileges attempts to increase its file size
 50438 limit.

50439 **EXAMPLES**

50440 None.

50441 **APPLICATION USAGE**

50442 Since the `ulimit()` function uses type **long** rather than `rlim_t`, this function is not sufficient for file
 50443 sizes on many current systems. Applications should use the `getrlimit()` or `setrlimit()` functions
 50444 instead of the obsolescent `ulimit()` function.

50445 **RATIONALE**

50446 None.

50447

FUTURE DIRECTIONS

50448

The *ulimit()* function may be removed in a future version.

50449

SEE ALSO

50450

exec, *getrlimit()*, *setrlimit()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<ulimit.h>**

50451

50452

CHANGE HISTORY

50453

First released in Issue 1. Derived from Issue 1 of the SVID.

50454

Issue 5

50455

In the description of `UL_SETFSIZE`, the text is corrected to refer to `rlim_t` rather than the spurious `rlimit_t`.

50456

50457

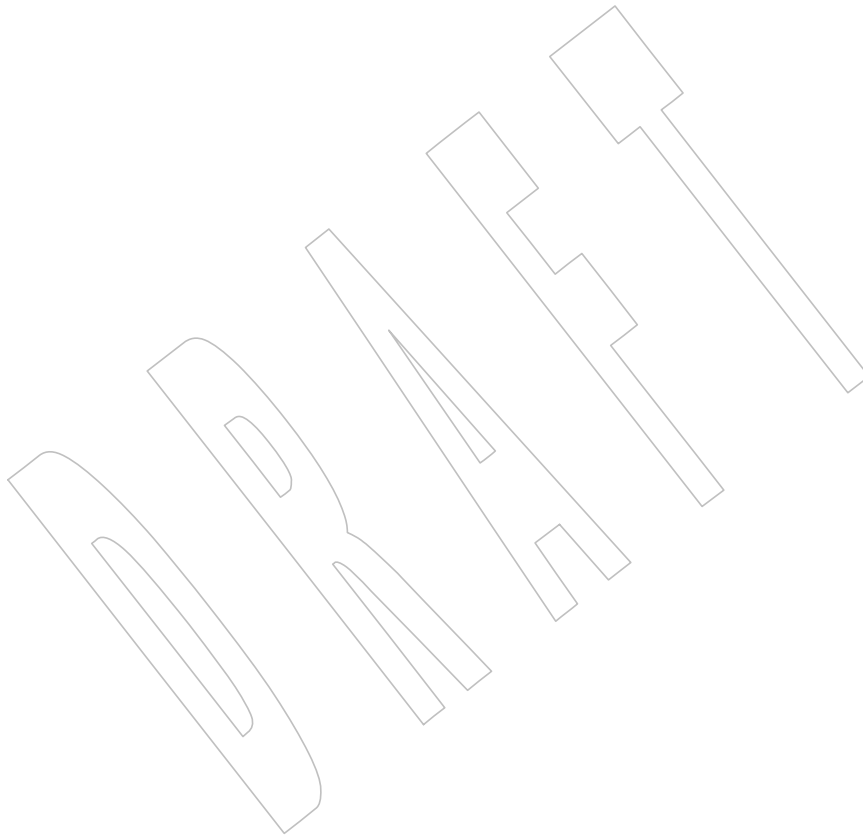
The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

50458

Issue 7

50459

The *ulimit()* function is marked obsolescent.



50460 **NAME**
 50461 `umask` — set and get the file mode creation mask

50462 **SYNOPSIS**
 50463 `#include <sys/stat.h>`
 50464 `mode_t umask(mode_t cmask);`

50465 **DESCRIPTION**
 50466 The `umask()` function shall set the file mode creation mask of the process to `cmask` and return the
 50467 previous value of the mask. Only the file permission bits of `cmask` (see `<sys/stat.h>`) are used; the
 50468 meaning of the other bits is implementation-defined.

50469 The file mode creation mask of the process is used to turn off permission bits in the `mode`
 50470 argument supplied during calls to the following functions:

- 50471 • `open()`, `openat()`, `creat()`, `mkdir()`, `mkdirat()`, `mkfifo()`, and `mkfifoat()`
- 50472 XSI • `mknod()`, `mknodat()`
- 50473 MSG • `mq_open()`
- 50474 • `sem_open()`

50475 Bit positions that are set in `cmask` are cleared in the mode of the created file.

50476 **RETURN VALUE**
 50477 The file permission bits in the value returned by `umask()` shall be the previous value of the file
 50478 mode creation mask. The state of any other bits in that value is unspecified, except that a
 50479 subsequent call to `umask()` with the returned value as `cmask` shall leave the state of the mask the
 50480 same as its state before the first call, including any unspecified use of those bits.

50481 **ERRORS**
 50482 No errors are defined.

50483 **EXAMPLES**
 50484 None.

50485 **APPLICATION USAGE**
 50486 None.

50487 **RATIONALE**
 50488 Unsigned argument and return types for `umask()` were proposed. The return type and the
 50489 argument were both changed to **mode_t**.

50490 Historical implementations have made use of additional bits in `cmask` for their implementation-
 50491 defined purposes. The addition of the text that the meaning of other bits of the field is
 50492 implementation-defined permits these implementations to conform to this volume of
 50493 IEEE Std 1003.1-200x.

50494 **FUTURE DIRECTIONS**
 50495 None.

50496 **SEE ALSO**
 50497 `creat()`, `exec`, `mkdir()`, `mkfifo()`, `mknod()`, `mq_open()`, `open()`, `sem_open()`, the Base Definitions
 50498 volume of IEEE Std 1003.1-200x, `<sys/stat.h>`, `<sys/types.h>`

50499

CHANGE HISTORY

50500

First released in Issue 1. Derived from Issue 1 of the SVID.

50501

Issue 6

50502

In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

50503

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

50504

50505

- The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

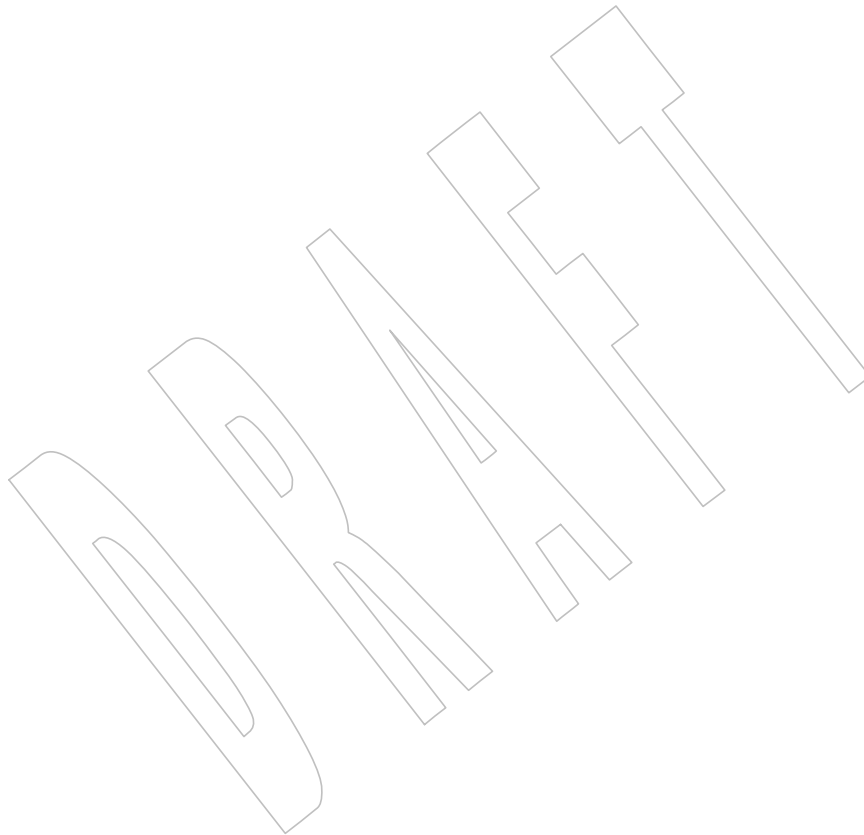
50506

50507

50508

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/143 is applied, adding the `mknod()`, `mq_open()`, and `sem_open()` functions to the DESCRIPTION and SEE ALSO sections.

50509



50510 **NAME**50511 **uname** — get the name of the current system50512 **SYNOPSIS**

50513 #include <sys/utsname.h>

50514 int uname(struct utsname *name);

50515 **DESCRIPTION**50516 The *uname()* function shall store information identifying the current system in the structure pointed to by *name*.50517
50518 The *uname()* function uses the **utsname** structure defined in <sys/utsname.h>.50519 The *uname()* function shall return a string naming the current system in the character array *sysname*. Similarly, *nodename* shall contain the name of this node within an implementation-defined communications network. The arrays *release* and *version* shall further identify the operating system. The array *machine* shall contain a name that identifies the hardware that the system is running on.50520
50521
50522
50523
50524 The format of each member is implementation-defined.50525 **RETURN VALUE**50526 Upon successful completion, a non-negative value shall be returned. Otherwise, -1 shall be returned and *errno* set to indicate the error.50527
50528 **ERRORS**

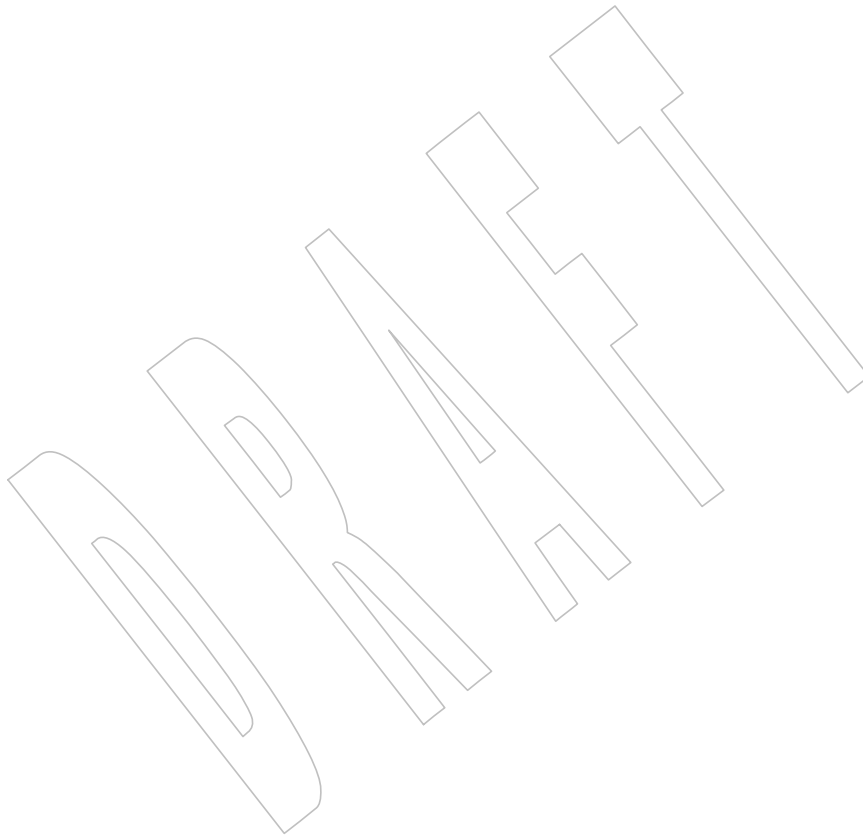
50529 No errors are defined.

50530 **EXAMPLES**

50531 None.

50532 **APPLICATION USAGE**50533 The inclusion of the *nodename* member in this structure does not imply that it is sufficient information for interfacing to communications networks.50534
50535 **RATIONALE**50536 The values of the structure members are not constrained to have any relation to the version of this volume of IEEE Std 1003.1-200x implemented in the operating system. An application should instead depend on `_POSIX_VERSION` and related constants defined in <unistd.h>.50537
50538
50539 This volume of IEEE Std 1003.1-200x does not define the sizes of the members of the structure and permits them to be of different sizes, although most implementations define them all to be the same size: eight bytes plus one byte for the string terminator. That size for *nodename* is not enough for use with many networks.50540
50541
50542
50543 The *uname()* function originated in System III, System V, and related implementations, and it does not exist in Version 7 or 4.3 BSD. The values it returns are set at system compile time in those historical implementations.50544
50545
50546 4.3 BSD has *gethostname()* and *gethostid()*, which return a symbolic name and a numeric value, respectively. There are related *sethostname()* and *sethostid()* functions that are used to set the values the other two functions return. The former functions are included in this specification, the latter are not.

- 50550 **FUTURE DIRECTIONS**
50551 None.
- 50552 **SEE ALSO**
50553 The Base Definitions volume of IEEE Std 1003.1-200x, **<sys/utsname.h>**
- 50554 **CHANGE HISTORY**
50555 First released in Issue 1. Derived from Issue 1 of the SVID.



50556 **NAME**

50557 ungetc — push byte back into input stream

50558 **SYNOPSIS**

50559 #include <stdio.h>

50560 int ungetc(int *c*, FILE **stream*);50561 **DESCRIPTION**

50562 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 50563 conflict between the requirements described here and the ISO C standard is unintentional. This
 50564 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

50565 The *ungetc()* function shall push the byte specified by *c* (converted to an **unsigned char**) back
 50566 onto the input stream pointed to by *stream*. The pushed-back bytes shall be returned by
 50567 subsequent reads on that stream in the reverse order of their pushing. A successful intervening
 50568 call (with the stream pointed to by *stream*) to a file-positioning function (*fseek()*, *fsetpos()*, or
 50569 *rewind()*) shall discard any pushed-back bytes for the stream. The external storage
 50570 corresponding to the stream shall be unchanged.

50571 One byte of push-back shall be provided. If *ungetc()* is called too many times on the same stream
 50572 without an intervening read or file-positioning operation on that stream, the operation may fail.

50573 If the value of *c* equals that of the macro EOF, the operation shall fail and the input stream shall
 50574 be left unchanged.

50575 A successful call to *ungetc()* shall clear the end-of-file indicator for the stream. The value of the
 50576 file-position indicator for the stream after reading or discarding all pushed-back bytes shall be
 50577 the same as it was before the bytes were pushed back. The file-position indicator is decremented
 50578 by each successful call to *ungetc()*; if its value was 0 before a call, its value is unspecified after
 50579 the call.

50580 **RETURN VALUE**

50581 Upon successful completion, *ungetc()* shall return the byte pushed back after conversion.
 50582 Otherwise, it shall return EOF.

50583 **ERRORS**

50584 No errors are defined.

50585 **EXAMPLES**

50586 None.

50587 **APPLICATION USAGE**

50588 None.

50589 **RATIONALE**

50590 None.

50591 **FUTURE DIRECTIONS**

50592 None.

50593 **SEE ALSO**

50594 *fseek()*, *getc()*, *fsetpos()*, *read()*, *rewind()*, *setbuf()*, the Base Definitions volume of
 50595 IEEE Std 1003.1-200x, <stdio.h>

50596
50597

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.



50598 **NAME**
 50599 `ungetwc` — push wide-character code back into the input stream

50600 **SYNOPSIS**
 50601 `#include <stdio.h>`
 50602 `#include <wchar.h>`
 50603 `wint_t ungetwc(wint_t wc, FILE *stream);`

50604 **DESCRIPTION**
 50605 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 50606 conflict between the requirements described here and the ISO C standard is unintentional. This
 50607 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

50608 The `ungetwc()` function shall push the character corresponding to the wide-character code
 50609 specified by `wc` back onto the input stream pointed to by `stream`. The pushed-back characters
 50610 shall be returned by subsequent reads on that stream in the reverse order of their pushing. A
 50611 successful intervening call (with the stream pointed to by `stream`) to a file-positioning function
 50612 (`fseek()`, `fsetpos()`, or `rewind()`) discards any pushed-back characters for the stream. The external
 50613 storage corresponding to the stream is unchanged.

50614 At least one character of push-back shall be provided. If `ungetwc()` is called too many times on
 50615 the same stream without an intervening read or file-positioning operation on that stream, the
 50616 operation may fail.

50617 If the value of `wc` equals that of the macro `WEOF`, the operation shall fail and the input stream
 50618 shall be left unchanged.

50619 A successful call to `ungetwc()` shall clear the end-of-file indicator for the stream. The value of the
 50620 file-position indicator for the stream after reading or discarding all pushed-back characters shall
 50621 be the same as it was before the characters were pushed back. The file-position indicator is
 50622 decremented (by one or more) by each successful call to `ungetwc()`; if its value was 0 before a
 50623 call, its value is unspecified after the call.

50624 **RETURN VALUE**
 50625 Upon successful completion, `ungetwc()` shall return the wide-character code corresponding to
 50626 the pushed-back character. Otherwise, it shall return `WEOF`.

50627 **ERRORS**
 50628 The `ungetwc()` function may fail if:

50629 CX [EILSEQ] An invalid character sequence is detected, or a wide-character code does not
 50630 correspond to a valid character.

50631 **EXAMPLES**
 50632 None.

50633 **APPLICATION USAGE**
 50634 None.

50635 **RATIONALE**
 50636 None.

50637 **FUTURE DIRECTIONS**
 50638 None.

50639
50640
50641
50642
50643
50644
50645
50646
50647

SEE ALSO

fseek(), *fsetpos()*, *read()*, *rewind()*, *setbuf()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdio.h>`, `<wchar.h>`

CHANGE HISTORY

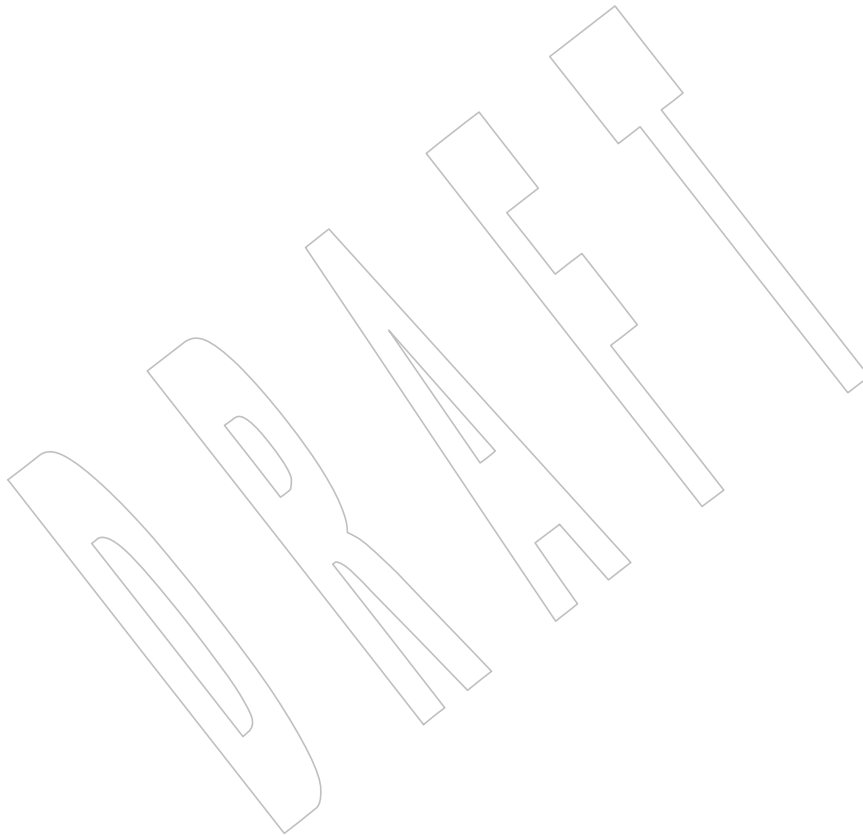
First released in Issue 4. Derived from the MSE working draft.

Issue 5

The Optional Header (OH) marking is removed from `<stdio.h>`.

Issue 6

The [EILSEQ] optional error condition is marked CX.



50648 **NAME**50649 `unlink, unlinkat` — remove a directory entry relative to directory file descriptor50650 **SYNOPSIS**50651 `#include <unistd.h>`50652 `int unlink(const char *path);`50653 `int unlinkat(int fd, const char *path, int flag);`50654 **DESCRIPTION**

50655 The `unlink()` function shall remove a link to a file. If `path` names a symbolic link, `unlink()` shall
 50656 remove the symbolic link named by `path` and shall not affect any file or directory named by the
 50657 contents of the symbolic link. Otherwise, `unlink()` shall remove the link named by the pathname
 50658 pointed to by `path` and shall decrement the link count of the file referenced by the link.

50659 When the file's link count becomes 0 and no process has the file open, the space occupied by the
 50660 file shall be freed and the file shall no longer be accessible. If one or more processes have the file
 50661 open when the last link is removed, the link shall be removed before `unlink()` returns, but the
 50662 removal of the file contents shall be postponed until all references to the file are closed.

50663 The `path` argument shall not name a directory unless the process has appropriate privileges and
 50664 the implementation supports using `unlink()` on directories.

50665 Upon successful completion, `unlink()` shall mark for update the `st_ctime` and `st_mtime` fields of
 50666 the parent directory. Also, if the file's link count is not 0, the `st_ctime` field of the file shall be
 50667 marked for update.

50668 The `unlinkat()` function shall be equivalent to the `unlink()` or `rmdir()` function except in the case
 50669 where `path` specifies a relative path. In this case the directory entry to be removed is determined
 50670 relative to the directory associated with the file descriptor `fd` instead of the current working
 50671 directory. It is unspecified whether directory searches are permitted based on whether the file
 50672 was opened with search permission or on the current permissions of the directory underlying
 50673 the file descriptor.

50674 Values for `flag` are constructed by a bitwise-inclusive OR of flags from the following list, defined
 50675 in `<fcntl.h>`:

50676 `AT_REMOVEDIR` Remove the directory entry specified by `fd` and `path` as a directory, not a
 50677 normal file.

50678 If `unlinkat()` is passed the special value `AT_FDCWD` in the `fd` parameter, the current working
 50679 directory is used and the behavior shall be identical to a call to `unlink()` or `rmdir()` respectively,
 50680 depending on whether or not the `AT_REMOVEDIR` bit is set in `flag`.

50681 **RETURN VALUE**

50682 Upon successful completion, these functions shall return 0. Otherwise, these functions shall
 50683 return `-1` and set `errno` to indicate the error. If `-1` is returned, the named file shall not be changed.

50684 **ERRORS**

50685 These functions shall fail and shall not unlink the file if:

50686 `[EACCES]` Search permission is denied for a component of the path prefix, or write
 50687 permission is denied on the directory containing the directory entry to be
 50688 removed.

50689 `[EBUSY]` The file named by the `path` argument cannot be unlinked because it is being
 50690 used by the system or another process and the implementation considers this
 50691 an error.

50692		[ELOOP]	A loop exists in symbolic links encountered during resolution of the <i>path</i> argument.
50693			
50694		[ENAMETOOLONG]	The length of the <i>path</i> argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
50695			
50696			
50697		[ENOENT]	A component of <i>path</i> does not name an existing file or <i>path</i> is an empty string.
50698		[ENOTDIR]	A component of the path prefix is not a directory.
50699		[EPERM]	The file named by <i>path</i> is a directory, and either the calling process does not have appropriate privileges, or the implementation prohibits using <i>unlink()</i> on directories.
50700			
50701			
50702	XSI	[EPERM] or [EACCES]	The S_ISVTX flag is set on the directory containing the file referred to by the <i>path</i> argument and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges.
50703			
50704			
50705			
50706		[EROFS]	The directory entry to be unlinked is part of a read-only file system.
50707			The <i>unlinkat()</i> function shall fail if:
50708		[EBADF]	The <i>path</i> argument does not specify an absolute path and the <i>fd</i> argument is neither AT_FDCWD nor a valid file descriptor open for searching.
50709			
50710		[EEXIST] or [ENOTEMPTY]	The <i>flag</i> parameter has the AT_REMOVEDIR bit set and the <i>path</i> argument names a directory that is not an empty directory, or there are hard links to the directory other than dot or a single entry in dot-dot.
50711			
50712			
50713			
50714		[ENOTDIR]	The <i>flag</i> parameter has the AT_REMOVEDIR bit set and <i>path</i> does not name a directory.
50715			
50716			These functions may fail and not unlink the file if:
50717	XSI	[EBUSY]	The file named by <i>path</i> is a named STREAM.
50718		[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the <i>path</i> argument.
50719			
50720		[ENAMETOOLONG]	As a result of encountering a symbolic link in resolution of the <i>path</i> argument, the length of the substituted pathname string exceeded {PATH_MAX}.
50721			
50722			
50723		[ETXTBSY]	The entry to be unlinked is the last directory entry to a pure procedure (shared text) file that is being executed.
50724			
50725			The <i>unlinkat()</i> function may fail if:
50726		[EINVAL]	The value of the <i>flag</i> argument is not valid.
50727		[ENOTDIR]	The <i>path</i> argument is not an absolute path and <i>fd</i> is neither AT_FDCWD nor a file descriptor associated with a directory.
50728			

50729 EXAMPLES

50730 **Removing a Link to a File**

50731 The following example shows how to remove a link to a file named `/home/cnd/mod1` by
50732 removing the entry named `/modules/pass1`.

```
50733 #include <unistd.h>
50734
50734 char *path = "/modules/pass1";
50735 int status;
50736 ...
50737 status = unlink(path);
```

50738 **Checking for an Error**

50739 The following example fragment creates a temporary password lock file named **LOCKFILE**,
50740 which is defined as `/etc/ptmp`, and gets a file descriptor for it. If the file cannot be opened for
50741 writing, `unlink()` is used to remove the link between the file descriptor and **LOCKFILE**.

```
50742 #include <sys/types.h>
50743 #include <stdio.h>
50744 #include <fcntl.h>
50745 #include <errno.h>
50746 #include <unistd.h>
50747 #include <sys/stat.h>
50748
50748 #define LOCKFILE "/etc/ptmp"
50749
50749 int pfd; /* Integer for file descriptor returned by open call. */
50750 FILE *fpfd; /* File pointer for use in putpwent(). */
50751 ...
50752 /* Open password Lock file. If it exists, this is an error. */
50753 if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL, S_IRUSR
50754 | S_IWUSR | S_IRGRP | S_IROTH)) == -1) {
50755     fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
50756     exit(1);
50757 }
50758
50758 /* Lock file created; proceed with fdopen of lock file so that
50759 putpwent() can be used.
50760 */
50761 if ((fpfd = fdopen(pfd, "w")) == NULL) {
50762     close(pfd);
50763     unlink(LOCKFILE);
50764     exit(1);
50765 }
50766 }
```

50766 **Replacing Files**

50767 The following example fragment uses `unlink()` to discard links to files, so that they can be
50768 replaced with new versions of the files. The first call removes the link to **LOCKFILE** if an error
50769 occurs. Successive calls remove the links to **SAVEFILE** and **PASSWDFILE** so that new links can
50770 be created, then removes the link to **LOCKFILE** when it is no longer needed.

```
50771 #include <sys/types.h>
50772 #include <stdio.h>
50773 #include <fcntl.h>
```

```

50774     #include <errno.h>
50775     #include <unistd.h>
50776     #include <sys/stat.h>
50777
50777     #define LOCKFILE "/etc/ptmp"
50778     #define PASSWDFILE "/etc/passwd"
50779     #define SAVEFILE "/etc/opasswd"
50780     ...
50781     /* If no change was made, assume error and leave passwd unchanged. */
50782     if (!valid_change) {
50783         fprintf(stderr, "Could not change password for user %s\n", user);
50784         unlink(LOCKFILE);
50785         exit(1);
50786     }
50787
50787     /* Change permissions on new password file. */
50788     chmod(LOCKFILE, S_IRUSR | S_IRGRP | S_IROTH);
50789
50789     /* Remove saved password file. */
50790     unlink(SAVEFILE);
50791
50791     /* Save current password file. */
50792     link(PASSWDFILE, SAVEFILE);
50793
50793     /* Remove current password file. */
50794     unlink(PASSWDFILE);
50795
50795     /* Save new password file as current password file. */
50796     link(LOCKFILE, PASSWDFILE);
50797
50797     /* Remove lock file. */
50798     unlink(LOCKFILE);
50799
50799     exit(0);

```

APPLICATION USAGE

Applications should use *rmdir()* to remove a directory.

RATIONALE

Unlinking a directory is restricted to the superuser in many historical implementations for reasons given in *link()* (see also *rename()*).

The meaning of [EBUSY] in historical implementations is “mount point busy”. Since this volume of IEEE Std 1003.1-200x does not cover the system administration concepts of mounting and unmounting, the description of the error was changed to “resource busy”. (This meaning is used by some device drivers when a second process tries to open an exclusive use device.) The wording is also intended to allow implementations to refuse to remove a directory if it is the root or current working directory of any process.

The standard developers reviewed TR 24715-2006 and noted that LSB-conforming implementations may return [EISDIR] instead of [EPERM] when unlinking a directory. A change to permit this behavior by changing the requirement for [EPERM] to [EPERM] or [EISDIR] was considered, but decided against since it would break existing strictly conforming and conforming applications. Applications written for portability to both this standard and the LSB should be prepared to handle either error code.

The purpose of the *unlinkat()* function is to remove directory entries in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *unlink()*, resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *unlinkat()* function it can be guaranteed that the removed directory entry is located relative to the desired directory.

50822
50823
50824
50825
50826
50827
50828
50829
50830
50831
50832
50833
50834
50835
50836
50837
50838
50839
50840
50841
50842
50843
50844
50845

FUTURE DIRECTIONS

None.

SEE ALSO

close(), *link()*, *remove()*, *rename()*, *rmdir()*, *symlink()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<fcntl.h>`, `<unistd.h>`

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The [EBUSY] error is added to the optional part of the ERRORS section.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the effect is specified if *path* specifies a symbolic link.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.
- The [ETXTBSY] optional error condition is added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

The normative text is updated to avoid use of the term “must” for application requirements.

Issue 7

Text arising from the LSB Conflicts TR is added to the RATIONALE about the use of [EPERM] and [EISDIR].

The *unlinkat()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

50846 **NAME**
50847 `unlinkat` — remove a directory entry relative to directory file descriptor

50848 **SYNOPSIS**
50849 `#include <unistd.h>`
50850 `int unlinkat(int fd, const char *path, int flag);`

50851 **DESCRIPTION**
50852 Refer to *unlink()*.



50853 **NAME**
 50854 unlockpt — unlock a pseudo-terminal master/slave pair

50855 **SYNOPSIS**

```
50856 XSI #include <stdlib.h>
50857 int unlockpt(int fildev);
```

50858 **DESCRIPTION**

50859 The *unlockpt()* function shall unlock the slave pseudo-terminal device associated with the master
 50860 to which *fildev* refers.

50861 Conforming applications shall ensure that they call *unlockpt()* before opening the slave side of a
 50862 pseudo-terminal device.

50863 **RETURN VALUE**

50864 Upon successful completion, *unlockpt()* shall return 0. Otherwise, it shall return -1 and set *errno*
 50865 to indicate the error.

50866 **ERRORS**

50867 The *unlockpt()* function may fail if:

50868	[EBADF]	The <i>fildev</i> argument is not a file descriptor open for writing.
50869	[EINVAL]	The <i>fildev</i> argument is not associated with a master pseudo-terminal device.

50870 **EXAMPLES**

50871 None.

50872 **APPLICATION USAGE**

50873 None.

50874 **RATIONALE**

50875 None.

50876 **FUTURE DIRECTIONS**

50877 None.

50878 **SEE ALSO**

50879 *grantpt()*, *open()*, *ptsname()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

50880 **CHANGE HISTORY**

50881 First released in Issue 4, Version 2.

50882 **Issue 5**

50883 Moved from X/OPEN UNIX extension to BASE.

50884 **Issue 6**

50885 The normative text is updated to avoid use of the term “must” for application requirements.

50886

NAME

50887

unsetenv — remove an environment variable

50888

SYNOPSIS

50889

CX `#include <stdlib.h>`

50890

`int unsetenv(const char *name);`

50891

DESCRIPTION

50892

The `unsetenv()` function shall remove an environment variable from the environment of the calling process. The `name` argument points to a string, which is the name of the variable to be removed. The named argument shall not contain an '=' character. If the named variable does not exist in the current environment, the environment shall be unchanged and the function is considered to have completed successfully.

50893

50894

50895

50896

50897

If the application modifies `environ` or the pointers to which it points, the behavior of `unsetenv()` is undefined. The `unsetenv()` function shall update the list of pointers to which `environ` points.

50898

50899

The `unsetenv()` function need not be thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

50900

50901

RETURN VALUE

50902

Upon successful completion, zero shall be returned. Otherwise, -1 shall be returned, `errno` set to indicate the error, and the environment shall be unchanged.

50903

50904

ERRORS

50905

The `unsetenv()` function shall fail if:

50906

[EINVAL]

The `name` argument is a null pointer, points to an empty string, or points to a string containing an '=' character.

50907

50908

EXAMPLES

50909

None.

50910

APPLICATION USAGE

50911

None.

50912

RATIONALE

50913

Refer to the RATIONALE section in `setenv()`.

50914

FUTURE DIRECTIONS

50915

None.

50916

SEE ALSO

50917

`getenv()`, `setenv()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdlib.h>`, `<sys/types.h>`, `<unistd.h>`

50918

50919

CHANGE HISTORY

50920

First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

50921 **NAME**

50922 uselocale — use locale in current thread

50923 **SYNOPSIS**

```
50924 CX      #include <locale.h>
50925         locale_t uselocale(locale_t newloc);
```

50926 **DESCRIPTION**

50927 The *uselocale()* function shall set the current locale for the current thread to the locale
50928 represented by *newloc*.

50929 The value for the *newloc* argument shall be one of the following:

- 50930 1. A value returned by the *newlocale()* or *duplocale()* functions
- 50931 2. The special locale object descriptor *LC_GLOBAL_LOCALE*
- 50932 3. **(locale_t)0**

50933 Once the *uselocale()* function has been called to install a thread-local locale, the behavior of every
50934 interface using data from the current locale shall be affected for the calling thread. The current
50935 locale for other threads shall remain unchanged.

50936 If the *newloc* argument is a null pointer, the object returned is the current locale or
50937 *LC_GLOBAL_LOCALE* if there has been no previous call to *uselocale()* for the current thread.

50938 If the *newloc* argument is *LC_GLOBAL_LOCALE*, the thread shall use the global locale
50939 determined by the *setlocale()* function.

50940 **RETURN VALUE**

50941 The *uselocale()* function returns the locale handle from the previous call for the current thread. If
50942 there was no such previous call, the function shall return the value *LC_GLOBAL_LOCALE*.

50943 **ERRORS**

50944 The *uselocale()* function may fail if:

50945 [EINVAL] *locale* is not a valid locale object.

50946 **EXAMPLES**

50947 None.

50948 **APPLICATION USAGE**

50949 Unlike the *setlocale()* function, the *uselocale()* function does not allow replacing some locale
50950 categories only. Applications that need to install a locale which differs only in a few categories
50951 must use *newlocale()* to change a locale object equivalent to the currently used locale and install
50952 it.

50953 **RATIONALE**

50954 None.

50955 **FUTURE DIRECTIONS**

50956 None.

50957 **SEE ALSO**

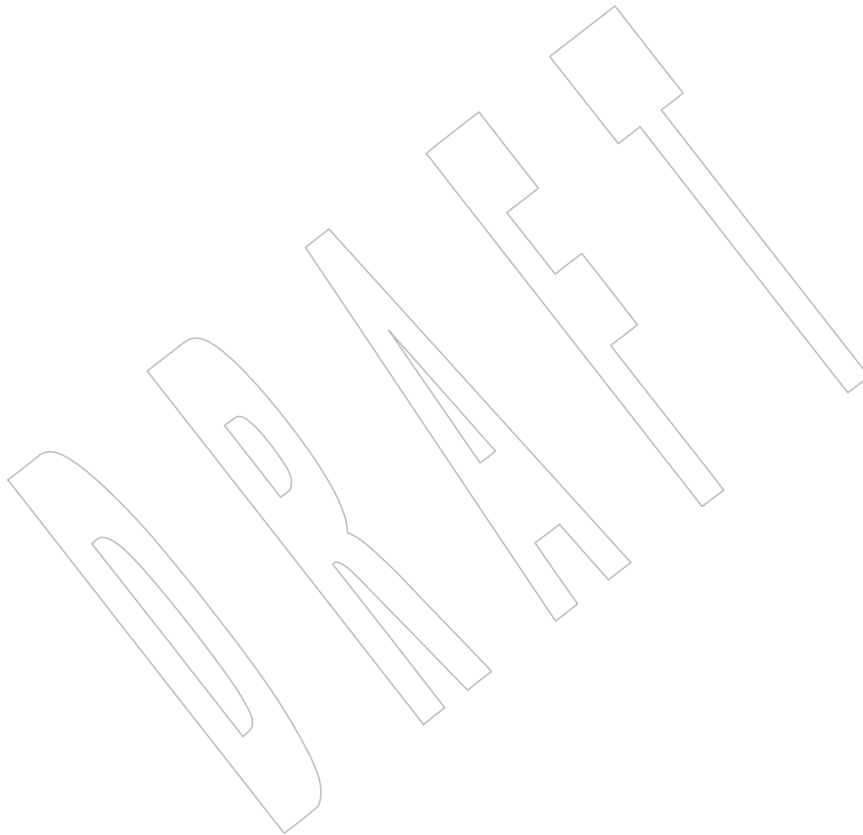
50958 *duplocale()*, *freelocale()*, *newlocale()*, *setlocale()*, the Base Definitions volume of
50959 IEEE Std 1003.1-200x, **<locale.h>**

50960

50961

CHANGE HISTORY

First released in Issue 7.



50962 **NAME**50963 `utime` — set file access and modification times50964 **SYNOPSIS**50965 `#include <utime.h>`50966 `int utime(const char *path, const struct utimbuf *times);`50967 **DESCRIPTION**50968 The `utime()` function shall set the access and modification times of the file named by the `path`
50969 argument.50970 If `times` is a null pointer, the access and modification times of the file shall be set to the current
50971 time. The effective user ID of the process shall match the owner of the file, or the process has
50972 write permission to the file or has appropriate privileges, to use `utime()` in this manner.50973 If `times` is not a null pointer, `times` shall be interpreted as a pointer to a **utimbuf** structure and the
50974 access and modification times shall be set to the values contained in the designated structure.
50975 Only a process with the effective user ID equal to the user ID of the file or a process with
50976 appropriate privileges may use `utime()` this way.50977 The **utimbuf** structure is defined in the `<utime.h>` header. The times in the structure **utimbuf**
50978 are measured in seconds since the Epoch.50979 Upon successful completion, `utime()` shall mark the time of the last file status change, `st_ctime`, to
50980 be updated; see `<sys/stat.h>`.50981 **RETURN VALUE**50982 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and `errno` shall
50983 be set to indicate the error, and the file times shall not be affected.50984 **ERRORS**50985 The `utime()` function shall fail if:50986 [EACCES] Search permission is denied by a component of the path prefix; or the `times`
50987 argument is a null pointer and the effective user ID of the process does not
50988 match the owner of the file, the process does not have write permission for the
50989 file, and the process does not have appropriate privileges.50990 [ELOOP] A loop exists in symbolic links encountered during resolution of the `path`
50991 argument.50992 [ENAMETOOLONG] The length of the `path` argument exceeds {PATH_MAX} or a pathname
50993 component is longer than {NAME_MAX}.
5099450995 [ENOENT] A component of `path` does not name an existing file or `path` is an empty string.

50996 [ENOTDIR] A component of the path prefix is not a directory.

50997 [EPERM] The `times` argument is not a null pointer and the effective user ID of the calling
50998 process does not match the owner of the file and the calling process does not
50999 have the appropriate privileges.

51000 [EROFS] The file system containing the file is read-only.

51001 The `utime()` function may fail if:51002 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
51003 resolution of the `path` argument.

51004 [ENAMETOOLONG]

51005 As a result of encountering a symbolic link in resolution of the *path* argument,
51006 the length of the substituted pathname string exceeded {PATH_MAX}.

51007 **EXAMPLES**

51008 None.

51009 **APPLICATION USAGE**

51010 None.

51011 **RATIONALE**

51012 The *actime* structure member must be present so that an application may set it, even though an
51013 implementation may ignore it and not change the access time on the file. If an application
51014 intends to leave one of the times of a file unchanged while changing the other, it should use
51015 *stat()* to retrieve the file's *st_atime* and *st_mtime* parameters, set *actime* and *modtime* in the buffer,
51016 and change one of them before making the *utime()* call.

51017 **FUTURE DIRECTIONS**

51018 None.

51019 **SEE ALSO**

51020 *utimes()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<sys/stat.h>`, `<utime.h>`

51021 **CHANGE HISTORY**

51022 First released in Issue 1. Derived from Issue 1 of the SVID.

51023 **Issue 6**

51024 The following new requirements on POSIX implementations derive from alignment with the
51025 Single UNIX Specification:

- 51026 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
51027 required for conforming implementations of previous POSIX specifications, it was not
51028 required for UNIX applications.
- 51029 • The [ELOOP] mandatory error condition is added.
- 51030 • A second [ENAMETOOLONG] is added as an optional error condition.

51031 The following changes were made to align with the IEEE P1003.1a draft standard:

- 51032 • The [ELOOP] optional error condition is added.

51033 The normative text is updated to avoid use of the term “must” for application requirements.

51034 **NAME**

51035 futimesat, utimes — set file access and modification times relative to directory file descriptor

51036 **SYNOPSIS**

```
51037 XSI #include <sys/time.h>
51038 int futimesat(int fd, const char *path, const struct timeval times[2]);
51039 int utimes(const char *path, const struct timeval times[2]);
```

51040 **DESCRIPTION**

51041 The *utimes()* function shall set the access and modification times of the file pointed to by the *path*
 51042 argument to the value of the *times* argument. The *utimes()* function allows time specifications
 51043 accurate to the microsecond.

51044 For *utimes()*, the *times* argument is an array of **timeval** structures. The first array member
 51045 represents the date and time of last access, and the second member represents the date and time
 51046 of last modification. The times in the **timeval** structure are measured in seconds and
 51047 microseconds since the Epoch, although rounding toward the nearest second may occur.

51048 If the *times* argument is a null pointer, the access and modification times of the file shall be set to
 51049 the current time. The effective user ID of the process shall match the owner of the file, or has
 51050 write access to the file or appropriate privileges to use this call in this manner. Upon completion,
 51051 *utimes()* shall mark the time of the last file status change, *st_ctime*, for update.

51052 The *futimesat()* function shall be equivalent to the *utimes()* function except in the case where *path*
 51053 specifies a relative path. In this case the access and modification time is set to that of a file
 51054 relative to the directory associated with the file descriptor *fd* instead of the current working
 51055 directory. It is unspecified whether directory searches are permitted based on whether the file
 51056 was opened with search permission or on the current permissions of the directory underlying
 51057 the file descriptor.

51058 If *futimesat()* is passed the special value *AT_FDCWD* in the *fd* parameter, the current working
 51059 directory is used and the behavior shall be identical to a call to *utimes()*.

51060 **RETURN VALUE**

51061 Upon successful completion, these functions shall return 0. Otherwise, these functions shall
 51062 return -1 and set *errno* to indicate the error. If -1 is returned, the file times shall not be affected.

51063 **ERRORS**

51064 These functions shall fail if:

51065 [EACCES] Search permission is denied by a component of the path prefix; or the *times*
 51066 argument is a null pointer and the effective user ID of the process does not
 51067 match the owner of the file and write access is denied.

51068 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 51069 argument.

51070 [ENAMETOOLONG]
 51071 The length of the *path* argument exceeds {PATH_MAX} or a pathname
 51072 component is longer than {NAME_MAX}.

51073 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

51074 [ENOTDIR] A component of the path prefix is not a directory.

- 51075 [EPERM] The *times* argument is not a null pointer and the calling process' effective user
51076 ID has write access to the file but does not match the owner of the file and the
51077 calling process does not have the appropriate privileges.
- 51078 [EROFS] The file system containing the file is read-only.
- 51079 The *futimesat()* function shall fail if:
- 51080 [EBADF] The *path* argument does not specify an absolute path and the *fd* argument is
51081 neither AT_FDCWD nor a valid file descriptor open for searching.
- 51082 These functions may fail if:
- 51083 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
51084 resolution of the *path* argument.
- 51085 [ENAMETOOLONG]
51086 Pathname resolution of a symbolic link produced an intermediate result
51087 whose length exceeds {PATH_MAX}.
- 51088 The *futimesat()* function may fail if:
- 51089 [ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a
51090 file descriptor associated with a directory.

EXAMPLES

51091 None.
51092

APPLICATION USAGE

51093 For applications portability, the *utime()* function should be used to set file access and
51094 modification times instead of *utimes()*.
51095

RATIONALE

51096 The purpose of the *futimesat()* function is to set the access and modification time of files in
51097 directories other than the current working directory without exposure to race conditions. Any
51098 part of the path of a file could be changed in parallel to a call to *utimes()*, resulting in unspecified
51099 behavior. By opening a file descriptor for the target directory and using the *futimesat()* function
51100 it can be guaranteed that the changed file is located relative to the desired directory.
51101

FUTURE DIRECTIONS

51102 None.
51103

SEE ALSO

51104 *utime()*, the Base Definitions volume of IEEE Std 1003.1-200x, <sys/time.h>
51105

CHANGE HISTORY

51106 First released in Issue 4, Version 2.
51107

Issue 5

51108 Moved from X/OPEN UNIX extension to BASE.
51109

Issue 6

51110 This function is marked LEGACY.
51111

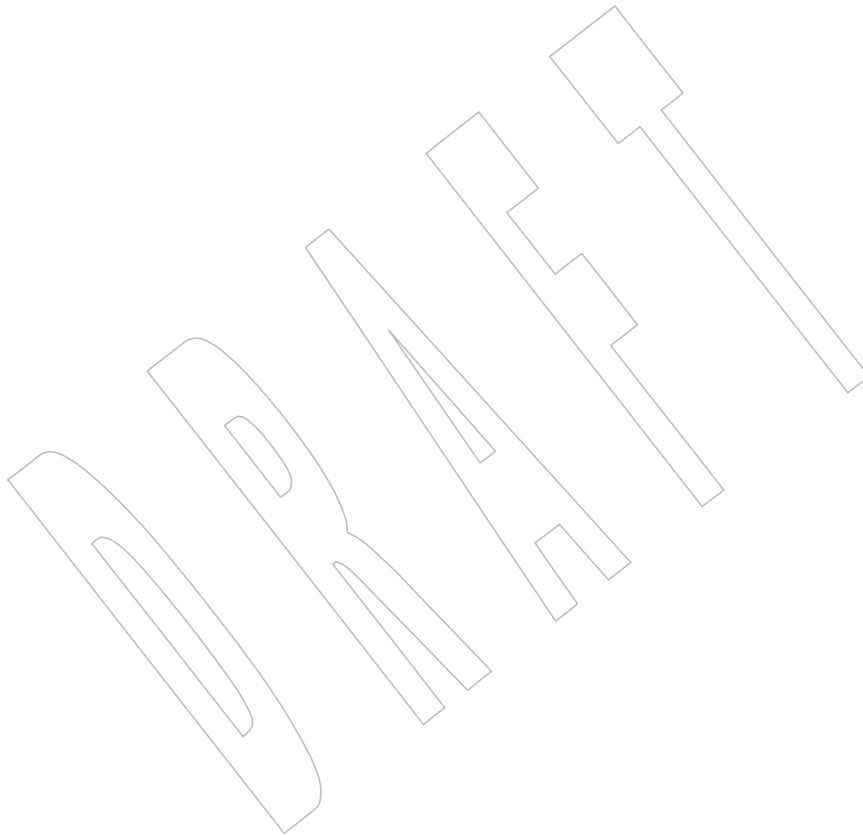
51112 The normative text is updated to avoid use of the term "must" for application requirements.

51113 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
51114 [ELOOP] error condition is added.

Issue 7

51115 The LEGACY marking is removed.
51116

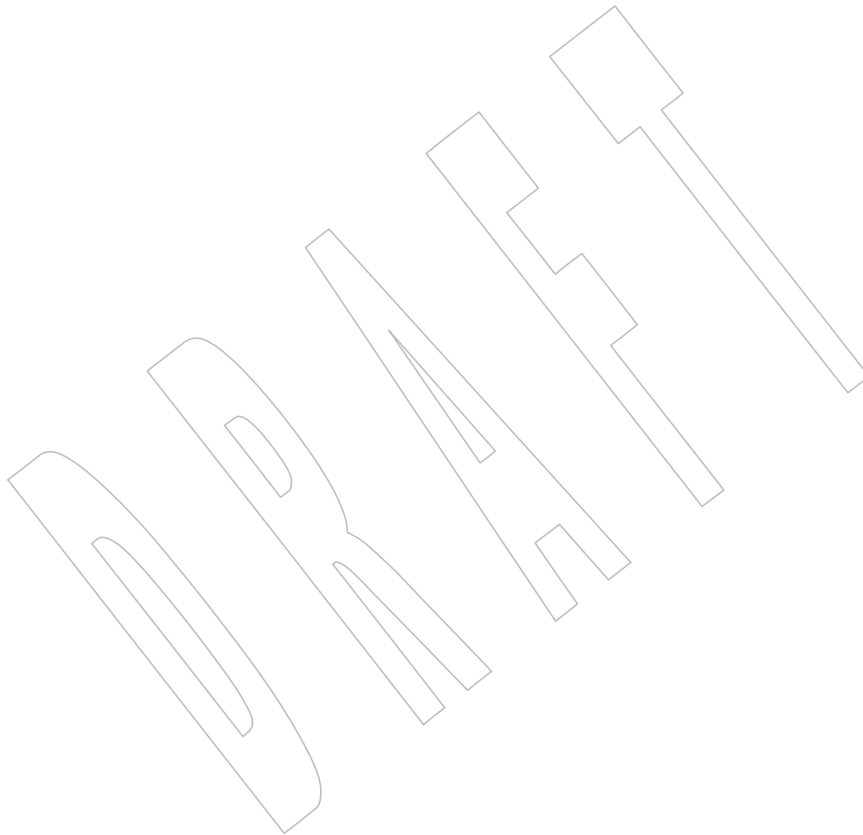
51117 The *futimesat()* function is added from The Open Group Technical Standard, 2006, Extended API
51118 Set Part 2.



51119 **NAME**
51120 `va_arg`, `va_copy`, `va_end`, `va_start` — handle variable argument list

51121 **SYNOPSIS**
51122 `#include <stdarg.h>`
51123 `type va_arg(va_list ap, type);`
51124 `void va_copy(va_list dest, va_list src);`
51125 `void va_end(va_list ap);`
51126 `void va_start(va_list ap, argN);`

51127 **DESCRIPTION**
51128 Refer to the Base Definitions volume of IEEE Std 1003.1-200x, `<stdarg.h>`.



51129 **NAME**

51130 vfprintf, vprintf, vsnprintf, vsprintf — format output of a stdarg argument list

51131 **SYNOPSIS**

51132 #include <stdarg.h>

51133 #include <stdio.h>

51134 int vfprintf(FILE *restrict stream, const char *restrict format,
51135 va_list ap);

51136 int vprintf(const char *restrict format, va_list ap);

51137 int vsnprintf(char *restrict s, size_t n, const char *restrict format,
51138 va_list ap);

51139 int vsprintf(char *restrict s, const char *restrict format, va_list ap);

51140 **DESCRIPTION**51141 CX The functionality described on this reference page is aligned with the ISO C standard. Any
51142 conflict between the requirements described here and the ISO C standard is unintentional. This
51143 volume of IEEE Std 1003.1-200x defers to the ISO C standard.51144 The *vprintf()*, *vfprintf()*, *vsnprintf()*, and *vsprintf()* functions shall be equivalent to *printf()*,
51145 *fprintf()*, *snprintf()*, and *sprintf()* respectively, except that instead of being called with a variable
51146 number of arguments, they are called with an argument list as defined by <stdarg.h>.51147 These functions shall not invoke the *va_end* macro. As these functions invoke the *va_arg* macro,
51148 the value of *ap* after the return is unspecified.51149 **RETURN VALUE**51150 Refer to *fprintf()*.51151 **ERRORS**51152 Refer to *fprintf()*.51153 **EXAMPLES**

51154 None.

51155 **APPLICATION USAGE**51156 Applications using these functions should call *va_end(ap)* afterwards to clean up.51157 **RATIONALE**

51158 None.

51159 **FUTURE DIRECTIONS**

51160 None.

51161 **SEE ALSO**51162 *fprintf()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdarg.h>, <stdio.h>51163 **CHANGE HISTORY**

51164 First released in Issue 1. Derived from Issue 1 of the SVID.

51165 **Issue 5**51166 The *vsnprintf()* function is added.51167 **Issue 6**51168 The *vfprintf()*, *vprintf()*, *vsnprintf()*, and *vsprintf()* functions are updated for alignment with the
51169 ISO/IEC 9899:1999 standard.

51170 **NAME**51171 `vfscanf`, `vscanf`, `vsscanf` — format input of a stdarg argument list51172 **SYNOPSIS**51173 `#include <stdarg.h>`51174 `#include <stdio.h>`51175 `int vfscanf(FILE *restrict stream, const char *restrict format,`
51176 `va_list arg);`51177 `int vscanf(const char *restrict format, va_list arg);`51178 `int vsscanf(const char *restrict s, const char *restrict format,`
51179 `va_list arg);`51180 **DESCRIPTION**51181 CX The functionality described on this reference page is aligned with the ISO C standard. Any
51182 conflict between the requirements described here and the ISO C standard is unintentional. This
51183 volume of IEEE Std 1003.1-200x defers to the ISO C standard.51184 The `vscanf()`, `vfscanf()`, and `vsscanf()` functions shall be equivalent to the `scanf()`, `fscanf()`, and
51185 `sscanf()` functions, respectively, except that instead of being called with a variable number of
51186 arguments, they are called with an argument list as defined in the `<stdarg.h>` header. These
51187 functions shall not invoke the `va_end` macro. As these functions invoke the `va_arg` macro, the
51188 value of `ap` after the return is unspecified.51189 **RETURN VALUE**51190 Refer to `fscanf()`.51191 **ERRORS**51192 Refer to `fscanf()`.51193 **EXAMPLES**

51194 None.

51195 **APPLICATION USAGE**51196 Applications using these functions should call `va_end(ap)` afterwards to clean up.51197 **RATIONALE**

51198 None.

51199 **FUTURE DIRECTIONS**

51200 None.

51201 **SEE ALSO**51202 `fscanf()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<stdarg.h>`, `<stdio.h>`51203 **CHANGE HISTORY**

51204 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

51205 **NAME**

51206 vfwprintf, vswprintf, vwprintf — wide-character formatted output of a stdarg argument list

51207 **SYNOPSIS**

51208 #include <stdarg.h>

51209 #include <stdio.h>

51210 #include <wchar.h>

51211 int vfwprintf(FILE *restrict stream, const wchar_t *restrict format,
51212 va_list arg);

51213 int vswprintf(wchar_t *restrict ws, size_t n,

51214 const wchar_t *restrict format, va_list arg);

51215 int vwprintf(const wchar_t *restrict format, va_list arg);

51216 **DESCRIPTION**51217 CX The functionality described on this reference page is aligned with the ISO C standard. Any
51218 conflict between the requirements described here and the ISO C standard is unintentional. This
51219 volume of IEEE Std 1003.1-200x defers to the ISO C standard.51220 The *vfwprintf()*, *vswprintf()*, and *vwprintf()* functions shall be equivalent to *fprintf()*, *swprintf()*,
51221 and *wprintf()* respectively, except that instead of being called with a variable number of
51222 arguments, they are called with an argument list as defined by <stdarg.h>.51223 These functions shall not invoke the *va_end* macro. However, as these functions do invoke the
51224 *va_arg* macro, the value of *ap* after the return is unspecified.51225 **RETURN VALUE**51226 Refer to *fprintf()*.51227 **ERRORS**51228 Refer to *fprintf()*.51229 **EXAMPLES**

51230 None.

51231 **APPLICATION USAGE**51232 Applications using these functions should call *va_end(ap)* afterwards to clean up.51233 **RATIONALE**

51234 None.

51235 **FUTURE DIRECTIONS**

51236 None.

51237 **SEE ALSO**51238 *fprintf()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdarg.h>, <stdio.h>,
51239 <wchar.h>51240 **CHANGE HISTORY**51241 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
51242 (E).51243 **Issue 6**51244 The *vfwprintf()*, *vswprintf()*, and *vwprintf()* prototypes are updated for alignment with the
51245 ISO/IEC 9899:1999 standard. ()

51246 **NAME**

51247 vfwscanf, vswscanf, vwscanf — wide-character formatted input of a stdarg argument list

51248 **SYNOPSIS**

51249 #include <stdarg.h>

51250 #include <stdio.h>

51251 #include <wchar.h>

51252 int vfwscanf(FILE *restrict stream, const wchar_t *restrict format,
51253 va_list arg);51254 int vswscanf(const wchar_t *restrict ws, const wchar_t *restrict format,
51255 va_list arg);

51256 int vwscanf(const wchar_t *restrict format, va_list arg);

51257 **DESCRIPTION**51258 CX The functionality described on this reference page is aligned with the ISO C standard. Any
51259 conflict between the requirements described here and the ISO C standard is unintentional. This
51260 volume of IEEE Std 1003.1-200x defers to the ISO C standard.51261 The *vfwscanf()*, *vswscanf()*, and *vwscanf()* functions shall be equivalent to the *fscanf()*,
51262 *swscanf()*, and *wscanf()* functions, respectively, except that instead of being called with a variable
51263 number of arguments, they are called with an argument list as defined in the <stdarg.h> header.
51264 These functions shall not invoke the *va_end* macro. As these functions invoke the *va_arg* macro,
51265 the value of *ap* after the return is unspecified.51266 **RETURN VALUE**51267 Refer to *fscanf()*.51268 **ERRORS**51269 Refer to *fscanf()*.51270 **EXAMPLES**

51271 None.

51272 **APPLICATION USAGE**51273 Applications using these functions should call *va_end(ap)* afterwards to clean up.51274 **RATIONALE**

51275 None.

51276 **FUTURE DIRECTIONS**

51277 None.

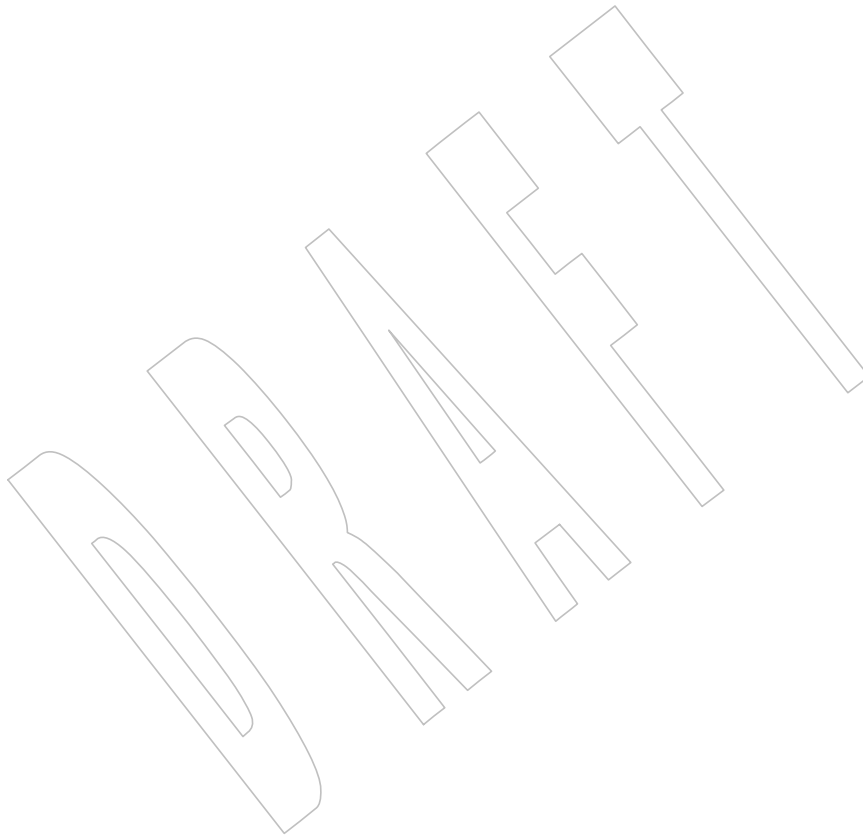
51278 **SEE ALSO**51279 *fscanf()*, the Base Definitions volume of IEEE Std 1003.1-200x, <stdarg.h>, <stdio.h>,
51280 <wchar.h>51281 **CHANGE HISTORY**

51282 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

51283 **NAME**
51284 `vprintf` — format the output of a `stdarg` argument list

51285 **SYNOPSIS**
51286 `#include <stdarg.h>`
51287 `#include <stdio.h>`
51288 `int vprintf(const char *restrict format, va_list ap);`

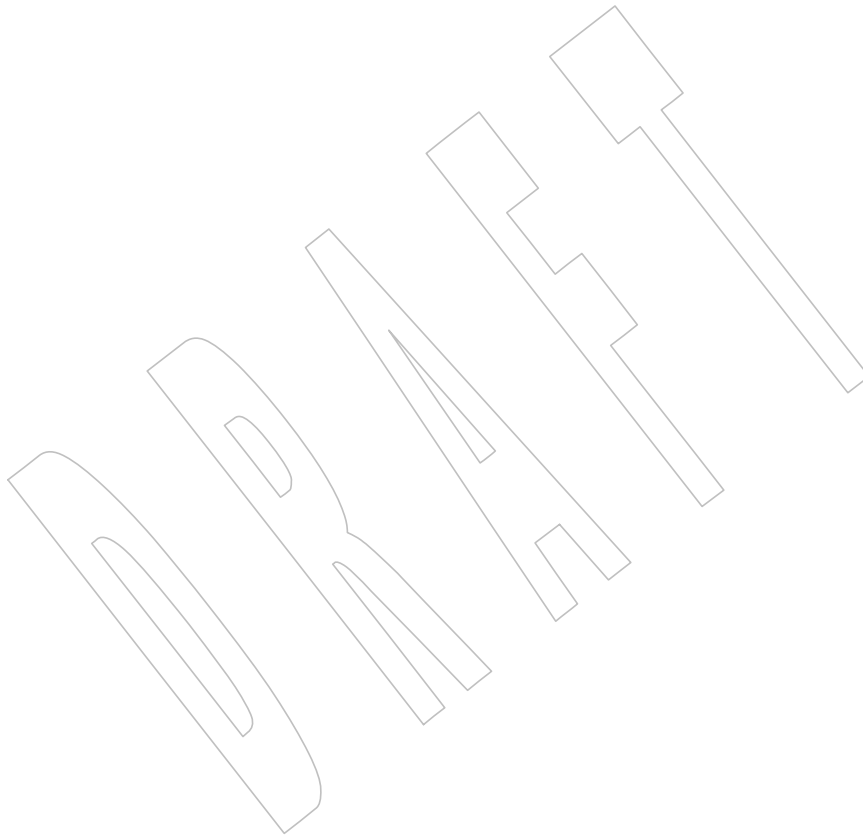
51289 **DESCRIPTION**
51290 Refer to *vfprintf()*.



51291 **NAME**
51292 vscanf — format input of a stdarg argument list

51293 **SYNOPSIS**
51294 #include <stdarg.h>
51295 #include <stdio.h>
51296 int vscanf(const char *restrict format, va_list arg);

51297 **DESCRIPTION**
51298 Refer to *vfscanf()*.



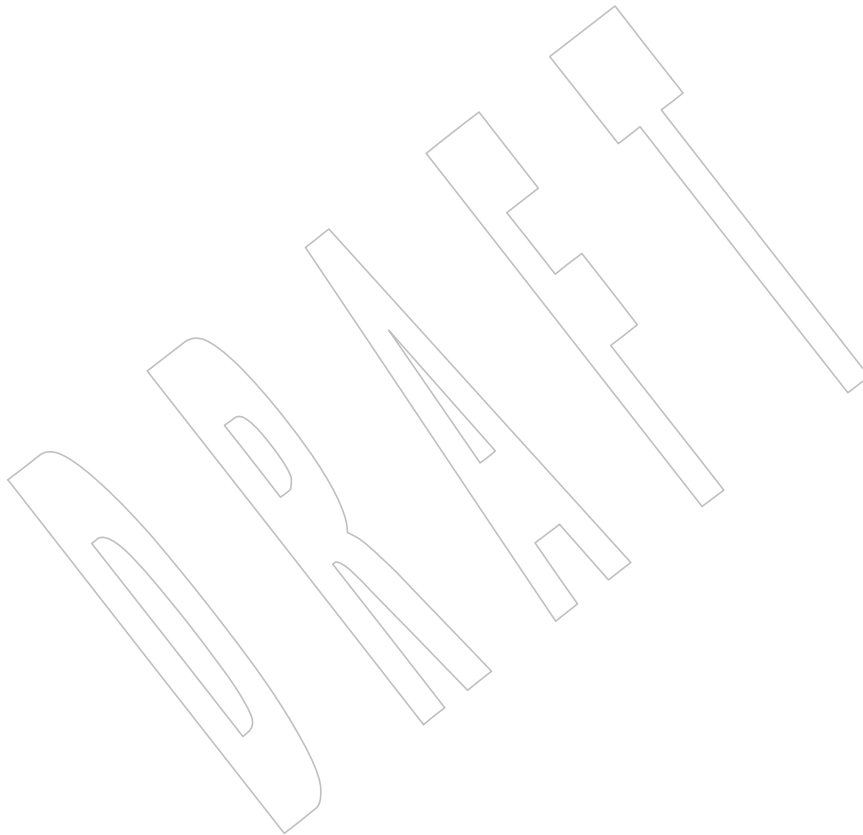
51299 **NAME**
51300 `vsnprintf`, `vsprintf` — format output of a stdarg argument list

51301 **SYNOPSIS**

```
51302 #include <stdarg.h>  
51303 #include <stdio.h>  
  
51304 int vsnprintf(char *restrict s, size_t n,  
51305             const char *restrict format, va_list ap);  
51306 int vsprintf(char *restrict s, const char *restrict format,  
51307             va_list ap);
```

51308 **DESCRIPTION**

51309 Refer to *fprintf()*.



51310 **NAME**
51311 vsscanf — format input of a stdarg argument list

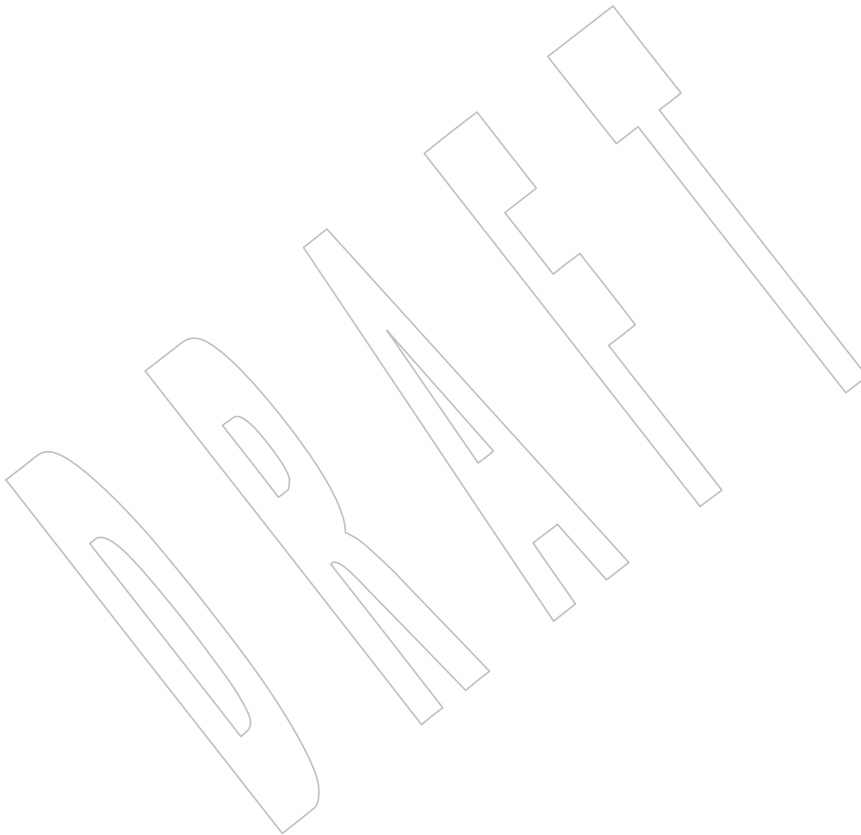
51312 **SYNOPSIS**

51313 #include <stdarg.h>
51314 #include <stdio.h>

51315 int vsscanf(const char *restrict *s*, const char *restrict *format*,
51316 va_list *arg*);

51317 **DESCRIPTION**

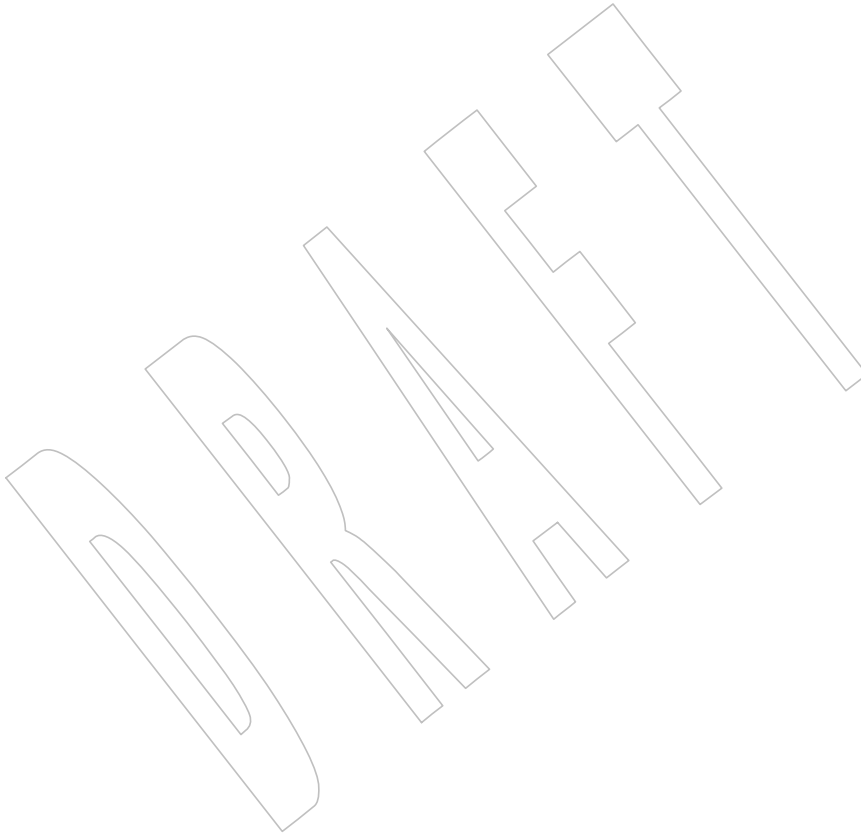
51318 Refer to *vfscanf()*.



51319 **NAME**
51320 `vswprintf` — wide-character formatted output of a `stdarg` argument list

51321 **SYNOPSIS**
51322 `#include <stdarg.h>`
51323 `#include <stdio.h>`
51324 `#include <wchar.h>`
51325 `int vswprintf(wchar_t *restrict ws, size_t n,`
51326 `const wchar_t *restrict format, va_list arg);`

51327 **DESCRIPTION**
51328 Refer to *[vfwprintf\(\)](#)*.



51329 **NAME**
51330 vswscanf — wide-character formatted input of a stdarg argument list

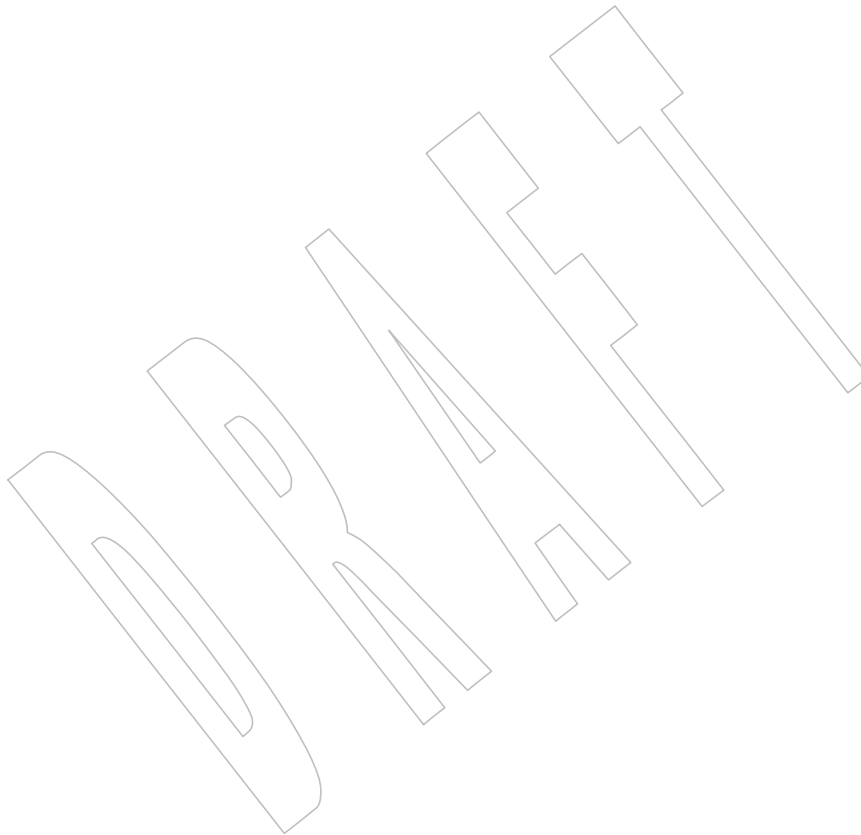
51331 **SYNOPSIS**

51332 #include <stdarg.h>
51333 #include <stdio.h>
51334 #include <wchar.h>

51335 int vswscanf(const wchar_t *restrict ws, const wchar_t *restrict format,
51336 va_list arg);

51337 **DESCRIPTION**

51338 Refer to *vscanf()*.



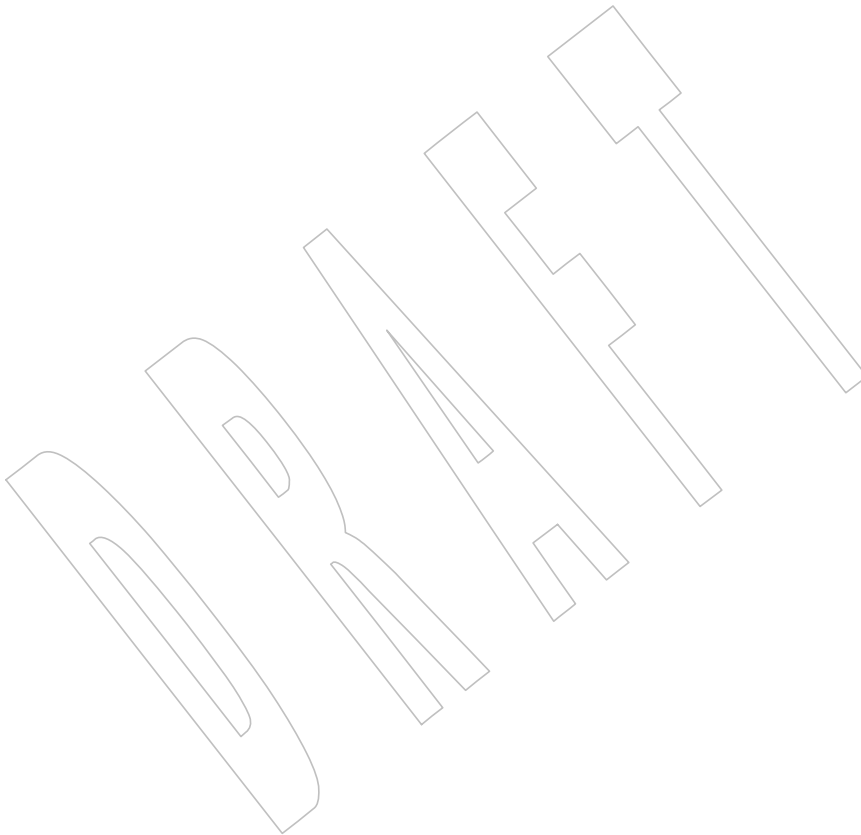
51339 **NAME**
51340 `vwprintf` — wide-character formatted output of a `stdarg` argument list

51341 **SYNOPSIS**

51342 `#include <stdarg.h>`
51343 `#include <stdio.h>`
51344 `#include <wchar.h>`
51345 `int vwprintf(const wchar_t *restrict format, va_list arg);`

51346 **DESCRIPTION**

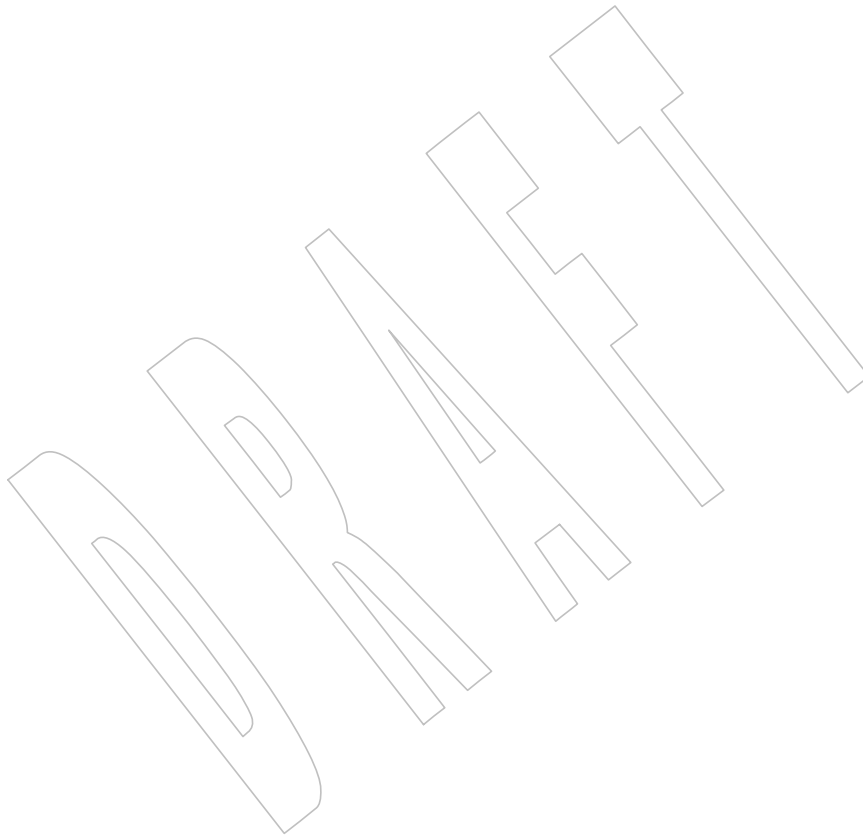
51347 Refer to *ofwprintf()*.



51348 **NAME**
51349 `vwscanf` — wide-character formatted input of a `stdarg` argument list

51350 **SYNOPSIS**
51351 `#include <stdarg.h>`
51352 `#include <stdio.h>`
51353 `#include <wchar.h>`
51354 `int vwscanf(const wchar_t *restrict format, va_list arg);`

51355 **DESCRIPTION**
51356 Refer to *[vwscanf\(\)](#)*.



51357 **NAME**
 51358 wait, waitpid — wait for a child process to stop or terminate

51359 **SYNOPSIS**
 51360 #include <sys/wait.h>
 51361 pid_t wait(int *stat_loc);
 51362 pid_t waitpid(pid_t pid, int *stat_loc, int options);

51363 **DESCRIPTION**
 51364 The *wait()* and *waitpid()* functions shall obtain status information pertaining to one of the
 51365 caller's child processes. Various options permit status information to be obtained for child
 51366 processes that have terminated or stopped. If status information is available for two or more
 51367 child processes, the order in which their status is reported is unspecified.

51368 The *wait()* function shall suspend execution of the calling thread until status information for one
 51369 of the terminated child processes of the calling process is available, or until delivery of a signal
 51370 whose action is either to execute a signal-catching function or to terminate the process. If more
 51371 than one thread is suspended in *wait()* or *waitpid()* awaiting termination of the same process,
 51372 exactly one thread shall return the process status at the time of the target process termination. If
 51373 status information is available prior to the call to *wait()*, return shall be immediate.

51374 The *waitpid()* function shall be equivalent to *wait()* if the *pid* argument is **(pid_t)-1** and the
 51375 *options* argument is 0. Otherwise, its behavior shall be modified by the values of the *pid* and
 51376 *options* arguments.

51377 The *pid* argument specifies a set of child processes for which *status* is requested. The *waitpid()*
 51378 function shall only return the status of a child process from this set:

- 51379 • If *pid* is equal to **(pid_t)-1**, *status* is requested for any child process. In this respect,
 51380 *waitpid()* is then equivalent to *wait()*.
- 51381 • If *pid* is greater than 0, it specifies the process ID of a single child process for which *status* is
 51382 requested.
- 51383 • If *pid* is 0, *status* is requested for any child process whose process group ID is equal to that
 51384 of the calling process.
- 51385 • If *pid* is less than **(pid_t)-1**, *status* is requested for any child process whose process group
 51386 ID is equal to the absolute value of *pid*.

51387 The *options* argument is constructed from the bitwise-inclusive OR of zero or more of the
 51388 following flags, defined in the **<sys/wait.h>** header:

51389 XSI **WCONTINUED** The *waitpid()* function shall report the status of any continued child process
 51390 specified by *pid* whose status has not been reported since it continued from a
 51391 job control stop.

51392 **WNOHANG** The *waitpid()* function shall not suspend execution of the calling thread if
 51393 *status* is not immediately available for one of the child processes specified by
 51394 *pid*.

51395 **WUNTRACED** The status of any child processes specified by *pid* that are stopped, and whose
 51396 status has not yet been reported since they stopped, shall also be reported to
 51397 the requesting process.

51398 XSI If the calling process has SA_NOCLDWAIT set or has SIGCHLD set to SIG_IGN, and the process
 51399 has no unwaited-for children that were transformed into zombie processes, the calling thread
 51400 shall block until all of the children of the process containing the calling thread terminate, and

51401		<code>wait()</code> and <code>waitpid()</code> shall fail and set <code>errno</code> to [ECHILD].
51402		If <code>wait()</code> or <code>waitpid()</code> return because the status of a child process is available, these functions shall return a value equal to the process ID of the child process. In this case, if the value of the argument <code>stat_loc</code> is not a null pointer, information shall be stored in the location pointed to by <code>stat_loc</code> . The value stored at the location pointed to by <code>stat_loc</code> shall be 0 if and only if the status returned is from a terminated child process that terminated by one of the following means:
51403		
51404		
51405		
51406		
51407		1. The process returned 0 from <code>main()</code> .
51408		2. The process called <code>_exit()</code> or <code>exit()</code> with a <code>status</code> argument of 0.
51409		3. The process was terminated because the last thread in the process terminated.
51410		Regardless of its value, this information may be interpreted using the following macros, which are defined in <code><sys/wait.h></code> and evaluate to integral expressions; the <code>stat_val</code> argument is the integer value pointed to by <code>stat_loc</code> .
51411		
51412		
51413		<code>WIFEXITED(stat_val)</code>
51414		Evaluates to a non-zero value if <code>status</code> was returned for a child process that terminated normally.
51415		
51416		<code>WEXITSTATUS(stat_val)</code>
51417		If the value of <code>WIFEXITED(stat_val)</code> is non-zero, this macro evaluates to the low-order 8 bits of the <code>status</code> argument that the child process passed to <code>_exit()</code> or <code>exit()</code> , or the value the child process returned from <code>main()</code> .
51418		
51419		
51420		<code>WIFSIGNALED(stat_val)</code>
51421		Evaluates to a non-zero value if <code>status</code> was returned for a child process that terminated due to the receipt of a signal that was not caught (see <code><signal.h></code>).
51422		
51423		<code>WTERMSIG(stat_val)</code>
51424		If the value of <code>WIFSIGNALED(stat_val)</code> is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.
51425		
51426		<code>WIFSTOPPED(stat_val)</code>
51427		Evaluates to a non-zero value if <code>status</code> was returned for a child process that is currently stopped.
51428		
51429		<code>WSTOPSIG(stat_val)</code>
51430		If the value of <code>WIFSTOPPED(stat_val)</code> is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.
51431		
51432	XSI	<code>WIFCONTINUED(stat_val)</code>
51433		Evaluates to a non-zero value if <code>status</code> was returned for a child process that has continued from a job control stop.
51434		
51435	SPN	It is unspecified whether the <code>status</code> value returned by calls to <code>wait()</code> or <code>waitpid()</code> for processes created by <code>posix_spawn()</code> or <code>posix_spawnnp()</code> can indicate a <code>WIFSTOPPED(stat_val)</code> before subsequent calls to <code>wait()</code> or <code>waitpid()</code> indicate <code>WIFEXITED(stat_val)</code> as the result of an error detected before the new process image starts executing.
51436		
51437		
51438		
51439		It is unspecified whether the <code>status</code> value returned by calls to <code>wait()</code> or <code>waitpid()</code> for processes created by <code>posix_spawn()</code> or <code>posix_spawnnp()</code> can indicate a <code>WIFSIGNALED(stat_val)</code> if a signal is sent to the parent's process group after <code>posix_spawn()</code> or <code>posix_spawnnp()</code> is called.
51440		
51441		
51442		If the information pointed to by <code>stat_loc</code> was stored by a call to <code>waitpid()</code> that specified the <code>WUNTRACED</code> flag and did not specify the <code>WCONTINUED</code> flag, exactly one of the macros <code>WIFEXITED(*stat_loc)</code> , <code>WIFSIGNALED(*stat_loc)</code> , and <code>WIFSTOPPED(*stat_loc)</code> shall evaluate to a non-zero value.
51443	XSI	
51444		
51445		
51446		If the information pointed to by <code>stat_loc</code> was stored by a call to <code>waitpid()</code> that specified the

wait()

51447 XSI WUNTRACED and WCONTINUED flags, exactly one of the macros WIFEXITED(*stat_loc),
 51448 XSI WIFSIGNALED(*stat_loc), WIFSTOPPED(*stat_loc), and WIFCONTINUED(*stat_loc) shall
 51449 evaluate to a non-zero value.

51450 If the information pointed to by *stat_loc* was stored by a call to *waitpid()* that did not specify the
 51451 XSI WUNTRACED or WCONTINUED flags, or by a call to the *wait()* function, exactly one of the
 51452 macros WIFEXITED(*stat_loc) and WIFSIGNALED(*stat_loc) shall evaluate to a non-zero value.

51453 If the information pointed to by *stat_loc* was stored by a call to *waitpid()* that did not specify the
 51454 XSI WUNTRACED flag and specified the WCONTINUED flag, or by a call to the *wait()* function,
 51455 XSI exactly one of the macros WIFEXITED(*stat_loc), WIFSIGNALED(*stat_loc), and
 51456 WIFCONTINUED(*stat_loc) shall evaluate to a non-zero value.

51457 If `_POSIX_REALTIME_SIGNALS` is defined, and the implementation queues the SIGCHLD
 51458 signal, then if *wait()* or *waitpid()* returns because the status of a child process is available, any
 51459 pending SIGCHLD signal associated with the process ID of the child process shall be discarded.
 51460 Any other pending SIGCHLD signals shall remain pending.

51461 Otherwise, if SIGCHLD is blocked, if *wait()* or *waitpid()* return because the status of a child
 51462 process is available, any pending SIGCHLD signal shall be cleared unless the status of another
 51463 child process is available.

51464 For all other conditions, it is unspecified whether child *status* will be available when a SIGCHLD
 51465 signal is delivered.

51466 There may be additional implementation-defined circumstances under which *wait()* or *waitpid()*
 51467 report *status*. This shall not occur unless the calling process or one of its child processes
 51468 explicitly makes use of a non-standard extension. In these cases the interpretation of the
 51469 reported *status* is implementation-defined.

51470 XSI If a parent process terminates without waiting for all of its child processes to terminate, the
 51471 remaining child processes shall be assigned a new parent process ID corresponding to an
 51472 implementation-defined system process.

RETURN VALUE

51473 If *wait()* or *waitpid()* returns because the status of a child process is available, these functions
 51474 shall return a value equal to the process ID of the child process for which *status* is reported. If
 51475 *wait()* or *waitpid()* returns due to the delivery of a signal to the calling process, `-1` shall be
 51476 returned and *errno* set to `[EINTR]`. If *waitpid()* was invoked with `WNOHANG` set in *options*, it
 51477 has at least one child process specified by *pid* for which *status* is not available, and *status* is not
 51478 available for any process specified by *pid*, `0` is returned. Otherwise, `(pid_t)-1` shall be returned,
 51479 and *errno* set to indicate the error.
 51480

ERRORS

51481 The *wait()* function shall fail if:

51482 [ECHILD] The calling process has no existing unwaited-for child processes.

51483 [EINTR] The function was interrupted by a signal. The value of the location pointed to
 51484 by *stat_loc* is undefined.

51485 The *waitpid()* function shall fail if:

51486 [ECHILD] The process specified by *pid* does not exist or is not a child of the calling
 51487 process, or the process group specified by *pid* does not exist or does not have
 51488 any member process that is a child of the calling process.

51489 [EINTR] The function was interrupted by a signal. The value of the location pointed to
 51490 by *stat_loc* is undefined.
 51491

51492 [EINVAL] The *options* argument is not valid.

51493 EXAMPLES

51494 Waiting for a Child Process and then Checking its Status

51495 The following example demonstrates the use of *waitpid()*, *fork()*, and the macros used to
 51496 interpret the status value returned by *waitpid()* (and *wait()*). The code segment creates a child
 51497 process which does some unspecified work. Meanwhile the parent loops performing calls to
 51498 *waitpid()* to monitor the status of the child. The loop terminates when child termination is
 51499 detected.

```

51500 #include <stdio.h>
51501 #include <stdlib.h>
51502 #include <unistd.h>
51503 #include <sys/wait.h>
51504 ...
51505 pid_t child_pid, wpid;
51506 int status;
51507
51508 child_pid = fork();
51509 if (child_pid == -1) {           /* fork() failed */
51510     perror("fork");
51511     exit(EXIT_FAILURE);
51512 }
51513 if (child_pid == 0) {           /* This is the child */
51514     /* Child does some work and then terminates */
51515     ...
51516 } else {                         /* This is the parent */
51517     do {
51518         wpid = waitpid(child_pid, &status, WUNTRACED
51519 #ifdef WCONTINUED           /* Not all implementations support this */
51520 | WCONTINUED
51521 #endif
51522 );
51523 if (wpid == -1) {
51524     perror("waitpid");
51525     exit(EXIT_FAILURE);
51526 }
51527 if (WIFEXITED(status)) {
51528     printf("child exited, status=%d\n", WEXITSTATUS(status));
51529 } else if (WIFSIGNALED(status)) {
51530     printf("child killed (signal %d)\n", WTERMSIG(status));
51531 } else if (WIFSTOPPED(status)) {
51532     printf("child stopped (signal %d)\n", WSTOPSIG(status));
51533 #ifdef WIFCONTINUED           /* Not all implementations support this */
51534 } else if (WIFCONTINUED(status)) {
51535     printf("child continued\n");
51536 #endif
51537 } else { /* Non-standard case -- may never happen */
51538     printf("Unexpected status (0x%x)\n", status);
51539 }
51540 } while (!WIFEXITED(status) && !WIFSIGNALED(status));

```

51540 }

51541 APPLICATION USAGE

51542 None.

51543 RATIONALE

51544 A call to the *wait()* or *waitpid()* function only returns *status* on an immediate child process of the
 51545 calling process; that is, a child that was produced by a single *fork()* call (perhaps followed by an
 51546 *exec* or other function calls) from the parent. If a child produces grandchildren by further use of
 51547 *fork()*, none of those grandchildren nor any of their descendants affect the behavior of a *wait()*
 51548 from the original parent process. Nothing in this volume of IEEE Std 1003.1-200x prevents an
 51549 implementation from providing extensions that permit a process to get *status* from a grandchild
 51550 or any other process, but a process that does not use such extensions must be guaranteed to see
 51551 *status* from only its direct children.

51552 The *waitpid()* function is provided for three reasons:

- 51553 1. To support job control
- 51554 2. To permit a non-blocking version of the *wait()* function
- 51555 3. To permit a library routine, such as *system()* or *pclose()*, to wait for its children without
 51556 interfering with other terminated children for which the process has not waited

51557 The first two of these facilities are based on the *wait3()* function provided by 4.3 BSD. The
 51558 function uses the *options* argument, which is equivalent to an argument to *wait3()*. The
 51559 WUNTRACED flag is used only in conjunction with job control on systems supporting job
 51560 control. Its name comes from 4.3 BSD and refers to the fact that there are two types of stopped
 51561 processes in that implementation: processes being traced via the *ptrace()* debugging facility and
 51562 (untraced) processes stopped by job control signals. Since *ptrace()* is not part of this volume of
 51563 IEEE Std 1003.1-200x, only the second type is relevant. The name WUNTRACED was retained
 51564 because its usage is the same, even though the name is not intuitively meaningful in this context.

51565 The third reason for the *waitpid()* function is to permit independent sections of a process to
 51566 spawn and wait for children without interfering with each other. For example, the following
 51567 problem occurs in developing a portable shell, or command interpreter:

```
51568 stream = popen("/bin/true");
51569 (void) system("sleep 100");
51570 (void) pclose(stream);
```

51571 On all historical implementations, the final *pclose()* fails to reap the *wait()* *status* of the *popen()*.

51572 The status values are retrieved by macros, rather than given as specific bit encodings as they are
 51573 in most historical implementations (and thus expected by existing programs). This was
 51574 necessary to eliminate a limitation on the number of signals an implementation can support that
 51575 was inherent in the traditional encodings. This volume of IEEE Std 1003.1-200x does require that
 51576 a *status* value of zero corresponds to a process calling *_exit(0)*, as this is the most common
 51577 encoding expected by existing programs. Some of the macro names were adopted from 4.3 BSD.

51578 These macros syntactically operate on an arbitrary integer value. The behavior is undefined
 51579 unless that value is one stored by a successful call to *wait()* or *waitpid()* in the location pointed to
 51580 by the *stat_loc* argument. An early proposal attempted to make this clearer by specifying each
 51581 argument as **stat_loc* rather than *stat_val*. However, that did not follow the conventions of other
 51582 specifications in this volume of IEEE Std 1003.1-200x or traditional usage. It also could have
 51583 implied that the argument to the macro must literally be **stat_loc*; in fact, that value can be
 51584 stored or passed as an argument to other functions before being interpreted by these macros.

51585 The extension that affects *wait()* and *waitpid()* and is common in historical implementations is
 51586 the *ptrace()* function. It is called by a child process and causes that child to stop and return a
 51587 *status* that appears identical to the *status* indicated by WIFSTOPPED. The *status* of *ptrace()*

51588 children is traditionally returned regardless of the WUNTRACED flag (or by the *wait()*
 51589 function). Most applications do not need to concern themselves with such extensions because
 51590 they have control over what extensions they or their children use. However, applications, such
 51591 as command interpreters, that invoke arbitrary processes may see this behavior when those
 51592 arbitrary processes misuse such extensions.

51593 Implementations that support **core** file creation or other implementation-defined actions on
 51594 termination of some processes traditionally provide a bit in the *status* returned by *wait()* to
 51595 indicate that such actions have occurred.

51596 Allowing the *wait()* family of functions to discard a pending SIGCHLD signal that is associated
 51597 with a successfully waited-for child process puts them into the *sigwait()* and *sigwaitinfo()*
 51598 category with respect to SIGCHLD.

51599 This definition allows implementations to treat a pending SIGCHLD signal as accepted by the
 51600 process in *wait()*, with the same meaning of “accepted” as when that word is applied to the
 51601 *sigwait()* family of functions.

51602 Allowing the *wait()* family of functions to behave this way permits an implementation to be able
 51603 to deal precisely with SIGCHLD signals.

51604 In particular, an implementation that does accept (discard) the SIGCHLD signal can make the
 51605 following guarantees regardless of the queuing depth of signals in general (the list of waitable
 51606 children can hold the SIGCHLD queue):

- 51607 1. If a SIGCHLD signal handler is established via *sigaction()* without the SA_RESETHAND
 51608 flag, SIGCHLD signals can be accurately counted; that is, exactly one SIGCHLD signal
 51609 will be delivered to or accepted by the process for every child process that terminates.
- 51610 2. A single *wait()* issued from a SIGCHLD signal handler can be guaranteed to return
 51611 immediately with status information for a child process.
- 51612 3. When SA_SIGINFO is requested, the SIGCHLD signal handler can be guaranteed to
 51613 receive a non-NULL pointer to a **siginfo_t** structure that describes a child process for
 51614 which a wait via *waitpid()* or *waitid()* will not block or fail.
- 51615 4. The *system()* function will not cause the SIGCHLD handler of a process to be called as a
 51616 result of the *fork()/exec* executed within *system()* because *system()* will accept the
 51617 SIGCHLD signal when it performs a *waitpid()* for its child process. This is a desirable
 51618 behavior of *system()* so that it can be used in a library without causing side effects to the
 51619 application linked with the library.

51620 An implementation that does not permit the *wait()* family of functions to accept (discard) a
 51621 pending SIGCHLD signal associated with a successfully waited-for child, cannot make the
 51622 guarantees described above for the following reasons:

51623 Guarantee #1

51624 Although it might be assumed that reliable queuing of all SIGCHLD signals generated by
 51625 the system can make this guarantee, the counter-example is the case of a process that blocks
 51626 SIGCHLD and performs an indefinite loop of *fork()/wait()* operations. If the
 51627 implementation supports queued signals, then eventually the system will run out of
 51628 memory for the queue. The guarantee cannot be made because there must be some limit to
 51629 the depth of queuing.

51630 Guarantees #2 and #3

51631 These cannot be guaranteed unless the *wait()* family of functions accepts the SIGCHLD
 51632 signal. Otherwise, a *fork()/wait()* executed while SIGCHLD is blocked (as in the *system()*
 51633 function) will result in an invocation of the handler when SIGCHLD is unblocked, after the
 51634 process has disappeared.

51635 Guarantee #4
51636 Although possible to make this guarantee, *system()* would have to set the SIGCHLD
51637 handler to SIG_DFL so that the SIGCHLD signal generated by its *fork()* would be discarded
51638 (the SIGCHLD default action is to be ignored), then restore it to its previous setting. This
51639 would have the undesirable side effect of discarding all SIGCHLD signals pending to the
51640 process.

51641 **FUTURE DIRECTIONS**
51642 None.

51643 **SEE ALSO**
51644 *exec*, *exit()*, *fork()*, *waitid()*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10,
51645 Memory Synchronization, `<signal.h>`, `<sys/wait.h>`

51646 **CHANGE HISTORY**
51647 First released in Issue 1. Derived from Issue 1 of the SVID.

51648 **Issue 5**
51649 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

51650 **Issue 6**
51651 The following new requirements on POSIX implementations derive from alignment with the
51652 Single UNIX Specification:
51653

- The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
51654 required for conforming implementations of previous POSIX specifications, it was not
51655 required for UNIX applications.

51656 The following changes were made to align with the IEEE P1003.1a draft standard:
51657

- The processing of the SIGCHLD signal and the [ECHILD] error is clarified.

51658 The semantics of *WIFSTOPPED(stat_val)*, *WIFEXITED(stat_val)*, and *WIFSIGNALED(stat_val)*
51659 are defined with respect to *posix_spawn()* or *posix_spawnnp()* for alignment with IEEE Std
51660 1003.1d-1999.
51661 The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.
51662 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/145 is applied, adding the example to the
51663 EXAMPLES section.

51664 **NAME**
 51665 waitid — wait for a child process to change state

51666 **SYNOPSIS**
 51667 #include <sys/wait.h>

51668 int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);

51669 **DESCRIPTION**

51670 The *waitid()* function shall suspend the calling thread until one child of the process containing
 51671 the calling thread changes state. It records the current state of a child in the structure pointed to
 51672 by *infop*. The fields of the structure pointed to by *infop* are filled in as described for the
 51673 SIGCHLD signal in <signal.h>. If a child process changed state prior to the call to *waitid()*,
 51674 *waitid()* shall return immediately. If more than one thread is suspended in *wait()* or *waitpid()*
 51675 waiting for termination of the same process, exactly one thread shall return the process status at
 51676 the time of the target process termination.

51677 The *idtype* and *id* arguments are used to specify which children *waitid()* waits for.

51678 If *idtype* is P_PID, *waitid()* shall wait for the child with a process ID equal to (**pid_t**)*id*.

51679 If *idtype* is P_PGID, *waitid()* shall wait for any child with a process group ID equal to (**pid_t**)*id*.

51680 If *idtype* is P_ALL, *waitid()* shall wait for any children and *id* is ignored.

51681 The *options* argument is used to specify which state changes *waitid()* shall wait for. It is formed
 51682 by OR'ing together the following flags:

51683 WEXITED Wait for processes that have exited.

51684 WSTOPPED Status shall be returned for any child that has stopped upon receipt of a signal.

51685 WCONTINUED Status shall be returned for any child that was stopped and has been
 51686 continued.

51687 WNOHANG Return immediately if there are no children to wait for.

51688 WNOWAIT Keep the process whose status is returned in *infop* in a waitable state. This
 51689 shall not affect the state of the process; the process may be waited for again
 51690 after this call completes.

51691 Applications shall specify at least one of the flags WEXITED, WSTOPPED, or WCONTINUED to
 51692 be OR'd in with the *options* argument.

51693 The application shall ensure that the *infop* argument points to a **siginfo_t** structure. If *waitid()*
 51694 returns because a child process was found that satisfied the conditions indicated by the
 51695 arguments *idtype* and *options*, then the structure pointed to by *infop* shall be filled in by the
 51696 system with the status of the process. The *si_signo* member shall always be equal to SIGCHLD.

51697 **RETURN VALUE**

51698 If WNOHANG was specified and there are no children to wait for, 0 shall be returned. If *waitid()*
 51699 returns due to the change of state of one of its children, 0 shall be returned. Otherwise, -1 shall
 51700 be returned and *errno* set to indicate the error.

51701 **ERRORS**

51702 The *waitid()* function shall fail if:

51703 [ECHILD] The calling process has no existing unwaited-for child processes.

waitid()

51704 [EINTR] The *waitid()* function was interrupted by a signal.

51705 [EINVAL] An invalid value was specified for *options*, or *idtype* and *id* specify an invalid

51706 set of processes.

EXAMPLES

51707 None.

51708

APPLICATION USAGE

51709 None.

51710

RATIONALE

51711 None.

51712

FUTURE DIRECTIONS

51713 None.

51714

SEE ALSO

51715 *exec*, *exit()*, *wait()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**,

51716 **<sys/wait.h>**

51717

CHANGE HISTORY

51718 First released in Issue 4, Version 2.

51719

Issue 5

51720 Moved from X/OPEN UNIX extension to BASE.

51721

51722 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

51723 The normative text is updated to avoid use of the term “must” for application requirements.

51724

Issue 7

51725 Austin Group Interpretation 1003.1-2001 #060 is applied, updating the DESCRIPTION.

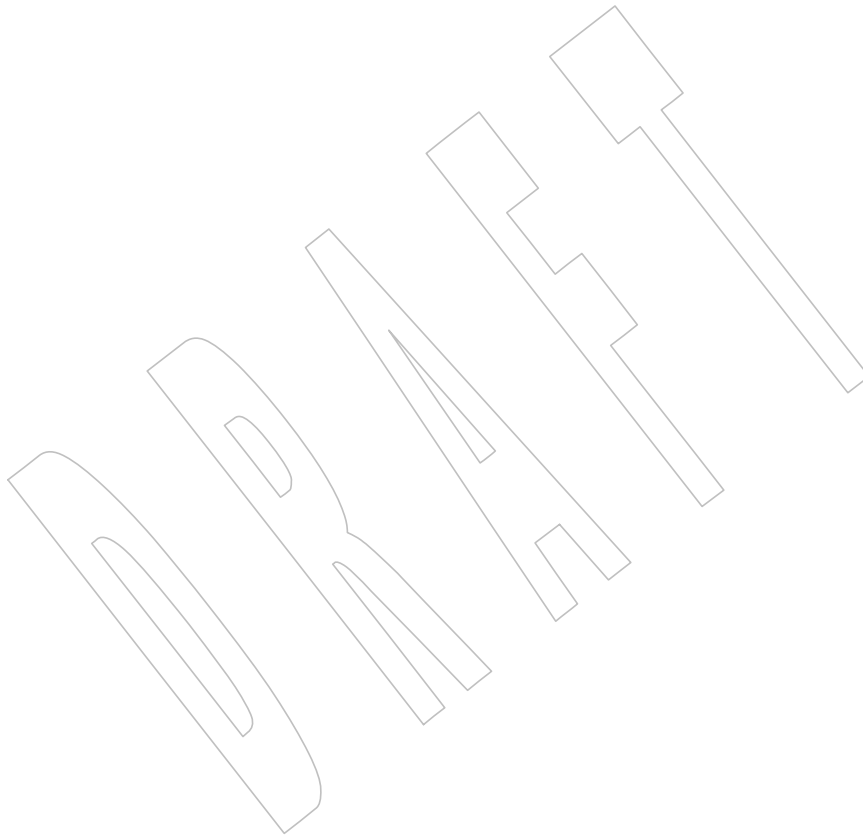
51726

51727 The *waitid()* function is moved from the XSI option to the Base.

51728 **NAME**
51729 `waitpid` — wait for a child process to stop or terminate

51730 **SYNOPSIS**
51731 `#include <sys/wait.h>`
51732 `pid_t waitpid(pid_t pid, int *stat_loc, int options);`

51733 **DESCRIPTION**
51734 Refer to *wait()*.



51735 **NAME**
51736 `wcpcpy` — copy a wide-character string, returning a pointer to its end

51737 **SYNOPSIS**

51738 CX `#include <wchar.h>`
51739 `wchar_t *wcpcpy(wchar_t *restrict ws1, const wchar_t *restrict ws2);`

51740 **DESCRIPTION**

51741 Refer to *wcscopy()*.

51742 **NAME**51743 `wcpncpy` — copy a fixed-size wide-character string, returning a pointer to its end51744 **SYNOPSIS**

```
51745 CX #include <wchar.h>  
51746     wchar_t *wcpncpy(wchar_t restrict *ws1, const wchar_t *restrict ws2,  
51747                     size_t n);
```

51748 **DESCRIPTION**51749 Refer to *wcsncpy()*.

51750 **NAME**51751 `wcrtomb` — convert a wide-character code to a character (restartable)51752 **SYNOPSIS**51753 `#include <stdio.h>`51754 `size_t wcrtomb(char *restrict s, wchar_t wc, mbstate_t *restrict ps);`51755 **DESCRIPTION**51756 CX The functionality described on this reference page is aligned with the ISO C standard. Any
51757 conflict between the requirements described here and the ISO C standard is unintentional. This
51758 volume of IEEE Std 1003.1-200x defers to the ISO C standard.51759 If *s* is a null pointer, the `wcrtomb()` function shall be equivalent to the call:51760 `wcrtomb(buf, L'\0', ps)`51761 where *buf* is an internal buffer.51762 If *s* is not a null pointer, the `wcrtomb()` function shall determine the number of bytes needed to
51763 represent the character that corresponds to the wide character given by *wc* (including any shift
51764 sequences), and store the resulting bytes in the array whose first element is pointed to by *s*. At
51765 most {MB_CUR_MAX} bytes are stored. If *wc* is a null wide character, a null byte shall be stored,
51766 preceded by any shift sequence needed to restore the initial shift state. The resulting state
51767 described shall be the initial conversion state.51768 If *ps* is a null pointer, the `wcrtomb()` function shall use its own internal **mbstate_t** object, which is
51769 initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object
51770 pointed to by *ps* shall be used to completely describe the current conversion state of the
51771 associated character sequence. The implementation shall behave as if no function defined in this
51772 volume of IEEE Std 1003.1-200x calls `wcrtomb()`.51773 CX If the application uses any of the `_POSIX_THREAD_SAFE_FUNCTIONS` or `_POSIX_THREADS`
51774 functions, the application shall ensure that the `wcrtomb()` function is called with a non-NULL *ps*
51775 argument.51776 The behavior of this function shall be affected by the `LC_CTYPE` category of the current locale.51777 **RETURN VALUE**51778 The `wcrtomb()` function shall return the number of bytes stored in the array object (including any
51779 shift sequences). When *wc* is not a valid wide character, an encoding error shall occur. In this
51780 case, the function shall store the value of the macro [EILSEQ] in *errno* and shall return (**size_t**)-1;
51781 the conversion state shall be undefined.51782 **ERRORS**51783 The `wcrtomb()` function may fail if:51784 CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

51785 [EILSEQ] Invalid wide-character code is detected.

51786

EXAMPLES

51787

None.

51788

APPLICATION USAGE

51789

None.

51790

RATIONALE

51791

None.

51792

FUTURE DIRECTIONS

51793

None.

51794

SEE ALSO

51795

mbsinit(), *wcsrombs()*, the Base Definitions volume of IEEE Std 1003.1-200x, <wchar.h>

51796

CHANGE HISTORY

51797

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

51798

51799

Issue 6

51800

In the DESCRIPTION, a note on using this function in a threaded application is added.

51801

Extensions beyond the ISO C standard are marked.

51802

The normative text is updated to avoid use of the term “must” for application requirements.

51803

The *wcr tomb()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

DRAFT

51804 **NAME**

51805 `wscasecmp`, `wscasecmp_l`, `wcsncasecmp`, `wcsncasecmp_l` — case-insensitive wide-character
 51806 string comparison

51807 **SYNOPSIS**

```
51808 CX #include <wchar.h>
51809
51809 int wscasecmp(const wchar_t *ws1, const wchar_t *ws2);
51810 int wscasecmp_l(const char *ws1, const char *ws2,
51811               locale_t locale);
51812 int wcsncasecmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
51813 int wcsncasecmp_l(const char *ws1, const char *ws2,
51814                  size_t n, locale_t locale);
```

51815 **DESCRIPTION**

51816 The `wscasecmp()` and `wcsncasecmp()` functions are the wide-character equivalent of the
 51817 `strcasecmp()` and `strncasecmp()` functions, respectively.

51818 The `wscasecmp()` and `wscasecmp_l()` functions shall compare, while ignoring differences in case,
 51819 the wide-character string pointed to by `ws1` to the wide-character string pointed to by `ws2`.

51820 The `wcsncasecmp()` and `wcsncasecmp_l()` functions shall compare, while ignoring differences in
 51821 case, not more than `n` wide-characters from the wide-character string pointed to by `ws1` to the
 51822 wide-character string pointed to by `ws2`.

51823 When the `LC_CTIME` category of the current locale is from the POSIX locale, these functions
 51824 shall behave as if the strings had been converted to lowercase and then a byte comparison
 51825 performed. Otherwise, the results are unspecified.

51826 The information for `wscasecmp_l()` and `wcsncasecmp_l()` about the case of the characters comes
 51827 from the locale represented by `locale`.

51828 **RETURN VALUE**

51829 Upon completion, the `wscasecmp()` and `wscasecmp_l()` functions shall return an integer greater
 51830 than, equal to, or less than 0 if the wide-character string pointed to by `ws1` is, ignoring case,
 51831 greater than, equal to, or less than the wide-character string pointed to by `ws2`, respectively.

51832 Upon completion, the `wcsncasecmp()` and `wcsncasecmp_l()` functions shall return an integer
 51833 greater than, equal to, or less than 0 if the possibly null wide-character terminated string pointed
 51834 to by `ws1` is, ignoring case, greater than, equal to, or less than the possibly null wide-character
 51835 terminated string pointed to by `ws2`, respectively.

51836 No return values are reserved to indicate an error.

51837 **ERRORS**

51838 The `wscasecmp_l()` and `wcsncasecmp_l()` functions may fail if:

51839 [EINVAL] `locale` is not a valid locale object handle.

51840

EXAMPLES

51841

None.

51842

APPLICATION USAGE

51843

None.

51844

RATIONALE

51845

None.

51846

FUTURE DIRECTIONS

51847

None.

51848

SEE ALSO

51849

strcasecmp(), *wscmp()*, *wcsncmp()*, the Base Definitions volume of IEEE Std 1003.1-200x,

51850

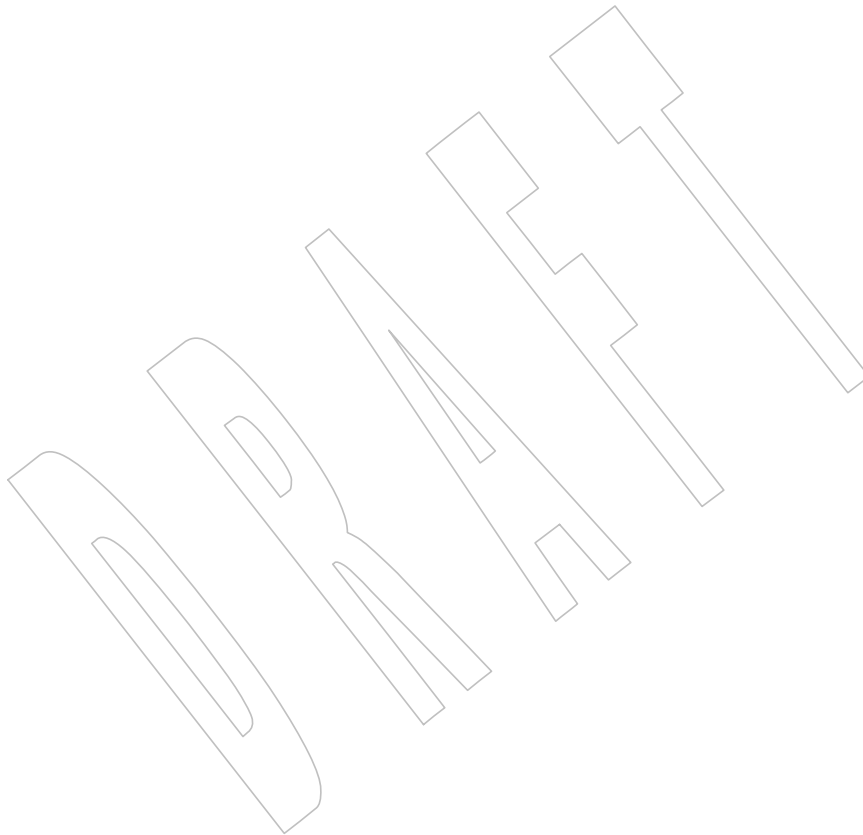
<wchar.h>

51851

CHANGE HISTORY

51852

First released in Issue 7.



51853 **NAME**51854 `wcscat` — concatenate two wide-character strings51855 **SYNOPSIS**51856 `#include <wchar.h>`51857 `wchar_t *wcscat(wchar_t *restrict ws1, const wchar_t *restrict ws2);`51858 **DESCRIPTION**51859 CX The functionality described on this reference page is aligned with the ISO C standard. Any
51860 conflict between the requirements described here and the ISO C standard is unintentional. This
51861 volume of IEEE Std 1003.1-200x defers to the ISO C standard.51862 The `wcscat()` function shall append a copy of the wide-character string pointed to by `ws2`
51863 (including the terminating null wide-character code) to the end of the wide-character string
51864 pointed to by `ws1`. The initial wide-character code of `ws2` shall overwrite the null wide-character
51865 code at the end of `ws1`. If copying takes place between objects that overlap, the behavior is
51866 undefined.51867 **RETURN VALUE**51868 The `wcscat()` function shall return `ws1`; no return value is reserved to indicate an error.51869 **ERRORS**

51870 No errors are defined.

51871 **EXAMPLES**

51872 None.

51873 **APPLICATION USAGE**

51874 None.

51875 **RATIONALE**

51876 None.

51877 **FUTURE DIRECTIONS**

51878 None.

51879 **SEE ALSO**51880 `wcsncat()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<wchar.h>`51881 **CHANGE HISTORY**

51882 First released in Issue 4. Derived from the MSE working draft.

51883 **Issue 6**51884 The Open Group Corrigendum U040/2 is applied. In the RETURN VALUE section, `s1` is
51885 changed to `ws1`.51886 The `wcscat()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

51887 **NAME**
 51888 `wcschr` — wide-character string scanning operation

51889 **SYNOPSIS**
 51890 `#include <wchar.h>`

51891 `wchar_t *wcschr(const wchar_t *ws, wchar_t wc);`

51892 **DESCRIPTION**

51893 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 51894 conflict between the requirements described here and the ISO C standard is unintentional. This
 51895 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

51896 The `wcschr()` function shall locate the first occurrence of `wc` in the wide-character string pointed
 51897 to by `ws`. The application shall ensure that the value of `wc` is a character representable as a type
 51898 `wchar_t` and a wide-character code corresponding to a valid character in the current locale. The
 51899 terminating null wide-character code is considered to be part of the wide-character string.

51900 **RETURN VALUE**

51901 Upon completion, `wcschr()` shall return a pointer to the wide-character code, or a null pointer if
 51902 the wide-character code is not found.

51903 **ERRORS**

51904 No errors are defined.

51905 **EXAMPLES**

51906 None.

51907 **APPLICATION USAGE**

51908 None.

51909 **RATIONALE**

51910 None.

51911 **FUTURE DIRECTIONS**

51912 None.

51913 **SEE ALSO**

51914 `wcsrchr()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<wchar.h>`

51915 **CHANGE HISTORY**

51916 First released in Issue 4. Derived from the MSE working draft.

51917 **Issue 6**

51918 The normative text is updated to avoid use of the term “must” for application requirements.

51919 **NAME**51920 `wcscmp` — compare two wide-character strings51921 **SYNOPSIS**51922 `#include <wchar.h>`51923 `int wcscmp(const wchar_t *ws1, const wchar_t *ws2);`51924 **DESCRIPTION**51925 CX The functionality described on this reference page is aligned with the ISO C standard. Any
51926 conflict between the requirements described here and the ISO C standard is unintentional. This
51927 volume of IEEE Std 1003.1-200x defers to the ISO C standard.51928 The `wcscmp()` function shall compare the wide-character string pointed to by `ws1` to the wide-
51929 character string pointed to by `ws2`.51930 The sign of a non-zero return value shall be determined by the sign of the difference between the
51931 values of the first pair of wide-character codes that differ in the objects being compared.51932 **RETURN VALUE**51933 Upon completion, `wcscmp()` shall return an integer greater than, equal to, or less than 0, if the
51934 wide-character string pointed to by `ws1` is greater than, equal to, or less than the wide-character
51935 string pointed to by `ws2`, respectively.51936 **ERRORS**

51937 No errors are defined.

51938 **EXAMPLES**

51939 None.

51940 **APPLICATION USAGE**

51941 None.

51942 **RATIONALE**

51943 None.

51944 **FUTURE DIRECTIONS**

51945 None.

51946 **SEE ALSO**51947 [*wcscasecmp\(\)*](#), [*wcsncmp\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<wchar.h>`51948 **CHANGE HISTORY**

51949 First released in Issue 4. Derived from the MSE working draft.

51950 **NAME**
 51951 `wscoll`, `wscoll_l` — wide-character string comparison using collating information

51952 **SYNOPSIS**

51953 `#include <wchar.h>`
 51954 `int wscoll(const wchar_t *ws1, const wchar_t *ws2);`
 51955 CX `int wscoll_l(const wchar_t *ws1, const wchar_t *ws2,`
 51956 `locale_t locale);`

51957 **DESCRIPTION**

51958 CX For `wscoll()`: The functionality described on this reference page is aligned with the ISO C
 51959 standard. Any conflict between the requirements described here and the ISO C standard is
 51960 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

51961 CX The `wscoll()` and `wscoll_l()` functions shall compare the wide-character string pointed to by
 51962 `ws1` to the wide-character string pointed to by `ws2`, both interpreted as appropriate to the
 51963 CX `LC_COLLATE` category of the current locale of the process, or the locale represented by `locale`,
 51964 respectively.

51965 CX The `wscoll()` and `wscoll_l()` functions shall not change the setting of `errno` if successful.

51966 CX An application wishing to check for error situations should set `errno` to 0 before calling `wscoll()`
 51967 or `wscoll_l()`. If `errno` is non-zero on return, an error has occurred.

51968 **RETURN VALUE**

51969 CX Upon successful completion, `wscoll()` and `wscoll_l()` shall return an integer greater than, equal
 51970 to, or less than 0, according to whether the wide-character string pointed to by `ws1` is greater
 51971 than, equal to, or less than the wide-character string pointed to by `ws2`, when both are
 51972 CX interpreted as appropriate to the current locale, or to the locale represented by `locale`,
 51973 CX respectively. On error, `wscoll()` and `wscoll_l()` shall set `errno`, but no return value is reserved
 51974 to indicate an error.

51975 **ERRORS**

51976 These functions may fail if:

51977 CX [EINVAL] The `ws1` or `ws2` arguments contain wide-character codes outside the domain of
 51978 the collating sequence.

51979 The `wscoll_l()` function may fail if:

51980 CX [EINVAL] `locale` is not a valid locale object handle.

51981 **EXAMPLES**

51982 None.

51983 **APPLICATION USAGE**

51984 The `wcsxfrm()` and `wscmp()` functions should be used for sorting large lists.

51985 **RATIONALE**

51986 None.

51987 **FUTURE DIRECTIONS**

51988 None.

51989

SEE ALSO

51990

wscmp(), *wcsxfrm()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

51991

CHANGE HISTORY

51992

First released in Issue 4. Derived from the MSE working draft.

51993

Issue 5

51994

Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

51995

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

51996

Issue 7

51997

The *wscoll_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

51998



51999 **NAME**52000 `wcpcpy`, `wcscopy` — copy a wide-character string, returning a pointer to its end52001 **SYNOPSIS**52002 `#include <wchar.h>`52003 CX `wchar_t *wcpcpy(wchar_t *restrict ws1, const wchar_t *restrict ws2);`52004 `wchar_t *wcscopy(wchar_t *restrict ws1, const wchar_t *restrict ws2);`52005 **DESCRIPTION**52006 CX For `wcscopy()`: The functionality described on this reference page is aligned with the ISO C
52007 standard. Any conflict between the requirements described here and the ISO C standard is
52008 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.52009 CX The `wcpcpy()` and `wcscopy()` functions shall copy the wide-character string pointed to by `ws2`
52010 (including the terminating null wide-character code) into the array pointed to by `ws1`.52011 The application shall ensure that there is room for at least `wcslen(ws2)+1` wide characters in the
52012 `ws1` array, and that the `ws2` and `ws1` arrays do not overlap.

52013 If copying takes place between objects that overlap, the behavior is undefined.

52014 **RETURN VALUE**52015 CX The `wcpcpy()` function shall return a pointer to the terminating null wide-character code copied
52016 into the `ws1` buffer.52017 The `wcscopy()` function shall return `ws1`.

52018 No return values are reserved to indicate an error.

52019 **ERRORS**

52020 No errors are defined.

52021 **EXAMPLES**

52022 None.

52023 **APPLICATION USAGE**

52024 None.

52025 **RATIONALE**

52026 None.

52027 **FUTURE DIRECTIONS**

52028 None.

52029 **SEE ALSO**52030 [*strcpy\(\)*](#), [*wcsdup\(\)*](#), [*wcsncpy\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<wchar.h>`52031 **CHANGE HISTORY**

52032 First released in Issue 4. Derived from the MSE working draft.

52033 **Issue 6**52034 The `wcscopy()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.52035 **Issue 7**52036 The `wcpcpy()` function is added from The Open Group Technical Standard, 2006, Extended API
52037 Set Part 1.

52038 **NAME**
 52039 `wcscspn` — get the length of a complementary wide substring

52040 **SYNOPSIS**
 52041 `#include <wchar.h>`
 52042 `size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);`

52043 **DESCRIPTION**
 52044 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 52045 conflict between the requirements described here and the ISO C standard is unintentional. This
 52046 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52047 The `wcscspn()` function shall compute the length (in wide characters) of the maximum initial
 52048 segment of the wide-character string pointed to by `ws1` which consists entirely of wide-character
 52049 codes *not* from the wide-character string pointed to by `ws2`.

52050 **RETURN VALUE**
 52051 The `wcscspn()` function shall return the length of the initial substring of `ws1`; no return value is
 52052 reserved to indicate an error.

52053 **ERRORS**
 52054 No errors are defined.

52055 **EXAMPLES**
 52056 None.

52057 **APPLICATION USAGE**
 52058 None.

52059 **RATIONALE**
 52060 None.

52061 **FUTURE DIRECTIONS**
 52062 None.

52063 **SEE ALSO**
 52064 [*wcsspn\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<wchar.h>`

52065 **CHANGE HISTORY**
 52066 First released in Issue 4. Derived from the MSE working draft.

52067 **Issue 5**
 52068 The RETURN VALUE section is updated to indicate that `wcscspn()` returns the length of `ws1`,
 52069 rather than `ws1` itself.

52070 **NAME**
 52071 wcsdup — duplicate a wide-character string

52072 **SYNOPSIS**
 52073 CX #include <wchar.h>
 52074 wchar_t *wcsdup(const wchar_t *string);

52075 **DESCRIPTION**
 52076 The *wcsdup()* function is the wide-character equivalent of the *strdup()* function.

52077 The *wcsdup()* function shall return a pointer to a new wide-character string, which is the
 52078 duplicate of the wide-character string *string*. The returned pointer can be passed to *free()*. A
 52079 null pointer is returned if the new wide-character string cannot be created.

52080 **RETURN VALUE**
 52081 Upon successful completion, the *wcsdup()* function shall return a pointer to the newly allocated
 52082 wide-character string. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

52083 **ERRORS**
 52084 The *wcsdup()* function shall fail if:
 52085 [ENOMEM] Memory large enough for the duplicate string could not be allocated.

52086 **EXAMPLES**
 52087 None.

52088 **APPLICATION USAGE**
 52089 None.

52090 **RATIONALE**
 52091 None.

52092 **FUTURE DIRECTIONS**
 52093 None.

52094 **SEE ALSO**
 52095 *free()*, *strdup()*, *wscpy()*, the Base Definitions volume of IEEE Std 1003.1-200x, <wchar.h>

52096 **CHANGE HISTORY**
 52097 First released in Issue 7.

52098 **NAME**52099 `wcsftime` — convert date and time to a wide-character string52100 **SYNOPSIS**52101 `#include <wchar.h>`52102 `size_t wcsftime(wchar_t *restrict wcs, size_t maxsize,`
52103 `const wchar_t *restrict format, const struct tm *restrict timeptr);`52104 **DESCRIPTION**52105 CX The functionality described on this reference page is aligned with the ISO C standard. Any
52106 conflict between the requirements described here and the ISO C standard is unintentional. This
52107 volume of IEEE Std 1003.1-200x defers to the ISO C standard.52108 The `wcsftime()` function shall be equivalent to the `strftime()` function, except that:

- 52109
- 52110 • The argument `wcs` points to the initial element of an array of wide characters into which
the generated output is to be placed.
 - 52111 • The argument `maxsize` indicates the maximum number of wide characters to be placed in
52112 the output array.
 - 52113 • The argument `format` is a wide-character string and the conversion specifications are
52114 replaced by corresponding sequences of wide characters.
 - 52115 • The return value indicates the number of wide characters placed in the output array.

52116 If copying takes place between objects that overlap, the behavior is undefined.

52117 **RETURN VALUE**52118 If the total number of resulting wide-character codes including the terminating null wide-
52119 character code is no more than `maxsize`, `wcsftime()` shall return the number of wide-character
52120 codes placed into the array pointed to by `wcs`, not including the terminating null wide-character
52121 code. Otherwise, zero is returned and the contents of the array are unspecified.52122 **ERRORS**

52123 No errors are defined.

52124 **EXAMPLES**

52125 None.

52126 **APPLICATION USAGE**

52127 None.

52128 **RATIONALE**

52129 None.

52130 **FUTURE DIRECTIONS**

52131 None.

52132 **SEE ALSO**52133 [*strftime\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<wchar.h>`52134 **CHANGE HISTORY**

52135 First released in Issue 4.

52136 **Issue 5**

52137 Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

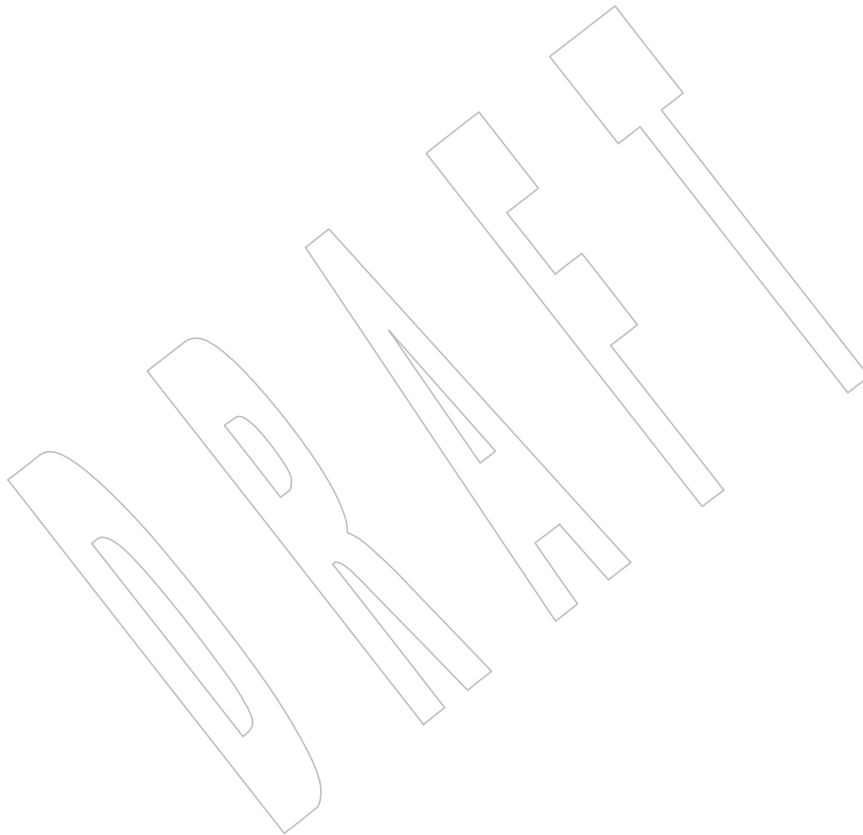
52138 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of the `format`
52139 argument is changed from `const char *` to `const wchar_t *`.

52140

Issue 6

52141

The *wcsftime()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.



52142 **NAME**52143 `wcslen`, `wcsnlen` — get length of a fixed-sized wide-character string52144 **SYNOPSIS**52145 `#include <wchar.h>`52146 `size_t wcslen(const wchar_t *ws);`52147 CX `size_t wcsnlen(const wchar_t *ws, size_t maxlen);`52148 **DESCRIPTION**52149 CX For `wcslen()`: The functionality described on this reference page is aligned with the ISO C
52150 standard. Any conflict between the requirements described here and the ISO C standard is
52151 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.52152 The `wcslen()` function shall compute the number of wide-character codes in the wide-character
52153 string to which `ws` points, not including the terminating null wide-character code.52154 CX The `wcsnlen()` function shall compute the smaller of the number of wide characters in the string
52155 to which `ws` points, not including the terminating null wide-character code, and the value of
52156 `maxlen`. The `wcsnlen()` function shall never examine more than the first `maxlen` characters of the
52157 wide-character string pointed to by `ws`.52158 **RETURN VALUE**52159 The `wcslen()` function shall return the length of `ws`.52160 CX The `wcsnlen()` function shall return an integer containing the smaller of either the length of the
52161 wide-character string pointed to by `ws` or `maxlen`.

52162 No return values are reserved to indicate an error.

52163 **ERRORS**

52164 No errors are defined.

52165 **EXAMPLES**

52166 None.

52167 **APPLICATION USAGE**

52168 None.

52169 **RATIONALE**

52170 None.

52171 **FUTURE DIRECTIONS**

52172 None.

52173 **SEE ALSO**52174 `strlen()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<wchar.h>`52175 **CHANGE HISTORY**

52176 First released in Issue 4. Derived from the MSE working draft.

52177 **Issue 7**52178 The `wcsnlen()` function is added from The Open Group Technical Standard, 2006, Extended API
52179 Set Part 1.

52180 **NAME**52181 `wcsncasecmp`, `wcsncasecmp_l` — case-insensitive wide-character string comparison52182 **SYNOPSIS**

```
52183 CX #include <wchar.h>
52184 int wcsncasecmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
52185 int wcsncasecmp_l(const char *ws1, const char *ws2,
52186 size_t n, locale_t locale);
```

52187 **DESCRIPTION**52188 Refer to [*wcscasecmp\(\)*](#).

52189 **NAME**52190 `wcsncat` — concatenate a wide-character string with part of another52191 **SYNOPSIS**52192 `#include <wchar.h>`52193 `wchar_t *wcsncat(wchar_t *restrict ws1, const wchar_t *restrict ws2,`
52194 `size_t n);`52195 **DESCRIPTION**52196 CX The functionality described on this reference page is aligned with the ISO C standard. Any
52197 conflict between the requirements described here and the ISO C standard is unintentional. This
52198 volume of IEEE Std 1003.1-200x defers to the ISO C standard.52199 The `wcsncat()` function shall append not more than *n* wide-character codes (a null wide-
52200 character code and wide-character codes that follow it are not appended) from the array pointed
52201 to by *ws2* to the end of the wide-character string pointed to by *ws1*. The initial wide-character
52202 code of *ws2* shall overwrite the null wide-character code at the end of *ws1*. A terminating null
52203 wide-character code shall always be appended to the result. If copying takes place between
52204 objects that overlap, the behavior is undefined.52205 **RETURN VALUE**52206 The `wcsncat()` function shall return *ws1*; no return value is reserved to indicate an error.52207 **ERRORS**

52208 No errors are defined.

52209 **EXAMPLES**

52210 None.

52211 **APPLICATION USAGE**

52212 None.

52213 **RATIONALE**

52214 None.

52215 **FUTURE DIRECTIONS**

52216 None.

52217 **SEE ALSO**52218 [*wcscat\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<wchar.h>`52219 **CHANGE HISTORY**

52220 First released in Issue 4. Derived from the MSE working draft.

52221 **Issue 6**52222 The `wcsncat()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

52223 **NAME**
 52224 `wcsncmp` — compare part of two wide-character strings

52225 **SYNOPSIS**
 52226 `#include <wchar.h>`
 52227 `int wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);`

52228 **DESCRIPTION**
 52229 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 52230 conflict between the requirements described here and the ISO C standard is unintentional. This
 52231 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52232 The `wcsncmp()` function shall compare not more than *n* wide-character codes (wide-character
 52233 codes that follow a null wide-character code are not compared) from the array pointed to by *ws1*
 52234 to the array pointed to by *ws2*.

52235 The sign of a non-zero return value shall be determined by the sign of the difference between the
 52236 values of the first pair of wide-character codes that differ in the objects being compared.

52237 **RETURN VALUE**
 52238 Upon successful completion, `wcsncmp()` shall return an integer greater than, equal to, or less
 52239 than 0, if the possibly null-terminated array pointed to by *ws1* is greater than, equal to, or less
 52240 than the possibly null-terminated array pointed to by *ws2*, respectively.

52241 **ERRORS**
 52242 No errors are defined.

52243 **EXAMPLES**
 52244 None.

52245 **APPLICATION USAGE**
 52246 None.

52247 **RATIONALE**
 52248 None.

52249 **FUTURE DIRECTIONS**
 52250 None.

52251 **SEE ALSO**
 52252 [*wscasecmp\(\)*](#), [*wscmp\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<wchar.h>`

52253 **CHANGE HISTORY**
 52254 First released in Issue 4. Derived from the MSE working draft.

52255 **NAME**

52256 wcpncpy, wcsncpy — copy a fixed-size wide-character string, returning a pointer to its end

52257 **SYNOPSIS**

52258 #include <wchar.h>

52259 CX wchar_t *wcpncpy(wchar_t restrict *ws1, const wchar_t *restrict ws2,
52260 size_t n);52261 wchar_t *wcsncpy(wchar_t *restrict ws1, const wchar_t *restrict ws2,
52262 size_t n);52263 **DESCRIPTION**52264 CX For *wcsncpy()*: The functionality described on this reference page is aligned with the ISO C
52265 standard. Any conflict between the requirements described here and the ISO C standard is
52266 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.52267 CX The *wcpncpy()* and *wcsncpy()* functions shall copy not more than *n* wide-character codes (wide-
52268 character codes that follow a null wide-character code are not copied) from the array pointed to
52269 by *ws2* to the array pointed to by *ws1*. If copying takes place between objects that overlap, the
52270 behavior is undefined.52271 If the array pointed to by *ws2* is a wide-character string that is shorter than *n* wide-character
52272 codes, null wide-character codes shall be appended to the copy in the array pointed to by *ws1*,
52273 until *n* wide-character codes in all are written.52274 **RETURN VALUE**52275 CX If any null wide-character codes were written into the destination, the *wcpncpy()* function shall
52276 return the address of the first such null wide-character code. Otherwise, it shall return *&ws1[n]*.52277 The *wcsncpy()* function shall return *ws1*.

52278 No return values are reserved to indicate an error.

52279 **ERRORS**

52280 No errors are defined.

52281 **EXAMPLES**

52282 None.

52283 **APPLICATION USAGE**52284 If there is no null wide-character code in the first *n* wide-character codes of the array pointed to
52285 by *ws2*, the result is not null-terminated.52286 **RATIONALE**

52287 None.

52288 **FUTURE DIRECTIONS**

52289 None.

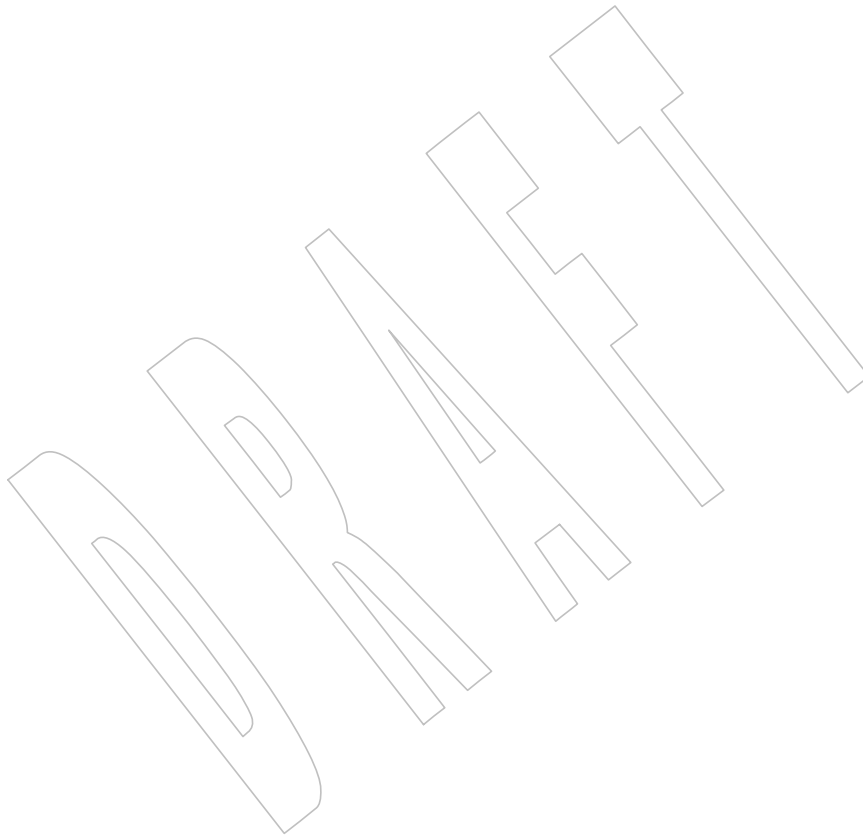
52290 **SEE ALSO**52291 *strncpy()*, *wcscpy()*, the Base Definitions volume of IEEE Std 1003.1-200x, <wchar.h>52292 **CHANGE HISTORY**

52293 First released in Issue 4. Derived from the MSE working draft.

52294 **Issue 6**52295 The *wcsncpy()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

52296
52297
52298**Issue 7**

The *wcpcnpy()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.



wcsnlen()

52299 **NAME**
52300 `wcsnlen` — get length of a fixed-sized wide-character string

52301 **SYNOPSIS**

52302 CX `#include <wchar.h>`
52303 `size_t wcsnlen(const wchar_t *ws, size_t maxlen);`

52304 **DESCRIPTION**

52305 Refer to *wcslen()*.

52306 **NAME**
52307 `wcsnrtoombs` — convert wide-character string to multi-byte string

52308 **SYNOPSIS**

```
52309 CX #include <wchar.h>  
52310 size_t wcsnrtoombs(char *dst, const wchar_t **src, size_t nwc,  
52311 size_t len, mbstate_t *ps);
```

52312 **DESCRIPTION**

52313 Refer to [wcsrtombs\(\)](#).

52314 **NAME**52315 `wcspbrk` — scan a wide-character string for a wide-character code52316 **SYNOPSIS**52317 `#include <wchar.h>`52318 `wchar_t *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);`52319 **DESCRIPTION**52320 CX The functionality described on this reference page is aligned with the ISO C standard. Any
52321 conflict between the requirements described here and the ISO C standard is unintentional. This
52322 volume of IEEE Std 1003.1-200x defers to the ISO C standard.52323 The `wcspbrk()` function shall locate the first occurrence in the wide-character string pointed to by
52324 `ws1` of any wide-character code from the wide-character string pointed to by `ws2`.52325 **RETURN VALUE**52326 Upon successful completion, `wcspbrk()` shall return a pointer to the wide-character code or a null
52327 pointer if no wide-character code from `ws2` occurs in `ws1`.52328 **ERRORS**

52329 No errors are defined.

52330 **EXAMPLES**

52331 None.

52332 **APPLICATION USAGE**

52333 None.

52334 **RATIONALE**

52335 None.

52336 **FUTURE DIRECTIONS**

52337 None.

52338 **SEE ALSO**52339 [*wcchr\(\)*](#), [*wcsrchr\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<wchar.h>`52340 **CHANGE HISTORY**

52341 First released in Issue 4. Derived from the MSE working draft.

52342 **NAME**
 52343 wcsrchr — wide-character string scanning operation

52344 **SYNOPSIS**
 52345 #include <wchar.h>
 52346 wchar_t *wcsrchr(const wchar_t *ws, wchar_t wc);

52347 **DESCRIPTION**
 52348 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 52349 conflict between the requirements described here and the ISO C standard is unintentional. This
 52350 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52351 The *wcsrchr()* function shall locate the last occurrence of *wc* in the wide-character string pointed
 52352 to by *ws*. The application shall ensure that the value of *wc* is a character representable as a type
 52353 **wchar_t** and a wide-character code corresponding to a valid character in the current locale. The
 52354 terminating null wide-character code shall be considered to be part of the wide-character string.

52355 **RETURN VALUE**
 52356 Upon successful completion, *wcsrchr()* shall return a pointer to the wide-character code or a null
 52357 pointer if *wc* does not occur in the wide-character string.

52358 **ERRORS**
 52359 No errors are defined.

52360 **EXAMPLES**
 52361 None.

52362 **APPLICATION USAGE**
 52363 None.

52364 **RATIONALE**
 52365 None.

52366 **FUTURE DIRECTIONS**
 52367 None.

52368 **SEE ALSO**
 52369 *wcschr()*, the Base Definitions volume of IEEE Std 1003.1-200x, <**wchar.h**>

52370 **CHANGE HISTORY**
 52371 First released in Issue 4. Derived from the MSE working draft.

52372 **Issue 6**
 52373 The normative text is updated to avoid use of the term “must” for application requirements.

52374 **NAME**
 52375 `wcsnrtoombs, wcsrtoombs` — convert a wide-character string to a character string (restartable)

52376 SYNOPSIS

52377 `#include <wchar.h>`

```
52378 CX size_t wcsnrtoombs(char *dst, const wchar_t **src, size_t nwc,  

52379 size_t len, mbstate_t *ps);  

52380 size_t wcsrtoombs(char *restrict dst, const wchar_t **restrict src,  

52381 size_t len, mbstate_t *restrict ps);
```

52382 DESCRIPTION

52383 CX For `wcsrtoombs()`: The functionality described on this reference page is aligned with the ISO C
 52384 standard. Any conflict between the requirements described here and the ISO C standard is
 52385 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52386 The `wcsrtoombs()` function shall convert a sequence of wide characters from the array indirectly
 52387 pointed to by `src` into a sequence of corresponding characters, beginning in the conversion state
 52388 described by the object pointed to by `ps`. If `dst` is not a null pointer, the converted characters
 52389 shall then be stored into the array pointed to by `dst`. Conversion continues up to and including a
 52390 terminating null wide character, which shall also be stored. Conversion shall stop earlier in the
 52391 following cases:

- 52392 • When a code is reached that does not correspond to a valid character
- 52393 • When the next character would exceed the limit of `len` total bytes to be stored in the array
 52394 pointed to by `dst` (and `dst` is not a null pointer)

52395 Each conversion shall take place as if by a call to the `wcrtomb()` function.

52396 If `dst` is not a null pointer, the pointer object pointed to by `src` shall be assigned either a null
 52397 pointer (if conversion stopped due to reaching a terminating null wide character) or the address
 52398 just past the last wide character converted (if any). If conversion stopped due to reaching a
 52399 terminating null wide character, the resulting state described shall be the initial conversion state.

52400 If `ps` is a null pointer, the `wcsrtoombs()` function shall use its own internal `mbstate_t` object, which
 52401 is initialized at program start-up to the initial conversion state. Otherwise, the `mbstate_t` object
 52402 pointed to by `ps` shall be used to completely describe the current conversion state of the
 52403 associated character sequence.

52404 CX If the application uses any of the `_POSIX_THREAD_SAFE_FUNCTIONS` or `_POSIX_THREADS`
 52405 functions, the application shall ensure that the `wcsrtoombs()` function is called with a non-NULL
 52406 `ps` argument.

52407 The `wcsnrtoombs()` function shall be equivalent to the `wcsrtoombs()` function, except that the
 52408 conversion is limited to the first `nwc` wide characters.

52409 The behavior of these functions shall be affected by the `LC_CTYPE` category of the current locale.

52410 The implementation shall behave as if no function defined in System Interfaces volume of
 52411 IEEE Std 1003.1-200x calls these functions.

52412 RETURN VALUE

52413 If conversion stops because a code is reached that does not correspond to a valid character, an
 52414 encoding error occurs. In this case, these functions shall store the value of the macro `[EILSEQ]` in
 52415 `errno` and return `(size_t)-1`; the conversion state is undefined. Otherwise, these functions shall
 52416 return the number of bytes in the resulting character sequence, not including the terminating
 52417 null (if any).

52418 **ERRORS**

52419 These functions may fail if:

52420 CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

52421 [EILSEQ] A wide-character code does not correspond to a valid character.

52422 **EXAMPLES**

52423 None.

52424 **APPLICATION USAGE**

52425 None.

52426 **RATIONALE**

52427 None.

52428 **FUTURE DIRECTIONS**

52429 None.

52430 **SEE ALSO**52431 *mbsinit()*, *wcrtomb()*, the Base Definitions volume of IEEE Std 1003.1-200x, <wchar.h>52432 **CHANGE HISTORY**52433 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
52434 (E).52435 **Issue 6**

52436 In the DESCRIPTION, a note on using this function in a threaded application is added.

52437 Extensions beyond the ISO C standard are marked.

52438 The normative text is updated to avoid use of the term “must” for application requirements.

52439 The *wcsrtoombs()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.52440 **Issue 7**52441 The *wcnsrtoombs()* function is added from The Open Group Technical Standard, 2006, Extended
52442 API Set Part 1.

52443 **NAME**
 52444 `wcssp``n` — get the length of a wide substring

52445 **SYNOPSIS**
 52446 `#include <wchar.h>`

52447 `size_t wcssp``n``(const wchar_t *ws1, const wchar_t *ws2);`

52448 **DESCRIPTION**

52449 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 52450 conflict between the requirements described here and the ISO C standard is unintentional. This
 52451 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52452 The `wcssp``n`(`)` function shall compute the length (in wide characters) of the maximum initial
 52453 segment of the wide-character string pointed to by `ws1` which consists entirely of wide-character
 52454 codes from the wide-character string pointed to by `ws2`.

52455 **RETURN VALUE**

52456 The `wcssp``n`(`)` function shall return the length of the initial substring of `ws1`; no return value is
 52457 reserved to indicate an error.

52458 **ERRORS**

52459 No errors are defined.

52460 **EXAMPLES**

52461 None.

52462 **APPLICATION USAGE**

52463 None.

52464 **RATIONALE**

52465 None.

52466 **FUTURE DIRECTIONS**

52467 None.

52468 **SEE ALSO**

52469 `wcscsp``n`(`)`, the Base Definitions volume of IEEE Std 1003.1-200x, `<wchar.h>`

52470 **CHANGE HISTORY**

52471 First released in Issue 4. Derived from the MSE working draft.

52472 **Issue 5**

52473 The RETURN VALUE section is updated to indicate that `wcssp``n`(`)` returns the length of `ws1`
 52474 rather than `ws1` itself.

52475 **NAME**52476 `wcsstr` — find a wide-character substring52477 **SYNOPSIS**52478 `#include <wchar.h>`52479 `wchar_t *wcsstr(const wchar_t *restrict ws1,`
52480 `const wchar_t *restrict ws2);`52481 **DESCRIPTION**52482 CX The functionality described on this reference page is aligned with the ISO C standard. Any
52483 conflict between the requirements described here and the ISO C standard is unintentional. This
52484 volume of IEEE Std 1003.1-200x defers to the ISO C standard.52485 The `wcsstr()` function shall locate the first occurrence in the wide-character string pointed to by
52486 `ws1` of the sequence of wide characters (excluding the terminating null wide character) in the
52487 wide-character string pointed to by `ws2`.52488 **RETURN VALUE**52489 Upon successful completion, `wcsstr()` shall return a pointer to the located wide-character string,
52490 or a null pointer if the wide-character string is not found.52491 If `ws2` points to a wide-character string with zero length, the function shall return `ws1`.52492 **ERRORS**

52493 No errors are defined.

52494 **EXAMPLES**

52495 None.

52496 **APPLICATION USAGE**

52497 None.

52498 **RATIONALE**

52499 None.

52500 **FUTURE DIRECTIONS**

52501 None.

52502 **SEE ALSO**52503 [wcschr\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<wchar.h>`52504 **CHANGE HISTORY**52505 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
52506 (E).52507 **Issue 6**52508 The `wcsstr()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

52509 NAME

52510 wcstod, wcstof, wcstold — convert a wide-character string to a double-precision number

52511 SYNOPSIS

52512 #include <wchar.h>

52513 double wcstod(const wchar_t *restrict nptr, wchar_t **restrict endptr);

52514 float wcstof(const wchar_t *restrict nptr, wchar_t **restrict endptr);

52515 long double wcstold(const wchar_t *restrict nptr,

52516 wchar_t **restrict endptr);

52517 DESCRIPTION

52518 CX The functionality described on this reference page is aligned with the ISO C standard. Any
52519 conflict between the requirements described here and the ISO C standard is unintentional. This
52520 volume of IEEE Std 1003.1-200x defers to the ISO C standard.52521 These functions shall convert the initial portion of the wide-character string pointed to by *nptr* to
52522 **double**, **float**, and **long double** representation, respectively. First, they shall decompose the
52523 input wide-character string into three parts:

- 52524 1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by
-
- 52525
- iswspace()*
-)
-
- 52526 2. A subject sequence interpreted as a floating-point constant or representing infinity or
-
- 52527 NaN
-
- 52528 3. A final wide-character string of one or more unrecognized wide-character codes,
-
- 52529 including the terminating null wide-character code of the input wide-character string

52530 Then they shall attempt to convert the subject sequence to a floating-point number, and return
52531 the result.52532 The expected form of the subject sequence is an optional plus or minus sign, then one of the
52533 following:

- 52534 • A non-empty sequence of decimal digits optionally containing a radix character; then an
-
- 52535 optional exponent part consisting of the wide character 'e' or the wide character 'E',
-
- 52536 optionally followed by a '+' or '-' wide character, and then followed by one or more
-
- 52537 decimal digits
-
- 52538 • A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix
-
- 52539 character; then an optional binary exponent part consisting of the wide character 'p' or
-
- 52540 the wide character 'P', optionally followed by a '+' or '-' wide character, and then
-
- 52541 followed by one or more decimal digits
-
- 52542 • One of INF or INFINITY, or any other wide string equivalent except for case
-
- 52543 • One of NAN or NAN(
- n-wchar-sequence_{opt}*
-), or any other wide string ignoring case in the
-
- 52544 NAN part, where:

52545 n-wchar-sequence :

52546 digit

52547 nondigit

52548 n-wchar-sequence digit

52549 n-wchar-sequence nondigit

52550 The subject sequence is defined as the longest initial subsequence of the input wide string,
52551 starting with the first non-white-space wide character, that is of the expected form. The subject
52552 sequence contains no wide characters if the input wide string is not of the expected form.

52553 If the subject sequence has the expected form for a floating-point number, the sequence of wide
 52554 characters starting with the first digit or the radix character (whichever occurs first) shall be
 52555 interpreted as a floating constant according to the rules of the C language, except that the radix
 52556 character shall be used in place of a period, and that if neither an exponent part nor a radix
 52557 character appears in a decimal floating-point number, or if a binary exponent part does not
 52558 appear in a hexadecimal floating-point number, an exponent part of the appropriate type with
 52559 value zero shall be assumed to follow the last digit in the string. If the subject sequence begins
 52560 with a minus sign, the sequence shall be interpreted as negated. A wide-character sequence INF
 52561 or INFINITY shall be interpreted as an infinity, if representable in the return type, else as if it
 52562 were a floating constant that is too large for the range of the return type. A wide-character
 52563 sequence NAN or NAN(*n-wchar-sequence_{opt}*) shall be interpreted as a quiet NaN, if supported in
 52564 the return type, else as if it were a subject sequence part that does not have the expected form;
 52565 the meaning of the *n-wchar* sequences is implementation-defined. A pointer to the final wide
 52566 string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

52567 If the subject sequence has the hexadecimal form and FLT_RADIX is a power of 2, the
 52568 conversion shall be rounded in an implementation-defined manner.

52569 CX The radix character shall be as defined in the locale of the process (category *LC_NUMERIC*). In
 52570 the POSIX locale, or in a locale where the radix character is not defined, the radix character shall
 52571 default to a period (' . ').

52572 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
 52573 accepted.

52574 If the subject sequence is empty or does not have the expected form, no conversion shall be
 52575 performed; the value of *nptr* shall be stored in the object pointed to by *endptr*, provided that
 52576 *endptr* is not a null pointer.

52577 CX The *wcstod()* function shall not change the setting of *errno* if successful.

52578 Since 0 is returned on error and is also a valid return on success, an application wishing to check
 52579 for error situations should set *errno* to 0, then call *wcstod()*, *wcstof()*, or *wcstold()*, then check
 52580 *errno*.

52581 RETURN VALUE

52582 Upon successful completion, these functions shall return the converted value. If no conversion
 52583 CX could be performed, 0 shall be returned and *errno* may be set to [EINVAL].

52584 If the correct value is outside the range of representable values, \pm HUGE_VAL, \pm HUGE_VALF, or
 52585 \pm HUGE_VALL shall be returned (according to the sign of the value), and *errno* shall be set to
 52586 [ERANGE].

52587 If the correct value would cause underflow, a value whose magnitude is no greater than the
 52588 smallest normalized positive number in the return type shall be returned and *errno* set to
 52589 [ERANGE].

52590 ERRORS

52591 The *wcstod()* function shall fail if:

52592 [ERANGE] The value to be returned would cause overflow or underflow.

52593 The *wcstod()* function may fail if:

52594 CX [EINVAL] No conversion could be performed.

52595
52596**EXAMPLES**

None.

52597
52598
52599
52600
52601**APPLICATION USAGE**

If the subject sequence has the hexadecimal form and FLT_RADIX is not a power of 2, and the result is not exactly representable, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

52602
52603
52604
52605
52606
52607
52608
52609

If the subject sequence has the decimal form and at most DECIMAL_DIG (defined in `<float.h>`) significant digits, the result should be correctly rounded. If the subject sequence *D* has the decimal form and more than DECIMAL_DIG significant digits, consider the two bounding, adjacent decimal strings *L* and *U*, both having DECIMAL_DIG significant digits, such that the values of *L*, *D*, and *U* satisfy " $L \leq D \leq U$ ". The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding *L* and *U* according to the current rounding direction, with the extra stipulation that the error with respect to *D* should have a correct sign for the current rounding direction.

52610
52611**RATIONALE**

None.

52612
52613**FUTURE DIRECTIONS**

None.

52614
52615
52616**SEE ALSO**

iswspace(), *localeconv()*, *scanf()*, *setlocale()*, *wcstol()*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale, `<float.h>`, `<wchar.h>`

52617
52618**CHANGE HISTORY**

First released in Issue 4. Derived from the MSE working draft.

52619
52620**Issue 5**

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

52621
52622**Issue 6**

Extensions beyond the ISO C standard are marked.

52623
52624

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

52627
52628

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The *wcstod()* prototype is updated.
- The *wcstof()* and *wcstold()* functions are added.
- If the correct value for *wcstod()* would cause underflow, the return value changed from 0 (as specified in Issue 5) to the smallest normalized positive number.
- The DESCRIPTION, RETURN VALUE, and APPLICATION USAGE sections are extensively updated.

52634
52635

ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

52636

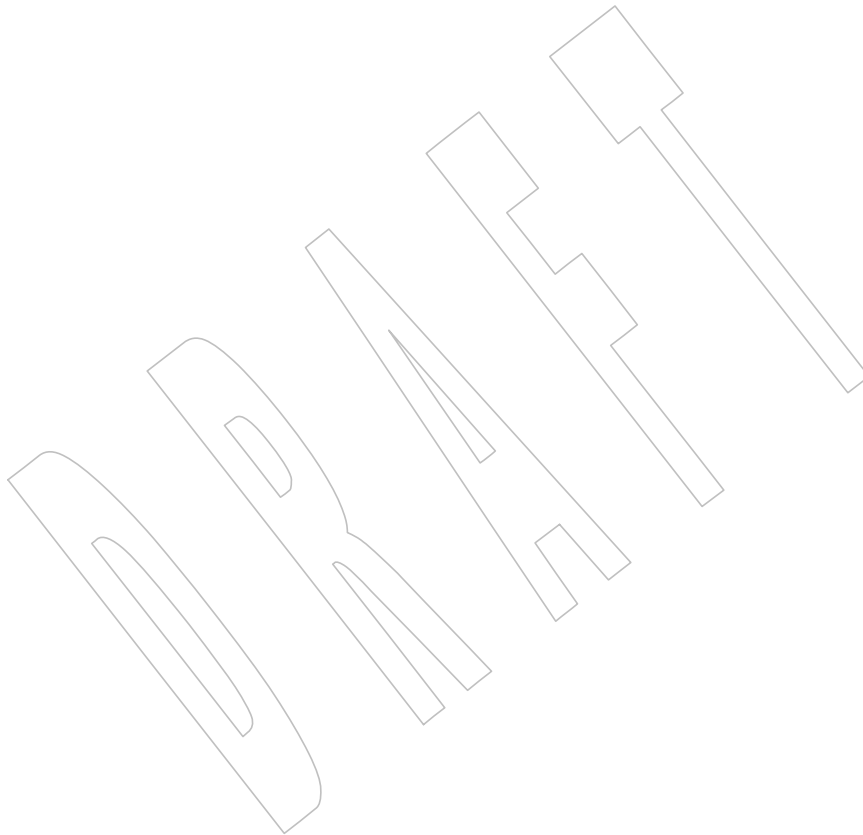
IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/66 is applied, correcting the second paragraph in the RETURN VALUE section.

52637

Issue 7

52638

Austin Group Interpretation 1003.1-2001 #015 is applied.



52639 **NAME**52640 `wcstoimax`, `wcstoumax` — convert a wide-character string to an integer type52641 **SYNOPSIS**52642 `#include <stddef.h>`52643 `#include <inttypes.h>`

```
52644 intmax_t wcstoimax(const wchar_t *restrict nptr,
52645                  wchar_t **restrict endptr, int base);
52646 uintmax_t wcstoumax(const wchar_t *restrict nptr,
52647                   wchar_t **restrict endptr, int base);
```

52648 **DESCRIPTION**

52649 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 52650 conflict between the requirements described here and the ISO C standard is unintentional. This
 52651 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52652 These functions shall be equivalent to the `wcstol()`, `wcstoll()`, `wcstoul()`, and `wcstoull()` functions,
 52653 respectively, except that the initial portion of the wide string shall be converted to `intmax_t` and
 52654 `uintmax_t` representation, respectively.

52655 **RETURN VALUE**

52656 These functions shall return the converted value, if any.

52657 If no conversion could be performed, zero shall be returned. If the correct value is outside the
 52658 range of representable values, `{INTMAX_MAX}`, `{INTMAX_MIN}`, or `{UINTMAX_MAX}` shall
 52659 be returned (according to the return type and sign of the value, if any), and `errno` shall be set to
 52660 `[ERANGE]`.

52661 **ERRORS**

52662 These functions shall fail if:

52663 `[EINVAL]` The value of `base` is not supported.52664 `[ERANGE]` The value to be returned is not representable.

52665 These functions may fail if:

52666 `[EINVAL]` No conversion could be performed.52667 **EXAMPLES**

52668 None.

52669 **APPLICATION USAGE**

52670 None.

52671 **RATIONALE**

52672 None.

52673 **FUTURE DIRECTIONS**

52674 None.

52675 **SEE ALSO**

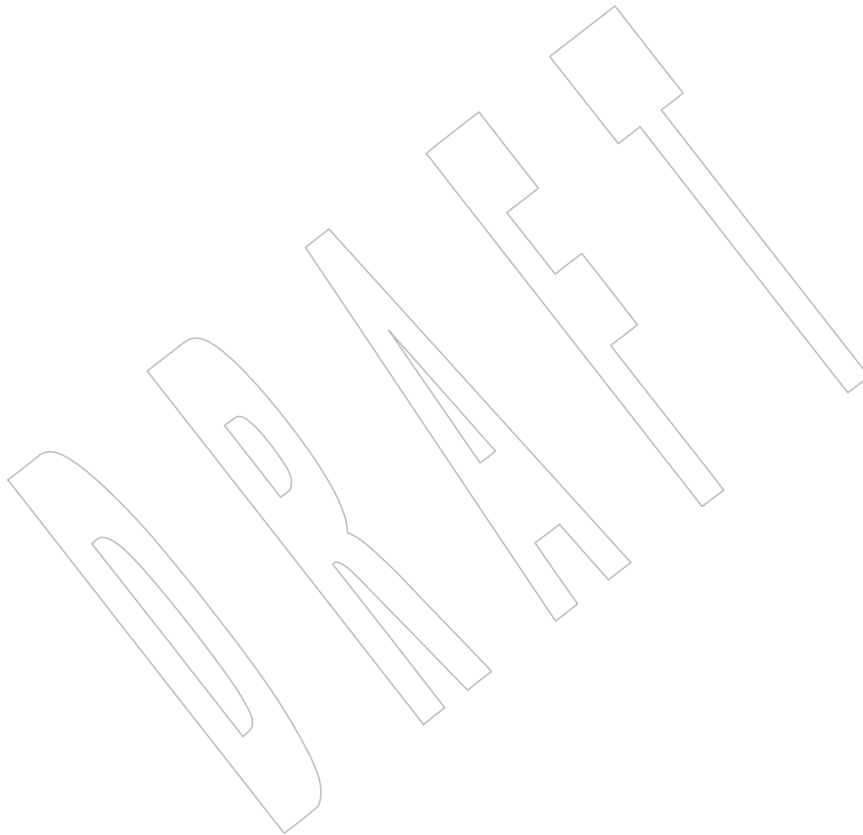
52676 `wcstol()`, `wcstoul()`, the Base Definitions volume of IEEE Std 1003.1-200x, `<inttypes.h>`,
 52677 `<stddef.h>`

52678

52679

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.



52680 **NAME**
 52681 `wcstok` — split a wide-character string into tokens

52682 **SYNOPSIS**
 52683 `#include <wchar.h>`

52684 `wchar_t *wcstok(wchar_t *restrict ws1, const wchar_t *restrict ws2,`
 52685 `wchar_t **restrict ptr);`

52686 **DESCRIPTION**

52687 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 52688 conflict between the requirements described here and the ISO C standard is unintentional. This
 52689 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52690 A sequence of calls to `wcstok()` shall break the wide-character string pointed to by `ws1` into a
 52691 sequence of tokens, each of which shall be delimited by a wide-character code from the wide-
 52692 character string pointed to by `ws2`. The `ptr` argument points to a caller-provided `wchar_t` pointer
 52693 into which the `wcstok()` function shall store information necessary for it to continue scanning the
 52694 same wide-character string.

52695 The first call in the sequence has `ws1` as its first argument, and is followed by calls with a null
 52696 pointer as their first argument. The separator string pointed to by `ws2` may be different from call
 52697 to call.

52698 The first call in the sequence shall search the wide-character string pointed to by `ws1` for the first
 52699 wide-character code that is *not* contained in the current separator string pointed to by `ws2`. If no
 52700 such wide-character code is found, then there are no tokens in the wide-character string pointed
 52701 to by `ws1` and `wcstok()` shall return a null pointer. If such a wide-character code is found, it shall
 52702 be the start of the first token.

52703 The `wcstok()` function shall then search from there for a wide-character code that *is* contained in
 52704 the current separator string. If no such wide-character code is found, the current token extends
 52705 to the end of the wide-character string pointed to by `ws1`, and subsequent searches for a token
 52706 shall return a null pointer. If such a wide-character code is found, it shall be overwritten by a
 52707 null wide character, which terminates the current token. The `wcstok()` function shall save a
 52708 pointer to the following wide-character code, from which the next search for a token shall start.

52709 Each subsequent call, with a null pointer as the value of the first argument, shall start searching
 52710 from the saved pointer and behave as described above.

52711 The implementation shall behave as if no function calls `wcstok()`.

52712 **RETURN VALUE**

52713 Upon successful completion, the `wcstok()` function shall return a pointer to the first wide-
 52714 character code of a token. Otherwise, if there is no token, `wcstok()` shall return a null pointer.

52715 **ERRORS**

52716 No errors are defined.

52717
52718

52719
52720

52721
52722

52723
52724

52725
52726

52727
52728

52729
52730
52731

52732
52733

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, <wchar.h>

CHANGE HISTORY

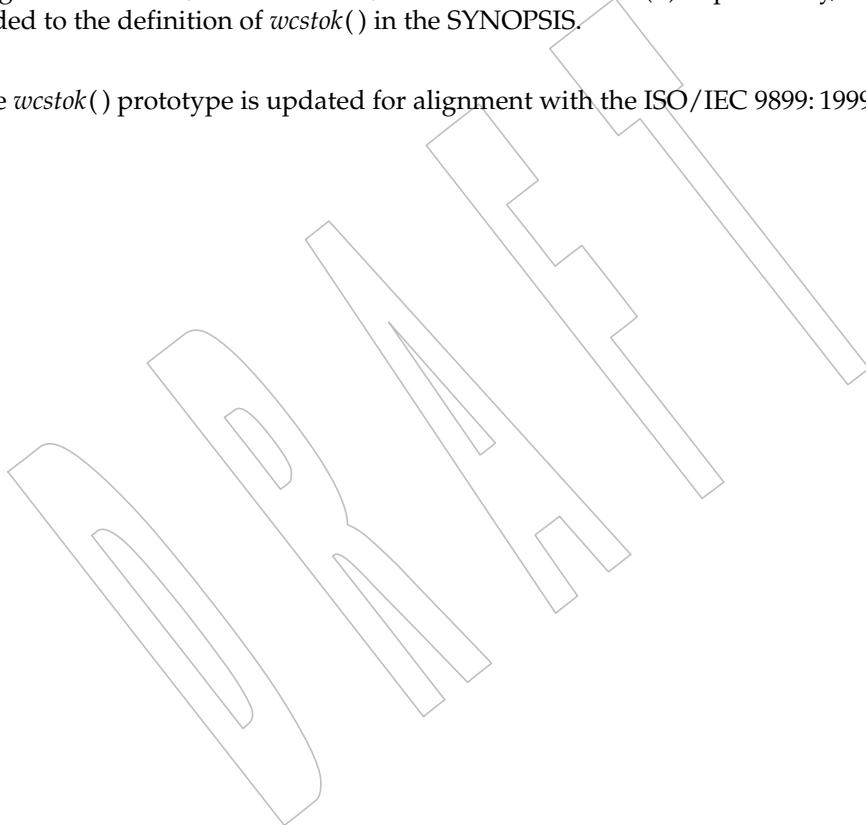
First released in Issue 4.

Issue 5

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, a third argument is added to the definition of *wcstok()* in the SYNOPSIS.

Issue 6

The *wcstok()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.



52734 **NAME**
 52735 `wcstol`, `wcstoll` — convert a wide-character string to a long integer

52736 **SYNOPSIS**
 52737

```
#include <wchar.h>
```


 52738

```
long wcstol(const wchar_t *restrict nptr, wchar_t **restrict endptr,
```


 52739

```
int base);
```


 52740

```
long long wcstoll(const wchar_t *restrict nptr,
```


 52741

```
wchar_t **restrict endptr, int base);
```

52742 **DESCRIPTION**

52743 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 52744 conflict between the requirements described here and the ISO C standard is unintentional. This
 52745 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52746 These functions shall convert the initial portion of the wide-character string pointed to by *nptr* to
 52747 **long** and **long long**, respectively. First, they shall decompose the input string into three parts:

- 52748 1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by
 52749 `iswspace()`)
- 52750 2. A subject sequence interpreted as an integer represented in some radix determined by the
 52751 value of *base*
- 52752 3. A final wide-character string of one or more unrecognized wide-character codes,
 52753 including the terminating null wide-character code of the input wide-character string

52754 Then they shall attempt to convert the subject sequence to an integer, and return the result.

52755 If *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant,
 52756 or hexadecimal constant, any of which may be preceded by a '+' or '-' sign. A decimal
 52757 constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal
 52758 constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7'
 52759 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the
 52760 decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

52761 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
 52762 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
 52763 by a '+' or '-' sign, but not including an integer suffix. The letters from 'a' (or 'A') to 'z'
 52764 (or 'Z') inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less
 52765 than that of *base* shall be permitted. If the value of *base* is 16, the wide-character code
 52766 representations of 0x or 0X may optionally precede the sequence of letters and digits, following
 52767 the sign if present.

52768 The subject sequence is defined as the longest initial subsequence of the input wide-character
 52769 string, starting with the first non-white-space wide-character code that is of the expected form.
 52770 The subject sequence contains no wide-character codes if the input wide-character string is
 52771 empty or consists entirely of white-space wide-character code, or if the first non-white-space
 52772 wide-character code is other than a sign or a permissible letter or digit.

52773 If the subject sequence has the expected form and *base* is 0, the sequence of wide-character codes
 52774 starting with the first digit shall be interpreted as an integer constant. If the subject sequence has
 52775 the expected form and the value of *base* is between 2 and 36, it shall be used as the base for
 52776 conversion, ascribing to each letter its value as given above. If the subject sequence begins with a
 52777 minus sign, the value resulting from the conversion shall be negated. A pointer to the final wide-
 52778 character string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a
 52779 null pointer.

52780 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
52781 accepted.

52782 If the subject sequence is empty or does not have the expected form, no conversion shall be
52783 performed; the value of *nptr* shall be stored in the object pointed to by *endptr*, provided that
52784 *endptr* is not a null pointer.

52785 CX These functions shall not change the setting of *errno* if successful.

52786 Since 0, {LONG_MIN} or {LLONG_MIN} and {LONG_MAX} or {LLONG_MAX} are returned on
52787 error and are also valid returns on success, an application wishing to check for error situations
52788 should set *errno* to 0, then call *wcstol()* or *wcstoll()*, then check *errno*.

52789 RETURN VALUE

52790 Upon successful completion, these functions shall return the converted value, if any. If no
52791 CX conversion could be performed, 0 shall be returned and *errno* may be set to indicate the error. If
52792 the correct value is outside the range of representable values, {LONG_MIN}, {LONG_MAX},
52793 {LLONG_MIN}, or {LLONG_MAX} shall be returned (according to the sign of the value), and
52794 *errno* set to [ERANGE].

52795 ERRORS

52796 These functions shall fail if:

52797 CX [EINVAL] The value of *base* is not supported.

52798 [ERANGE] The value to be returned is not representable.

52799 These functions may fail if:

52800 CX [EINVAL] No conversion could be performed.

52801 EXAMPLES

52802 None.

52803 APPLICATION USAGE

52804 None.

52805 RATIONALE

52806 None.

52807 FUTURE DIRECTIONS

52808 None.

52809 SEE ALSO

52810 *iswalpha()*, *scanf()*, *wcstod()*, the Base Definitions volume of IEEE Std 1003.1-200x, <wchar.h>

52811 CHANGE HISTORY

52812 First released in Issue 4. Derived from the MSE working draft.

52813 Issue 5

52814 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

52815 Issue 6

52816 Extensions beyond the ISO C standard are marked.

52817 The following new requirements on POSIX implementations derive from alignment with the
52818 Single UNIX Specification:

- 52819 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
- 52820 added if no conversion could be performed.

52821 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

wcstol()

52822

- The *wcstol()* prototype is updated.

52823

- The *wcstoll()* function is added.

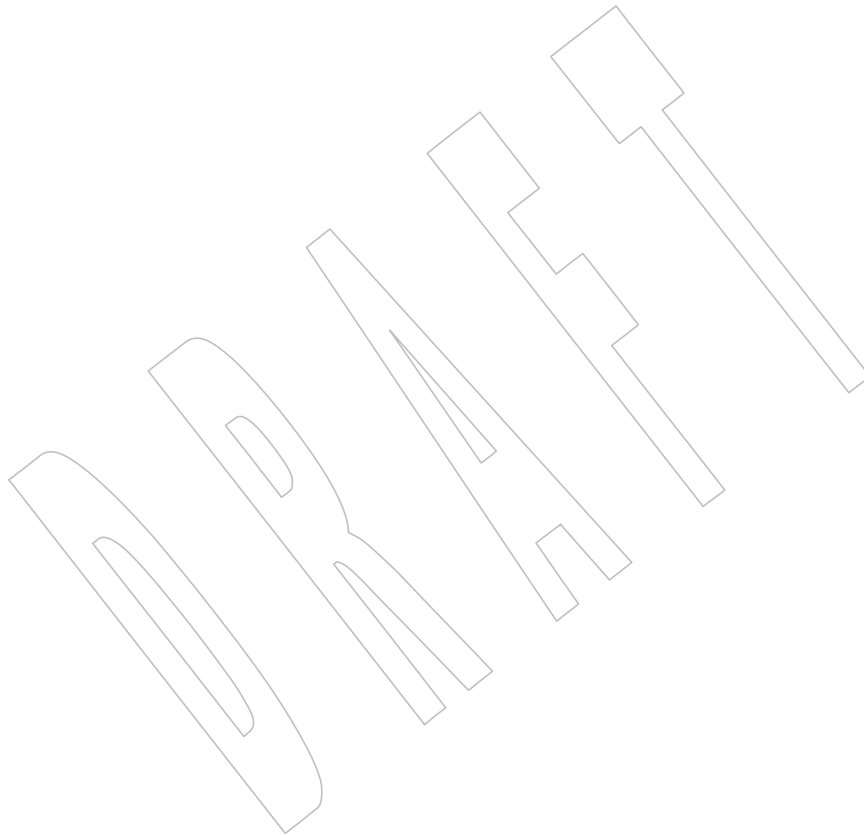
52824

Issue 7

52825

SD5-XSH-ERN-56 is applied, removing the reference to **unsigned long** and **unsigned long long** from the DESCRIPTION.

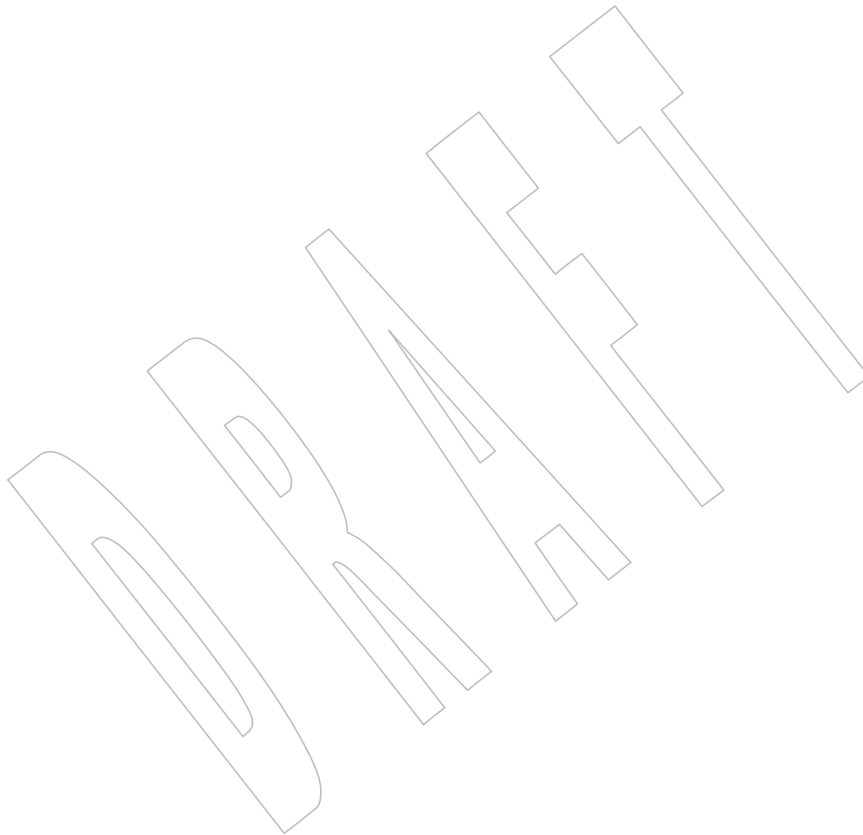
52826



52827 **NAME**
52828 `wcstold` — convert a wide-character string to a double-precision number

52829 **SYNOPSIS**
52830 `#include <wchar.h>`
52831 `long double wcstold(const wchar_t *restrict nptr,`
52832 `wchar_t **restrict endptr);`

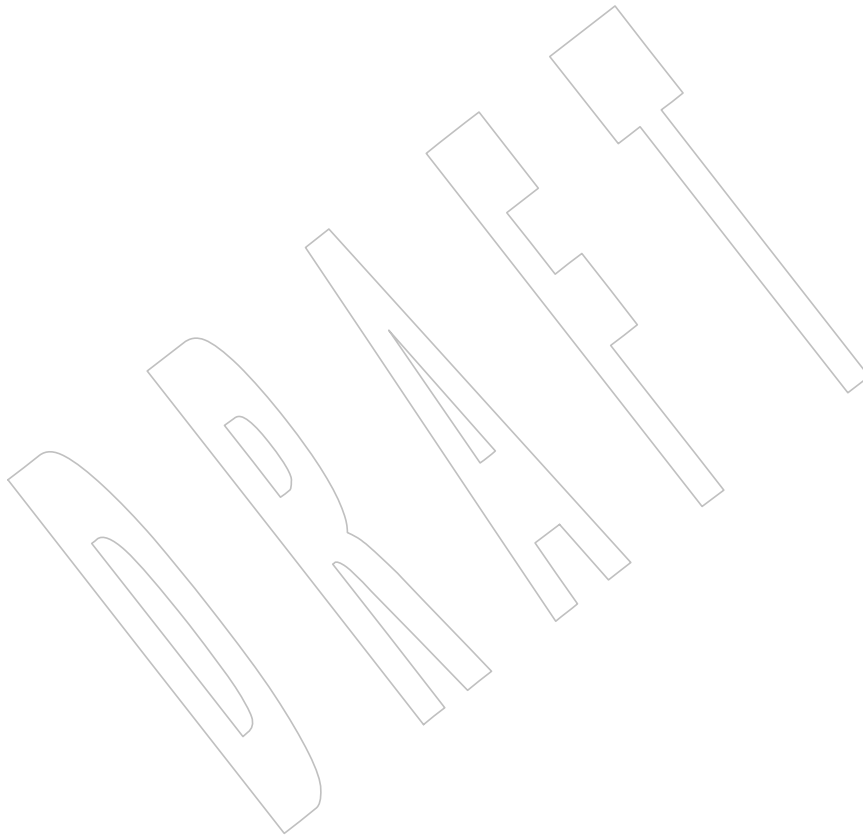
52833 **DESCRIPTION**
52834 Refer to *wcstod()*.



52835 **NAME**
52836 `wcstoll` — convert a wide-character string to a long integer

52837 **SYNOPSIS**
52838 `#include <wchar.h>`
52839 `long long wcstoll(const wchar_t *restrict nptr,`
52840 `wchar_t **restrict endptr, int base);`

52841 **DESCRIPTION**
52842 Refer to *wcstol()*.



52843 **NAME**
 52844 `wcstombs` — convert a wide-character string to a character string

52845 **SYNOPSIS**
 52846 `#include <stdlib.h>`
 52847 `size_t wcstombs(char *restrict s, const wchar_t *restrict pwcs,`
 52848 `size_t n);`

52849 DESCRIPTION

52850 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 52851 conflict between the requirements described here and the ISO C standard is unintentional. This
 52852 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52853 The `wcstombs()` function shall convert the sequence of wide-character codes that are in the array
 52854 pointed to by `pwcs` into a sequence of characters that begins in the initial shift state and store
 52855 these characters into the array pointed to by `s`, stopping if a character would exceed the limit of `n`
 52856 total bytes or if a null byte is stored. Each wide-character code shall be converted as if by a call to
 52857 `wctomb()`, except that the shift state of `wctomb()` shall not be affected.

52858 The behavior of this function shall be affected by the `LC_CTYPE` category of the current locale.

52859 No more than `n` bytes shall be modified in the array pointed to by `s`. If copying takes place
 52860 CX between objects that overlap, the behavior is undefined. If `s` is a null pointer, `wcstombs()` shall
 52861 return the length required to convert the entire array regardless of the value of `n`, but no values
 52862 are stored.

52863 The `wcstombs()` function need not be thread-safe. A function that is not required to be thread-
 52864 safe is not required to be reentrant.

52865 RETURN VALUE

52866 If a wide-character code is encountered that does not correspond to a valid character (of one or
 52867 more bytes each), `wcstombs()` shall return `(size_t)-1`. Otherwise, `wcstombs()` shall return the
 52868 number of bytes stored in the character array, not including any terminating null byte. The array
 52869 shall not be null-terminated if the value returned is `n`.

52870 ERRORS

52871 The `wcstombs()` function may fail if:

52872 CX **[EILSEQ]** A wide-character code does not correspond to a valid character.

52873 EXAMPLES

52874 None.

52875 APPLICATION USAGE

52876 None.

52877 RATIONALE

52878 None.

52879 FUTURE DIRECTIONS

52880 None.

52881 SEE ALSO

52882 `mblen()`, `mbtowc()`, `mbstowcs()`, `wctomb()`, the Base Definitions volume of IEEE Std 1003.1-200x,
 52883 `<stdlib.h>`

52884

CHANGE HISTORY

52885

First released in Issue 4. Derived from the ISO C standard.

52886

Issue 6

52887

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

52888

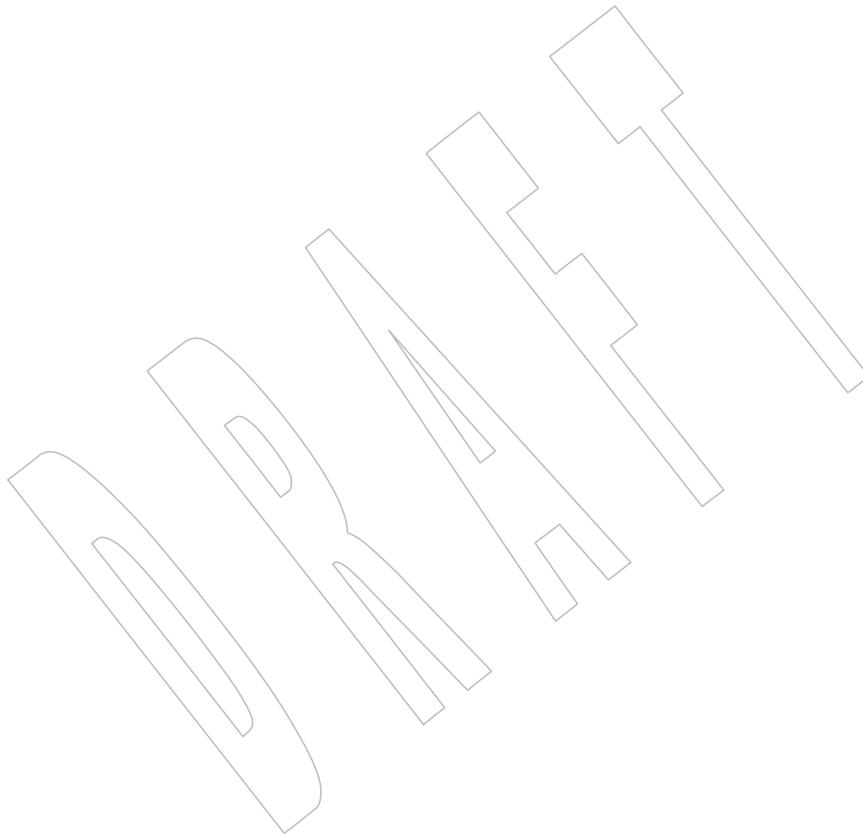
- The DESCRIPTION states the effect of when *s* is a null pointer.
- The [EILSEQ] error condition is added.

52889

52890

52891

The *wcstombs()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.



52892 **NAME**
 52893 `wcstoul`, `wcstoull` — convert a wide-character string to an unsigned long

52894 **SYNOPSIS**
 52895

```
#include <wchar.h>
```


 52896

```
unsigned long wcstoul(const wchar_t *restrict nptr,
```


 52897

```
                    wchar_t **restrict endptr, int base);
```


 52898

```
unsigned long long wcstoull(const wchar_t *restrict nptr,
```


 52899

```
                           wchar_t **restrict endptr, int base);
```

52900 **DESCRIPTION**

52901 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 52902 conflict between the requirements described here and the ISO C standard is unintentional. This
 52903 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52904 The `wcstoul()` and `wcstoull()` functions shall convert the initial portion of the wide-character
 52905 string pointed to by `nptr` to **unsigned long** and **unsigned long long** representation, respectively.
 52906 First, they shall decompose the input wide-character string into three parts:

- 52907 1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by
 52908 `iswspace()`)
- 52909 2. A subject sequence interpreted as an integer represented in some radix determined by the
 52910 value of `base`
- 52911 3. A final wide-character string of one or more unrecognized wide-character codes,
 52912 including the terminating null wide-character code of the input wide-character string

52913 Then they shall attempt to convert the subject sequence to an unsigned integer, and return the
 52914 result.

52915 If `base` is 0, the expected form of the subject sequence is that of a decimal constant, octal constant,
 52916 or hexadecimal constant, any of which may be preceded by a '+' or '-' sign. A decimal
 52917 constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal
 52918 constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7'
 52919 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the
 52920 decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

52921 If the value of `base` is between 2 and 36, the expected form of the subject sequence is a sequence
 52922 of letters and digits representing an integer with the radix specified by `base`, optionally preceded
 52923 by a '+' or '-' sign, but not including an integer suffix. The letters from 'a' (or 'A') to 'z'
 52924 (or 'Z') inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less
 52925 than that of `base` shall be permitted. If the value of `base` is 16, the wide-character codes 0x or 0X
 52926 may optionally precede the sequence of letters and digits, following the sign if present.

52927 The subject sequence is defined as the longest initial subsequence of the input wide-character
 52928 string, starting with the first wide-character code that is not white space and is of the expected
 52929 form. The subject sequence contains no wide-character codes if the input wide-character string is
 52930 empty or consists entirely of white-space wide-character codes, or if the first wide-character
 52931 code that is not white space is other than a sign or a permissible letter or digit.

52932 If the subject sequence has the expected form and `base` is 0, the sequence of wide-character codes
 52933 starting with the first digit shall be interpreted as an integer constant. If the subject sequence has
 52934 the expected form and the value of `base` is between 2 and 36, it shall be used as the base for
 52935 conversion, ascribing to each letter its value as given above. If the subject sequence begins with a
 52936 minus sign, the value resulting from the conversion shall be negated. A pointer to the final wide-
 52937 character string shall be stored in the object pointed to by `endptr`, provided that `endptr` is not a

wcstoul()

52938 null pointer.

52939 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
52940 accepted.

52941 If the subject sequence is empty or does not have the expected form, no conversion shall be
52942 performed; the value of *nptr* shall be stored in the object pointed to by *endptr*, provided that
52943 *endptr* is not a null pointer.

52944 CX The *wcstoul()* function shall not change the setting of *errno* if successful.

52945 Since 0, {ULONG_MAX}, and {ULLONG_MAX} are returned on error and 0 is also a valid return
52946 on success, an application wishing to check for error situations should set *errno* to 0, then call
52947 *wcstoul()* or *wcstoull()*, then check *errno*.

RETURN VALUE

52948 Upon successful completion, the *wcstoul()* and *wcstoull()* functions shall return the converted
52949 value, if any. If no conversion could be performed, 0 shall be returned and *errno* may be set to
52950 indicate the error. If the correct value is outside the range of representable values,
52951 {ULONG_MAX} or {ULLONG_MAX} respectively shall be returned and *errno* set to [ERANGE].
52952

ERRORS

52953 These functions shall fail if:

52954
52955 CX [EINVAL] The value of *base* is not supported.

52956 [ERANGE] The value to be returned is not representable.

52957 These functions may fail if:

52958 CX [EINVAL] No conversion could be performed.

EXAMPLES

52959 None.
52960

APPLICATION USAGE

52961 None.
52962

RATIONALE

52963 None.
52964

FUTURE DIRECTIONS

52965 None.
52966

SEE ALSO

52967 *iswalphabet()*, *scanf()*, *wcstod()*, *wcstol()*, the Base Definitions volume of IEEE Std 1003.1-200x,
52968 <wchar.h>
52969

CHANGE HISTORY

52970 First released in Issue 4. Derived from the MSE working draft.
52971

Issue 5

52972 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.
52973

Issue 6

52974 Extensions beyond the ISO C standard are marked.
52975

52976 The following new requirements on POSIX implementations derive from alignment with the
52977 Single UNIX Specification:

- The [EINVAL] error condition is added for when the value of *base* is not supported.

52979 In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
52980 added if no conversion could be performed.

52981

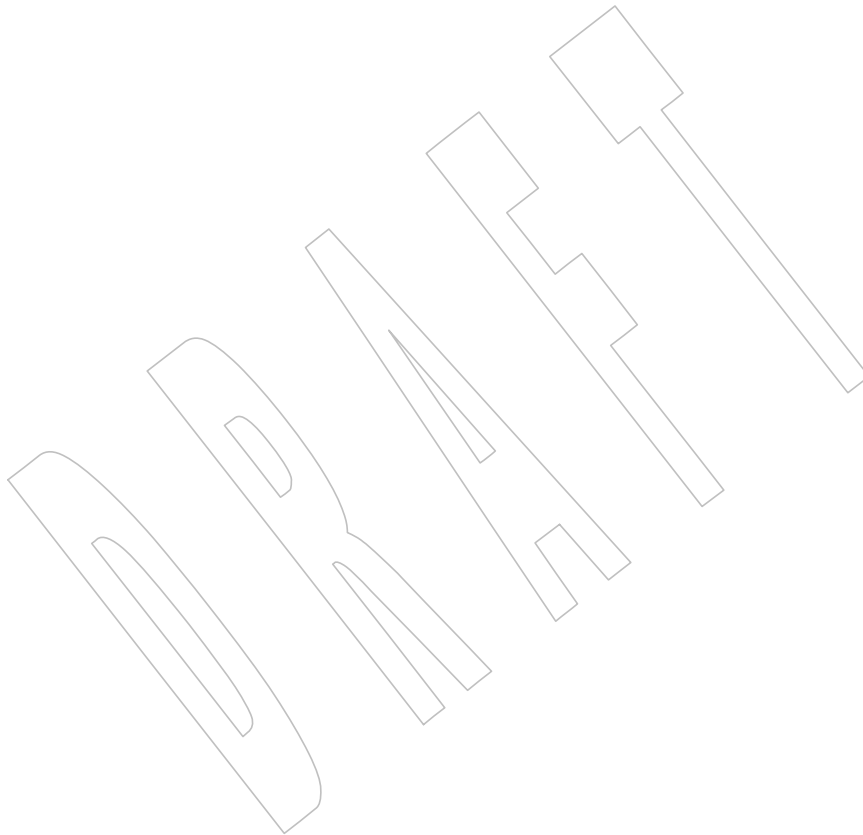
The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

52982

- The *wcstoul()* prototype is updated.

52983

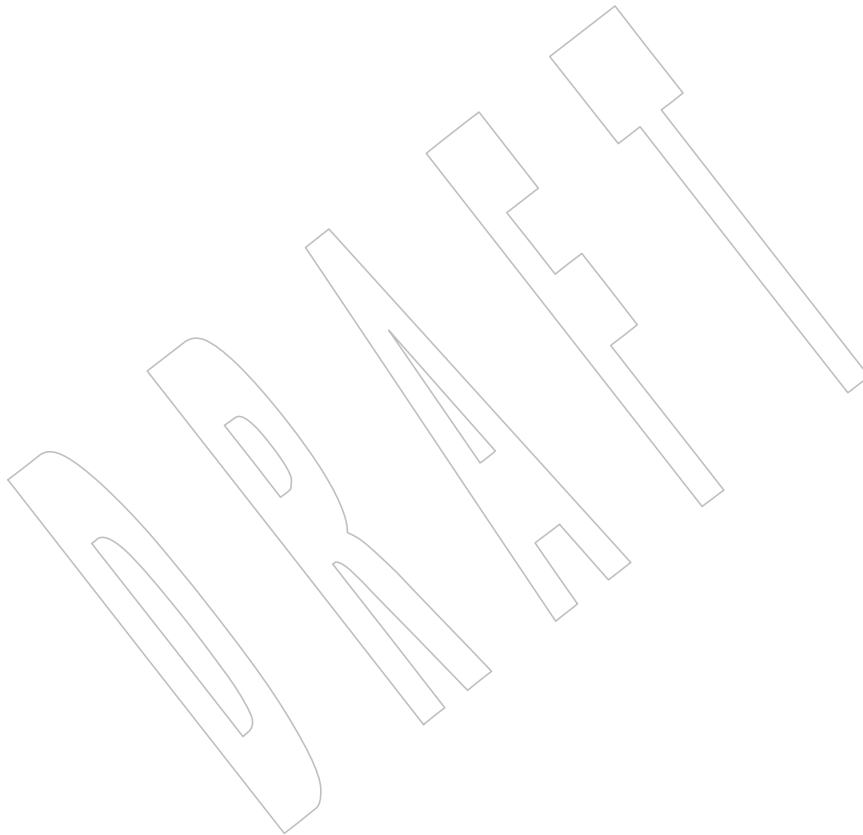
- The *wcstoull()* function is added.



52984 **NAME**
52985 `wcstoumax` — convert a wide-character string to an integer type

52986 **SYNOPSIS**
52987 `#include <stddef.h>`
52988 `#include <inttypes.h>`
52989 `uintmax_t wcstoumax(const wchar_t *restrict nptr,`
52990 `wchar_t **restrict endptr, int base);`

52991 **DESCRIPTION**
52992 Refer to [wcstoimax\(\)](#).



52993 **NAME**
 52994 `wcswidth` — number of column positions of a wide-character string

52995 **SYNOPSIS**

52996 XSI

```
#include <wchar.h>
```


 52997

```
int wcswidth(const wchar_t *pwcs, size_t n);
```

52998 **DESCRIPTION**

52999 The `wcswidth()` function shall determine the number of column positions required for n wide-
 53000 character codes (or fewer than n wide-character codes if a null wide-character code is
 53001 encountered before n wide-character codes are exhausted) in the string pointed to by `pwcs`.

53002 **RETURN VALUE**

53003 The `wcswidth()` function either shall return 0 (if `pwcs` points to a null wide-character code), or
 53004 return the number of column positions to be occupied by the wide-character string pointed to by
 53005 `pwcs`, or return -1 (if any of the first n wide-character codes in the wide-character string pointed
 53006 to by `pwcs` is not a printable wide-character code).

53007 **ERRORS**

53008 No errors are defined.

53009 **EXAMPLES**

53010 None.

53011 **APPLICATION USAGE**

53012 This function was removed from the final ISO/IEC 9899:1990/Amendment 1:1995 (E), and the
 53013 return value for a non-printable wide character is not specified.

53014 **RATIONALE**

53015 None.

53016 **FUTURE DIRECTIONS**

53017 None.

53018 **SEE ALSO**

53019 [wctype\(\)](#), the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.103, Column Position,
 53020 [<wchar.h>](#)

53021 **CHANGE HISTORY**

53022 First released in Issue 4. Derived from the MSE working draft.

53023 **Issue 6**

53024 The Open Group Corrigendum U021/11 is applied. The function is marked as an extension.

53025 **NAME**53026 `wcsxfrm`, `wcsxfrm_l` — wide-character string transformation53027 **SYNOPSIS**53028

```
#include <wchar.h>
```

53029

```
size_t wcsxfrm(wchar_t *restrict ws1, const wchar_t *restrict ws2,  
size_t n);
```

53031 CX

```
size_t wcsxfrm_l(wchar_t *restrict ws1, const wchar_t *restrict ws2,  
size_t n, locale_t locale);
```

53033 **DESCRIPTION**53034 CX For `wcsxfrm()`: The functionality described on this reference page is aligned with the ISO C
53035 standard. Any conflict between the requirements described here and the ISO C standard is
53036 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.53037 CX The `wcsxfrm()` and `wcsxfrm_l()` functions shall transform the wide-character string pointed to
53038 by `ws2` and place the resulting wide-character string into the array pointed to by `ws1`. The
53039 transformation shall be such that if `wcscmp()` is applied to two transformed wide strings, it shall
53040 return a value greater than, equal to, or less than 0, corresponding to the result of `wcscoll()` and
53041 `wcscoll_l()` applied to the same two original wide-character strings, and the same `LC_COLLATE`
53042 category of the locale of the process or the locale object `locale`, respectively. No more than `n`
53043 wide-character codes shall be placed into the resulting array pointed to by `ws1`, including the
53044 terminating null wide-character code. If `n` is 0, `ws1` is permitted to be a null pointer. If copying
53045 takes place between objects that overlap, the behavior is undefined.53046 CX The `wcsxfrm()` and `wcsxfrm_l()` functions shall not change the setting of `errno` if successful.53047 Since no return value is reserved to indicate an error, an application wishing to check for error
53048 situations should set `errno` to 0, then call `wcsxfrm()` or `wcsxfrm_l()`, then check `errno`.53049 **RETURN VALUE**53050 CX The `wcsxfrm()` and `wcsxfrm_l()` functions shall return the length of the transformed wide-
53051 character string (not including the terminating null wide-character code). If the value returned is
53052 `n` or more, the contents of the array pointed to by `ws1` are unspecified.53053 CX On error, the `wcsxfrm()` and `wcsxfrm_l()` functions may set `errno`, but no return value is reserved
53054 to indicate an error.53055 **ERRORS**

53056 These functions may fail if:

53057 CX [EINVAL] The wide-character string pointed to by `ws2` contains wide-character codes
53058 outside the domain of the collating sequence.53059 The `wcsxfrm_l()` function may fail if:53060 CX [EINVAL] `locale` is not a valid locale object handle.

53061
53062
53063
53064
53065
53066
53067
53068
53069
53070
53071
53072
53073
53074
53075
53076
53077
53078
53079
53080
53081
53082
53083
53084
53085
53086
53087
53088
53089
53090

EXAMPLES

None.

APPLICATION USAGE

The transformation function is such that two transformed wide-character strings can be ordered by *wcscmp()* as appropriate to collating sequence information in the locale of the process (category *LC_COLLATE*).

The fact that when *n* is 0 *ws1* is permitted to be a null pointer is useful to determine the size of the *ws1* array prior to making the transformation.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcscmp(), *wcscoll()*, the Base Definitions volume of IEEE Std 1003.1-200x, <*wchar.h*>

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

Issue 6

In previous versions, this function was required to return -1 on error.

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The *wcsxfrm()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

Issue 7

The *wcsxfrm_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

53091 **NAME**
 53092 `wctob` — wide-character to single-byte conversion

53093 **SYNOPSIS**
 53094 `#include <stdio.h>`
 53095 `#include <wchar.h>`
 53096 `int wctob(wint_t c);`

53097 **DESCRIPTION**
 53098 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 53099 conflict between the requirements described here and the ISO C standard is unintentional. This
 53100 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

53101 The `wctob()` function shall determine whether `c` corresponds to a member of the extended
 53102 character set whose character representation is a single byte when in the initial shift state.

53103 The behavior of this function shall be affected by the `LC_CTYPE` category of the current locale.

53104 **RETURN VALUE**
 53105 The `wctob()` function shall return EOF if `c` does not correspond to a character with length one in
 53106 the initial shift state. Otherwise, it shall return the single-byte representation of that character as
 53107 an **unsigned char** converted to **int**.

53108 **ERRORS**
 53109 No errors are defined.

53110 **EXAMPLES**
 53111 None.

53112 **APPLICATION USAGE**
 53113 None.

53114 **RATIONALE**
 53115 None.

53116 **FUTURE DIRECTIONS**
 53117 None.

53118 **SEE ALSO**
 53119 [*btowc\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, `<wchar.h>`

53120 **CHANGE HISTORY**
 53121 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
 53122 (E).

53123 **NAME**
 53124 `wctomb` — convert a wide-character code to a character

53125 **SYNOPSIS**
 53126 `#include <stdlib.h>`

53127 `int wctomb(char *s, wchar_t wchar);`

53128 DESCRIPTION

53129 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 53130 conflict between the requirements described here and the ISO C standard is unintentional. This
 53131 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

53132 The `wctomb()` function shall determine the number of bytes needed to represent the character
 53133 corresponding to the wide-character code whose value is `wchar` (including any change in the
 53134 shift state). It shall store the character representation (possibly multiple bytes and any special
 53135 bytes to change shift state) in the array object pointed to by `s` (if `s` is not a null pointer). At most
 53136 `{MB_CUR_MAX}` bytes shall be stored. If `wchar` is 0, a null byte shall be stored, preceded by any
 53137 shift sequence needed to restore the initial shift state, and `wctomb()` shall be left in the initial shift
 53138 state.

53139 CX The behavior of this function is affected by the `LC_CTYPE` category of the current locale. For a
 53140 state-dependent encoding, this function shall be placed into its initial state by a call for which its
 53141 character pointer argument, `s`, is a null pointer. Subsequent calls with `s` as other than a null
 53142 pointer shall cause the internal state of the function to be altered as necessary. A call with `s` as a
 53143 null pointer shall cause this function to return a non-zero value if encodings have state
 53144 dependency, and 0 otherwise. Changing the `LC_CTYPE` category causes the shift state of this
 53145 function to be unspecified.

53146 The `wctomb()` function need not be thread-safe. A function that is not required to be thread-safe
 53147 is not required to be reentrant.

53148 The implementation shall behave as if no function defined in this volume of
 53149 IEEE Std 1003.1-200x calls `wctomb()`.

53150 RETURN VALUE

53151 If `s` is a null pointer, `wctomb()` shall return a non-zero or 0 value, if character encodings,
 53152 respectively, do or do not have state-dependent encodings. If `s` is not a null pointer, `wctomb()`
 53153 shall return `-1` if the value of `wchar` does not correspond to a valid character, or return the
 53154 number of bytes that constitute the character corresponding to the value of `wchar`.

53155 In no case shall the value returned be greater than the value of the `{MB_CUR_MAX}` macro.

53156 ERRORS

53157 No errors are defined.

53158 EXAMPLES

53159 None.

53160 APPLICATION USAGE

53161 None.

53162 RATIONALE

53163 None.

wctomb()

53164

FUTURE DIRECTIONS

53165

None.

53166

SEE ALSO

53167

mblen(), *mbtowc()*, *mbstowcs()*, *wcstombs()*, the Base Definitions volume of IEEE Std 1003.1-200x,
<stdlib.h>

53168

53169

CHANGE HISTORY

53170

First released in Issue 4. Derived from the ANSI C standard.

53171

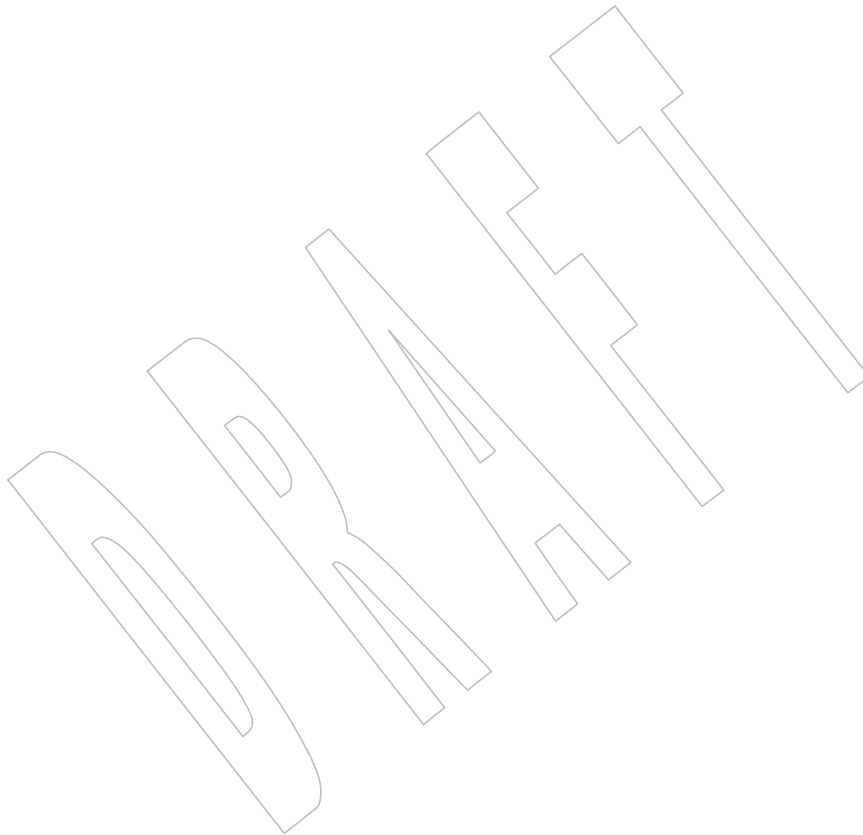
Issue 6

53172

Extensions beyond the ISO C standard are marked.

53173

In the DESCRIPTION, a note about reentrancy and thread-safety is added.



53174 **NAME**

53175 wctrans, wctrans_l — define character mapping

53176 **SYNOPSIS**

53177 #include <wctype.h>

53178 wctrans_t wctrans(const char *charclass);

53179 CX wctrans_t wctrans_l(const char *charclass, locale_t locale);

53180 **DESCRIPTION**53181 CX For *wctrans()*: The functionality described on this reference page is aligned with the ISO C
53182 standard. Any conflict between the requirements described here and the ISO C standard is
53183 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.53184 CX The *wctrans()* and *wctrans_l()* functions are defined for valid character mapping names
53185 identified in the current locale. The *charclass* is a string identifying a generic character mapping
53186 name for which codeset-specific information is required. The following character mapping
53187 names are defined in all locales: **tolower** and **toupper**.53188 These functions shall return a value of type **wctrans_t**, which can be used as the second
53189 argument to subsequent calls of *towctrans()* and *towctrans_l()*.53190 CX The *wctrans()* and *wctrans_l()* functions shall determine values of **wctrans_t** according to the
53191 rules of the coded character set defined by character mapping information in the locale of the
53192 process or in the locale represented by *locale*, respectively (category *LC_CTYPE*).53193 The values returned by *wctrans()* shall be valid until a call to *setlocale()* that modifies the
53194 category *LC_CTYPE*.53195 CX The values returned by *wctrans_l()* shall be valid only in calls to *wctrans_l()* with a locale
53196 represented by *locale* with the same *LC_CTYPE* category value.53197 **RETURN VALUE**53198 CX The *wctrans()* and *wctrans_l()* functions shall return 0 and may set *errno* to indicate the error if
53199 the given character mapping name is not valid for the current locale (category *LC_CTYPE*);
53200 otherwise, they shall return a non-zero object of type **wctrans_t** that can be used in calls to
53201 *towctrans()* and *towctrans_l()*.53202 **ERRORS**

53203 These functions may fail if:

53204 CX [EINVAL] The character mapping name pointed to by *charclass* is not valid in the current
53205 locale.53206 The *wctrans_l()* function may fail if:53207 CX [EINVAL] *locale* is not a valid locale object handle.53208 **EXAMPLES**

53209 None.

53210 **APPLICATION USAGE**

53211 None.

53212 **RATIONALE**

53213 None.

53214

FUTURE DIRECTIONS

53215

None.

53216

SEE ALSO

53217

towctrans(), the Base Definitions volume of IEEE Std 1003.1-200x, <wctype.h>

53218

CHANGE HISTORY

53219

First released in Issue 5. Derived from ISO/IEC 9899:1990/Amendment 1:1995 (E).

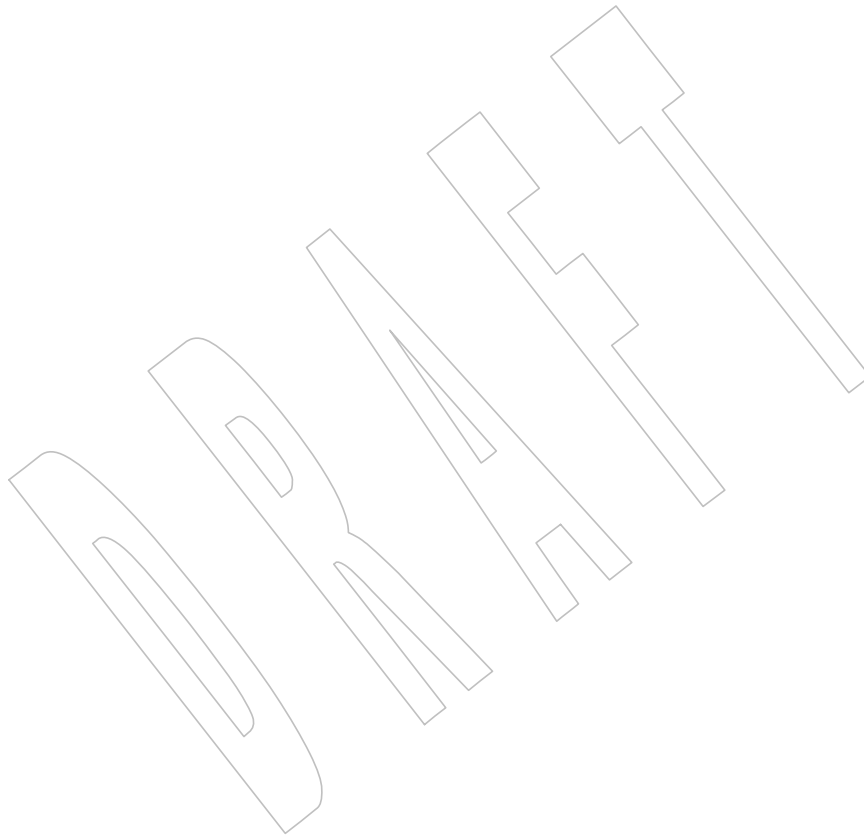
53220

Issue 7

53221

The *wctrans_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

53222



53223 **NAME**
 53224 `wctype, wctype_l` — define character class

53225 **SYNOPSIS**

53226 `#include <wctype.h>`
 53227 `wctype_t wctype(const char *property);`
 53228 CX `wctype_t wctype_l(const char *property, locale_t locale);`

53229 **DESCRIPTION**

53230 CX For `wctype()`: The functionality described on this reference page is aligned with the ISO C
 53231 standard. Any conflict between the requirements described here and the ISO C standard is
 53232 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

53233 CX The `wctype()` and `wctype_l()` functions are defined for valid character class names as defined in
 53234 CX the current locale or in the locale represented by `locale`, respectively.

53235 The `property` argument is a string identifying a generic character class for which codeset-specific
 53236 type information is required. The following character class names shall be defined in all locales:

53237	alnum	digit	punct
53238	alpha	graph	space
53239	blank	lower	upper
53240	cntrl	print	xdigit

53241 Additional character class names defined in the locale definition file (category `LC_CTYPE`) can
 53242 also be specified.

53243 These functions shall return a value of type `wctype_t`, which can be used as the second
 53244 CX argument to subsequent calls of `iswctype()` and `iswctype_l()`.

53245 CX The `wctype()` and `wctype_l()` functions shall determine values of `wctype_t` according to the
 53246 rules of the coded character set defined by character type information in the locale of the process
 53247 CX or in the locale represented by `locale`, respectively (category `LC_CTYPE`).

53248 The values returned by `wctype()` shall be valid until a call to `setlocale()` that modifies the category
 53249 `LC_CTYPE`.

53250 CX The values returned by `wctype_l()` shall be valid only in calls to `iswctype_l()` with a locale
 53251 represented by `locale` with the same `LC_CTYPE` category value.

53252 **RETURN VALUE**

53253 CX The `wctype()` and `wctype_l()` functions shall return 0 if the given character class name is not
 53254 valid for the current locale (category `LC_CTYPE`); otherwise, they shall return an object of type
 53255 CX `wctype_t` that can be used in calls to `iswctype()` and `iswctype_l()`.

53256 **ERRORS**

53257 The `wctype_l()` function may fail if:

53258 CX [EINVAL] `locale` is not a valid locale object handle.

53259

EXAMPLES

53260

None.

53261

APPLICATION USAGE

53262

None.

53263

RATIONALE

53264

None.

53265

FUTURE DIRECTIONS

53266

None.

53267

SEE ALSO

53268

iswctype(), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**

53269

CHANGE HISTORY

53270

First released in Issue 4.

53271

Issue 5

53272

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

53273

53274

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

53275

53276

Issue 7

53277

The *wctype_l()* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

53278

53279 **NAME**
 53280 `wcwidth` — number of column positions of a wide-character code

53281 **SYNOPSIS**

```
53282 XSI #include <wchar.h>
53283 int wcwidth(wchar_t wc);
```

53284 **DESCRIPTION**

53285 The `wcwidth()` function shall determine the number of column positions required for the wide
 53286 character `wc`. The application shall ensure that the value of `wc` is a character representable as a
 53287 **wchar_t**, and is a wide-character code corresponding to a valid character in the current locale.

53288 **RETURN VALUE**

53289 The `wcwidth()` function shall either return 0 (if `wc` is a null wide-character code), or return the
 53290 number of column positions to be occupied by the wide-character code `wc`, or return -1 (if `wc`
 53291 does not correspond to a printable wide-character code).

53292 **ERRORS**

53293 No errors are defined.

53294 **EXAMPLES**

53295 None.

53296 **APPLICATION USAGE**

53297 This function was removed from the final ISO/IEC 9899:1990/Amendment 1:1995 (E), and the
 53298 return value for a non-printable wide character is not specified.

53299 **RATIONALE**

53300 None.

53301 **FUTURE DIRECTIONS**

53302 None.

53303 **SEE ALSO**

53304 [*wcswidth\(\)*](#), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

53305 **CHANGE HISTORY**

53306 First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working
 53307 draft.

53308 **Issue 6**

53309 The Open Group Corrigendum U021/12 is applied. This function is marked as an extension.

53310 The normative text is updated to avoid use of the term “must” for application requirements.

53311 **NAME**

53312 wmemchr — find a wide character in memory

53313 **SYNOPSIS**

53314 #include <wchar.h>

53315 wchar_t *wmemchr(const wchar_t *ws, wchar_t wc, size_t n);

53316 **DESCRIPTION**53317 CX The functionality described on this reference page is aligned with the ISO C standard. Any
53318 conflict between the requirements described here and the ISO C standard is unintentional. This
53319 volume of IEEE Std 1003.1-200x defers to the ISO C standard.53320 The *wmemchr()* function shall locate the first occurrence of *wc* in the initial *n* wide characters of
53321 the object pointed to by *ws*. This function shall not be affected by locale and all **wchar_t** values
53322 shall be treated identically. The null wide character and **wchar_t** values not corresponding to
53323 valid characters shall not be treated specially.53324 If *n* is zero, the application shall ensure that *ws* is a valid pointer and the function behaves as if
53325 no valid occurrence of *wc* is found.53326 **RETURN VALUE**53327 The *wmemchr()* function shall return a pointer to the located wide character, or a null pointer if
53328 the wide character does not occur in the object.53329 **ERRORS**

53330 No errors are defined.

53331 **EXAMPLES**

53332 None.

53333 **APPLICATION USAGE**

53334 None.

53335 **RATIONALE**

53336 None.

53337 **FUTURE DIRECTIONS**

53338 None.

53339 **SEE ALSO**53340 *wmemcmp()*, *wmemcpy()*, *wmemmove()*, *wmemset()*, the Base Definitions volume of
53341 IEEE Std 1003.1-200x, <wchar.h>53342 **CHANGE HISTORY**53343 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
53344 (E).53345 **Issue 6**

53346 The normative text is updated to avoid use of the term “must” for application requirements.

53347 **NAME**
 53348 wmemcmp — compare wide characters in memory

53349 **SYNOPSIS**
 53350 #include <wchar.h>
 53351 int wmemcmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);

53352 **DESCRIPTION**
 53353 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 53354 conflict between the requirements described here and the ISO C standard is unintentional. This
 53355 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

53356 The *wmemcmp()* function shall compare the first *n* wide characters of the object pointed to by
 53357 *ws1* to the first *n* wide characters of the object pointed to by *ws2*. This function shall not be
 53358 affected by locale and all **wchar_t** values shall be treated identically. The null wide character and
 53359 **wchar_t** values not corresponding to valid characters shall not be treated specially.

53360 If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function
 53361 shall behave as if the two objects compare equal.

53362 **RETURN VALUE**
 53363 The *wmemcmp()* function shall return an integer greater than, equal to, or less than zero,
 53364 respectively, as the object pointed to by *ws1* is greater than, equal to, or less than the object
 53365 pointed to by *ws2*.

53366 **ERRORS**
 53367 No errors are defined.

53368 **EXAMPLES**
 53369 None.

53370 **APPLICATION USAGE**
 53371 None.

53372 **RATIONALE**
 53373 None.

53374 **FUTURE DIRECTIONS**
 53375 None.

53376 **SEE ALSO**
 53377 *wmemchr()*, *wmemcpy()*, *wmemmove()*, *wmemset()*, the Base Definitions volume of
 53378 IEEE Std 1003.1-200x, <wchar.h>

53379 **CHANGE HISTORY**
 53380 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
 53381 (E).

53382 **Issue 6**
 53383 The normative text is updated to avoid use of the term “must” for application requirements.

53384 **NAME**
 53385 `wmemcpy` — copy wide characters in memory

53386 **SYNOPSIS**
 53387 `#include <wchar.h>`

53388 `wchar_t *wmemcpy(wchar_t *restrict ws1, const wchar_t *restrict ws2,`
 53389 `size_t n);`

53390 **DESCRIPTION**

53391 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 53392 conflict between the requirements described here and the ISO C standard is unintentional. This
 53393 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

53394 The `wmemcpy()` function shall copy *n* wide characters from the object pointed to by *ws2* to the
 53395 object pointed to by *ws1*. This function shall not be affected by locale and all **wchar_t** values
 53396 shall be treated identically. The null wide character and **wchar_t** values not corresponding to
 53397 valid characters shall not be treated specially.

53398 If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function
 53399 shall copy zero wide characters.

53400 **RETURN VALUE**

53401 The `wmemcpy()` function shall return the value of *ws1*.

53402 **ERRORS**

53403 No errors are defined.

53404 **EXAMPLES**

53405 None.

53406 **APPLICATION USAGE**

53407 None.

53408 **RATIONALE**

53409 None.

53410 **FUTURE DIRECTIONS**

53411 None.

53412 **SEE ALSO**

53413 [wmemchr\(\)](#), [wmemcpy\(\)](#), [wmemmove\(\)](#), [wmemset\(\)](#), the Base Definitions volume of
 53414 IEEE Std 1003.1-200x, [<wchar.h>](#)

53415 **CHANGE HISTORY**

53416 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
 53417 (E).

53418 **Issue 6**

53419 The normative text is updated to avoid use of the term “must” for application requirements.

53420 The `wmemcpy()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wmemmove — copy wide characters in memory with overlapping areas

SYNOPSIS

```
#include <wchar.h>
```

```
wchar_t *wmemmove(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wmemmove()* function shall copy *n* wide characters from the object pointed to by *ws2* to the object pointed to by *ws1*. Copying shall take place as if the *n* wide characters from the object pointed to by *ws2* are first copied into a temporary array of *n* wide characters that does not overlap the objects pointed to by *ws1* or *ws2*, and then the *n* wide characters from the temporary array are copied into the object pointed to by *ws1*.

This function shall not be affected by locale and all **wchar_t** values shall be treated identically. The null wide character and **wchar_t** values not corresponding to valid characters shall not be treated specially.

If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function shall copy zero wide characters.

RETURN VALUE

The *wmemmove()* function shall return the value of *ws1*.

ERRORS

No errors are defined

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wmemchr(), *wmemcmp()*, *wmemcpy()*, *wmemset()*, *t*,

53460 **NAME**53461 `wmemset` — set wide characters in memory53462 **SYNOPSIS**53463 `#include <wchar.h>`53464 `wchar_t *wmemset(wchar_t *ws, wchar_t wc, size_t n);`53465 **DESCRIPTION**53466 CX The functionality described on this reference page is aligned with the ISO C standard. Any
53467 conflict between the requirements described here and the ISO C standard is unintentional. This
53468 volume of IEEE Std 1003.1-200x defers to the ISO C standard.53469 The `wmemset()` function shall copy the value of `wc` into each of the first `n` wide characters of the
53470 object pointed to by `ws`. This function shall not be affected by locale and all `wchar_t` values shall
53471 be treated identically. The null wide character and `wchar_t` values not corresponding to valid
53472 characters shall not be treated specially.53473 If `n` is zero, the application shall ensure that `ws` is a valid pointer, and the function shall copy
53474 zero wide characters.53475 **RETURN VALUE**53476 The `wmemset()` functions shall return the value of `ws`.53477 **ERRORS**

53478 No errors are defined.

53479 **EXAMPLES**

53480 None.

53481 **APPLICATION USAGE**

53482 None.

53483 **RATIONALE**

53484 None.

53485 **FUTURE DIRECTIONS**

53486 None.

53487 **SEE ALSO**53488 [wmemchr\(\)](#), [wmemcmp\(\)](#), [wmemcpy\(\)](#), [wmemmove\(\)](#), the Base Definitions volume of
53489 IEEE Std 1003.1-200x, [<wchar.h>](#)53490 **CHANGE HISTORY**53491 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
53492 (E).53493 **Issue 6**

53494 The normative text is updated to avoid use of the term “must” for application requirements.

53495 **NAME**
 53496 wordexp, wordfree — perform word expansions

53497 **SYNOPSIS**
 53498 #include <wordexp.h>
 53499 int wordexp(const char *restrict words, wordexp_t *restrict pwordexp,
 53500 int flags);
 53501 void wordfree(wordexp_t *pwordexp);

53502 **DESCRIPTION**
 53503 The *wordexp()* function shall perform word expansions as described in the Shell and Utilities
 53504 volume of IEEE Std 1003.1-200x, Section 2.6, Word Expansions, subject to quoting as in the Shell
 53505 and Utilities volume of IEEE Std 1003.1-200x, Section 2.2, Quoting, and place the list of
 53506 expanded words into the structure pointed to by *pwordexp*.

53507 The *words* argument is a pointer to a string containing one or more words to be expanded. The
 53508 expansions shall be the same as would be performed by the command line interpreter if *words*
 53509 were the part of a command line representing the arguments to a utility. Therefore, the
 53510 application shall ensure that *words* does not contain an unquoted <newline> or any of the
 53511 unquoted shell special characters '|', '&', ';', '<', '>' except in the context of command
 53512 substitution as specified in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.3,
 53513 Command Substitution. It also shall not contain unquoted parentheses or braces, except in the
 53514 context of command or variable substitution. The application shall ensure that every member of
 53515 *words* which it expects to have expanded by *wordexp()* does not contain an unquoted initial
 53516 comment character. The application shall also ensure that any words which it intends to be
 53517 ignored (because they begin or continue a comment) are deleted from *words*. If the argument
 53518 *words* contains an unquoted comment character (number sign) that is the beginning of a token,
 53519 *wordexp()* shall either treat the comment character as a regular character, or interpret it as a
 53520 comment indicator and ignore the remainder of *words*.

53521 The structure type **wordexp_t** is defined in the **<wordexp.h>** header and includes at least the
 53522 following members:

Member Type	Member Name	Description
size_t	<i>we_wordc</i>	Count of words matched by <i>words</i> .
char **	<i>we_wordv</i>	Pointer to list of expanded words.
size_t	<i>we_offs</i>	Slots to reserve at the beginning of <i>pwordexp->we_wordv</i> .

53527 The *wordexp()* function shall store the number of generated words into *pwordexp->we_wordc* and
 53528 a pointer to a list of pointers to words in *pwordexp->we_wordv*. Each individual field created
 53529 during field splitting (see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.5,
 53530 Field Splitting) or pathname expansion (see the Shell and Utilities volume of
 53531 IEEE Std 1003.1-200x, Section 2.6.6, Pathname Expansion) shall be a separate word in the
 53532 *pwordexp->we_wordv* list. The words shall be in order as described in the Shell and Utilities
 53533 volume of IEEE Std 1003.1-200x, Section 2.6, Word Expansions. The first pointer after the last
 53534 word pointer shall be a null pointer. The expansion of special parameters described in the Shell
 53535 and Utilities volume of IEEE Std 1003.1-200x, Section 2.5.2, Special Parameters is unspecified.

53536 It is the caller's responsibility to allocate the storage pointed to by *pwordexp*. The *wordexp()*
 53537 function shall allocate other space as needed, including memory pointed to by
 53538 *pwordexp->we_wordv*. The *wordfree()* function frees any memory associated with *pwordexp* from a
 53539 previous call to *wordexp()*.

53540 The *flags* argument is used to control the behavior of *wordexp()*. The value of *flags* is the bitwise-
 53541 inclusive OR of zero or more of the following constants, which are defined in **<wordexp.h>**:

53542	WRDE_APPEND	Append words generated to the ones from a previous call to <i>wordexp()</i> .
53543	WRDE_DOOFFS	Make use of <i>pwordexp->we_offs</i> . If this flag is set, <i>pwordexp->we_offs</i> is used to specify how many null pointers to add to the beginning of <i>pwordexp->we_wordv</i> . In other words, <i>pwordexp->we_wordv</i> shall point to <i>pwordexp->we_offs</i> null pointers, followed by <i>pwordexp->we_wordc</i> word pointers, followed by a null pointer.
53544		
53545		
53546		
53547		
53548	WRDE_NOCMD	If the implementation supports the utilities defined in the Shell and Utilities volume of IEEE Std 1003.1-200x, fail if command substitution, as specified in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.3, Command Substitution, is requested.
53549		
53550		
53551		
53552	WRDE_REUSE	The <i>pwordexp</i> argument was passed to a previous successful call to <i>wordexp()</i> , and has not been passed to <i>wordfree()</i> . The result shall be the same as if the application had called <i>wordfree()</i> and then called <i>wordexp()</i> without WRDE_REUSE.
53553		
53554		
53555		
53556	WRDE_SHOWERR	Do not redirect <i>stderr</i> to /dev/null .
53557	WRDE_UNDEF	Report error on an attempt to expand an undefined shell variable.
53558		The WRDE_APPEND flag can be used to append a new set of words to those generated by a previous call to <i>wordexp()</i> . The following rules apply to applications when two or more calls to <i>wordexp()</i> are made with the same value of <i>pwordexp</i> and without intervening calls to <i>wordfree()</i> :
53559		
53560		
53561		1. The first such call shall not set WRDE_APPEND. All subsequent calls shall set it.
53562		2. All of the calls shall set WRDE_DOOFFS, or all shall not set it.
53563		3. After the second and each subsequent call, <i>pwordexp->we_wordv</i> shall point to a list containing the following:
53564		
53565		a. Zero or more null pointers, as specified by WRDE_DOOFFS and <i>pwordexp->we_offs</i>
53566		
53567		b. Pointers to the words that were in the <i>pwordexp->we_wordv</i> list before the call, in the same order as before
53568		
53569		c. Pointers to the new words generated by the latest call, in the specified order
53570		4. The count returned in <i>pwordexp->we_wordc</i> shall be the total number of words from all of the calls.
53571		
53572		5. The application can change any of the fields after a call to <i>wordexp()</i> , but if it does it shall reset them to the original value before a subsequent call, using the same <i>pwordexp</i> value, to <i>wordfree()</i> or <i>wordexp()</i> with the WRDE_APPEND or WRDE_REUSE flag.
53573		
53574		
53575		If the implementation supports the utilities defined in the Shell and Utilities volume of IEEE Std 1003.1-200x, and <i>words</i> contains an unquoted character—<newline>, ' ', '&', ';', '<', '>', '(', ')', '{', '}'—in an inappropriate context, <i>wordexp()</i> shall fail, and the number of expanded words shall be 0.
53576		
53577		
53578		
53579		Unless WRDE_SHOWERR is set in <i>flags</i> , <i>wordexp()</i> shall redirect <i>stderr</i> to /dev/null for any utilities executed as a result of command substitution while expanding <i>words</i> . If WRDE_SHOWERR is set, <i>wordexp()</i> may write messages to <i>stderr</i> if syntax errors are detected while expanding <i>words</i> .
53580		
53581		
53582		
53583		The application shall ensure that if WRDE_DOOFFS is set, then <i>pwordexp->we_offs</i> has the same value for each <i>wordexp()</i> call and <i>wordfree()</i> call using a given <i>pwordexp</i> .
53584		
53585		The following constants are defined as error return values:

53586	WRDE_BADCHAR	One of the unquoted characters—<newline>, ' ', '&', ';', '<', '>', '(', ')', '{', '}'—appears in <i>words</i> in an inappropriate context.
53588	WRDE_BADVAL	Reference to undefined shell variable when WRDE_UNDEF is set in <i>flags</i> .
53589	WRDE_CMDSUB	Command substitution requested when WRDE_NOCMD was set in <i>flags</i> .
53590	WRDE_NOSPACE	Attempt to allocate memory failed.
53591	WRDE_SYNTAX	Shell syntax error, such as unbalanced parentheses or unterminated string.

RETURN VALUE

Upon successful completion, *wordexp()* shall return 0. Otherwise, a non-zero value, as described in <**wordexp.h**>, shall be returned to indicate an error. If *wordexp()* returns the value WRDE_NOSPACE, then *pwordexp*→*we_wordc* and *pwordexp*→*we_wordv* shall be updated to reflect any words that were successfully expanded. In other cases, they shall not be modified.

The *wordfree()* function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *wordexp()* function is intended to be used by an application that wants to do all of the shell's expansions on a word or words obtained from a user. For example, if the application prompts for a filename (or list of filenames) and then uses *wordexp()* to process the input, the user could respond with anything that would be valid as input to the shell.

The WRDE_NOCMD flag is provided for applications that, for security or other reasons, want to prevent a user from executing shell commands. Disallowing unquoted shell special characters also prevents unwanted side effects, such as executing a command or writing a file.

RATIONALE

This function was included as an alternative to *glob()*. There had been continuing controversy over exactly what features should be included in *glob()*. It is hoped that by providing *wordexp()* (which provides all of the shell word expansions, but which may be slow to execute) and *glob()* (which is faster, but which only performs pathname expansion, without tilde or parameter expansion) this will satisfy the majority of applications.

While *wordexp()* could be implemented entirely as a library routine, it is expected that most implementations run a shell in a subprocess to do the expansion.

Two different approaches have been proposed for how the required information might be presented to the shell and the results returned. They are presented here as examples.

One proposal is to extend the *echo* utility by adding a **-q** option. This option would cause *echo* to add a backslash before each backslash and <blank> that occurs within an argument. The *wordexp()* function could then invoke the shell as follows:

```
(void) strcpy(buffer, "echo -q");
(void) strcat(buffer, words);
if ((flags & WRDE_SHOWERR) == 0)
    (void) strcat(buffer, "2>/dev/null");
f = popen(buffer, "r");
```

The *wordexp()* function would read the resulting output, remove unquoted backslashes, and break into words at unquoted <blank>s. If the WRDE_NOCMD flag was set, *wordexp()* would have to scan *words* before starting the subshell to make sure that there would be no command

53632 substitution. In any case, it would have to scan *words* for unquoted special characters.

53633 Another proposal is to add the following options to *sh*:

53634 **-w** *wordlist*

53635 This option provides a wordlist expansion service to applications. The words in *wordlist*
53636 shall be expanded and the following written to standard output:

- 53637 1. The count of the number of words after expansion, in decimal, followed by a null
53638 byte
- 53639 2. The number of bytes needed to represent the expanded words (not including null
53640 separators), in decimal, followed by a null byte
- 53641 3. The expanded words, each terminated by a null byte

53642 If an error is encountered during word expansion, *sh* exits with a non-zero status after
53643 writing the former to report any words successfully expanded

53644 **-P** Run in “protected” mode. If specified with the **-w** option, no command substitution shall
53645 be performed.

53646 With these options, *wordexp()* could be implemented fairly simply by creating a subprocess
53647 using *fork()* and executing *sh* using the line:

53648 `execl(<shell path>, "sh", "-P", "-w", words, (char *)0);`

53649 after directing standard error to **/dev/null**.

53650 It seemed objectionable for a library routine to write messages to standard error, unless explicitly
53651 requested, so *wordexp()* is required to redirect standard error to **/dev/null** to ensure that no
53652 messages are generated, even for commands executed for command substitution. The
53653 **WRDE_SHOWERR** flag can be specified to request that error messages be written.

53654 The **WRDE_REUSE** flag allows the implementation to avoid the expense of freeing and
53655 reallocating memory, if that is possible. A minimal implementation can call *wordfree()* when
53656 **WRDE_REUSE** is set.

53657 **FUTURE DIRECTIONS**

53658 None.

53659 **SEE ALSO**

53660 *fnmatch()*, *glob()*, the Base Definitions volume of IEEE Std 1003.1-200x, **<wordexp.h>**, the Shell
53661 and Utilities volume of IEEE Std 1003.1-200x, Chapter 2, Shell Command Language

53662 **CHANGE HISTORY**

53663 First released in Issue 4. Derived from the ISO POSIX-2 standard.

53664 **Issue 5**

53665 Moved from POSIX2 C-language Binding to BASE.

53666 **Issue 6**

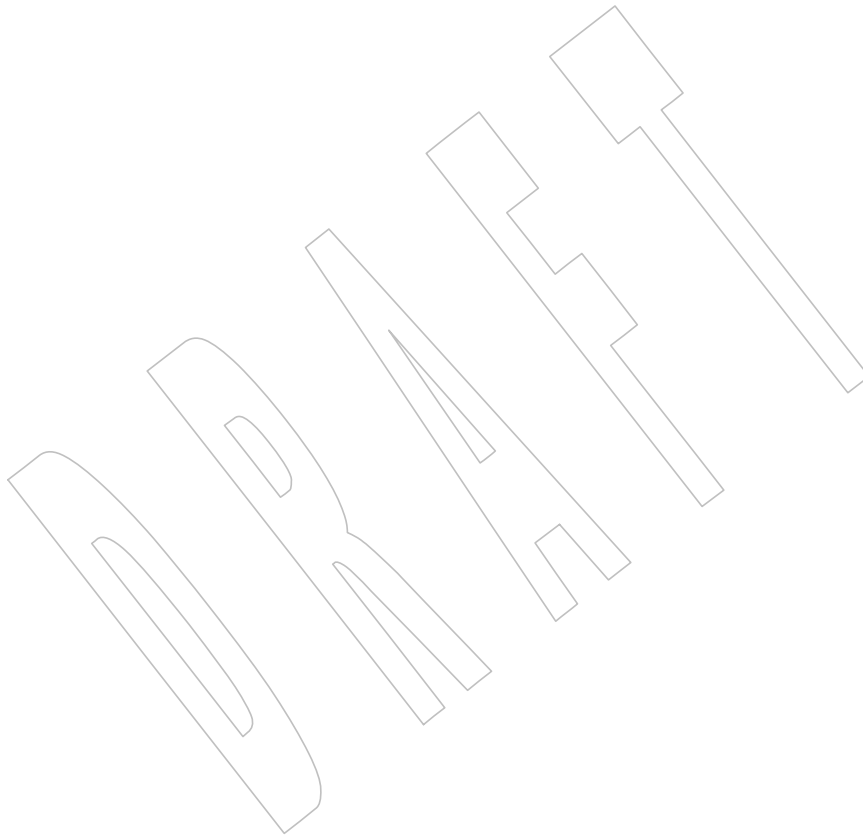
53667 The normative text is updated to avoid use of the term “must” for application requirements.

53668 The **restrict** keyword is added to the *wordexp()* prototype for alignment with the
53669 ISO/IEC 9899:1999 standard.

53670 **NAME**
53671 `wprintf` — print formatted wide-character output

53672 **SYNOPSIS**
53673 `#include <stdio.h>`
53674 `#include <wchar.h>`
53675 `int wprintf(const wchar_t *restrict format, ...);`

53676 **DESCRIPTION**
53677 Refer to *fwprintf()*.



53678 **NAME**

53679 pwrite, write — write on a file

53680 **SYNOPSIS**

```
53681 #include <unistd.h>
53682
53682 ssize_t pwrite(int fildes, const void *buf, size_t nbyte,
53683               off_t offset);
53684
53684 ssize_t write(int fildes, const void *buf, size_t nbyte);
```

53685 **DESCRIPTION**

53686 The *write()* function shall attempt to write *nbyte* bytes from the buffer pointed to by *buf* to the
 53687 file associated with the open file descriptor, *fildes*.

53688 Before any action described below is taken, and if *nbyte* is zero and the file is a regular file, the
 53689 *write()* function may detect and return errors as described below. In the absence of errors, or if
 53690 error detection is not performed, the *write()* function shall return zero and have no other results.
 53691 If *nbyte* is zero and the file is not a regular file, the results are unspecified.

53692 On a regular file or other file capable of seeking, the actual writing of data shall proceed from
 53693 the position in the file indicated by the file offset associated with *fildes*. Before successful return
 53694 from *write()*, the file offset shall be incremented by the number of bytes actually written. On a
 53695 regular file, if the position of the last byte written is greater than or equal to the length of the file,
 53696 the length of the file shall be set to this position plus one.

53697 On a file not capable of seeking, writing shall always take place starting at the current position.
 53698 The value of a file offset associated with such a device is undefined.

53699 If the `O_APPEND` flag of the file status flags is set, the file offset shall be set to the end of the file
 53700 prior to each write and no intervening file modification operation shall occur between changing
 53701 the file offset and the write operation.

53702 XSI If a *write()* requests that more bytes be written than there is room for (for example, the file size
 53703 limit of the process or the physical end of a medium), only as many bytes as there is room for
 53704 shall be written. For example, suppose there is space for 20 bytes more in a file before reaching a
 53705 limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes would
 53706 give a failure return (except as noted below).

53707 XSI If the request would cause the file size to exceed the soft file size limit for the process and there
 53708 is no room for any bytes to be written, the request shall fail and the implementation shall
 53709 generate the SIGXFSZ signal for the thread.

53710 If *write()* is interrupted by a signal before it writes any data, it shall return `-1` with *errno* set to
 53711 `[EINTR]`.

53712 If *write()* is interrupted by a signal after it successfully writes some data, it shall return the
 53713 number of bytes written.

53714 If the value of *nbyte* is greater than `{SSIZE_MAX}`, the result is implementation-defined.

53715 After a *write()* to a regular file has successfully returned:

- 53716 • Any successful *read()* from each byte position in the file that was modified by that write
 53717 shall return the data specified by the *write()* for that position until such byte positions are
 53718 again modified.
- 53719 • Any subsequent successful *write()* to the same byte position in the file shall overwrite that
 53720 file data.

53721 Write requests to a pipe or FIFO shall be handled in the same way as a regular file with the

53722

following exceptions:

53723

- There is no file offset associated with a pipe, hence each write request shall append to the end of the pipe.

53724

53725

- Write requests of {PIPE_BUF} bytes or less shall not be interleaved with data from other processes doing writes on the same pipe. Writes of greater than {PIPE_BUF} bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the O_NONBLOCK flag of the file status flags is set.

53726

53727

53728

53729

- If the O_NONBLOCK flag is clear, a write request may cause the thread to block, but on normal completion it shall return *nbyte*.

53730

53731

- If the O_NONBLOCK flag is set, *write()* requests shall be handled differently, in the following ways:

53732

53733

- The *write()* function shall not block the thread.

53734

- A write request for {PIPE_BUF} or fewer bytes shall have the following effect: if there is sufficient space available in the pipe, *write()* shall transfer all the data and return the number of bytes requested. Otherwise, *write()* shall transfer no data and return -1 with *errno* set to [EAGAIN].

53735

53736

53737

53738

- A write request for more than {PIPE_BUF} bytes shall cause one of the following:

53739

- When at least one byte can be written, transfer what it can and return the number of bytes written. When all data previously written to the pipe is read, it shall transfer at least {PIPE_BUF} bytes.

53740

53741

53742

- When no data can be written, transfer no data, and return -1 with *errno* set to [EAGAIN].

53743

53744

When attempting to write to a file descriptor (other than a pipe or FIFO) that supports non-blocking writes and cannot accept the data immediately:

53745

53746

- If the O_NONBLOCK flag is clear, *write()* shall block the calling thread until the data can be accepted.

53747

53748

- If the O_NONBLOCK flag is set, *write()* shall not block the thread. If some data can be written without blocking the thread, *write()* shall write what it can and return the number of bytes written. Otherwise, it shall return -1 and set *errno* to [EAGAIN].

53749

53750

53751

Upon successful completion, where *nbyte* is greater than 0, *write()* shall mark for update the *st_ctime* and *st_mtime* fields of the file, and if the file is a regular file, the S_ISUID and S_ISGID bits of the file mode may be cleared.

53752

53753

53754

For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with *fildev*.

53755

53756

If *fildev* refers to a socket, *write()* shall be equivalent to *send()* with no flags set.

53757

SIO

If the O_DSYNC bit has been set, write I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion.

53758

53759

If the O_SYNC bit has been set, write I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.

53760

53761

SHM

If *fildev* refers to a shared memory object, the result of the *write()* function is unspecified.

53762

TYM

If *fildev* refers to a typed memory object, the result of the *write()* function is unspecified.

53763

OB XSR

If *fildev* refers to a STREAM, the operation of *write()* shall be determined by the values of the minimum and maximum *nbyte* range (packet size) accepted by the STREAM. These values are determined by the topmost STREAM module. If *nbyte* falls within the packet size range, *nbyte*

53764

53765

53766 bytes shall be written. If *nbyte* does not fall within the range and the minimum packet size value
 53767 is 0, *write()* shall break the buffer into maximum packet size segments prior to sending the data
 53768 downstream (the last segment may contain less than the maximum packet size). If *nbyte* does not
 53769 fall within the range and the minimum value is non-zero, *write()* shall fail with *errno* set to
 53770 [ERANGE]. Writing a zero-length buffer (*nbyte* is 0) to a STREAMS device sends 0 bytes with 0
 53771 returned. However, writing a zero-length buffer to a STREAMS-based pipe or FIFO sends no
 53772 message and 0 is returned. The process may issue `I_SWROPT ioctl()` to enable zero-length
 53773 messages to be sent across the pipe or FIFO.

53774 When writing to a STREAM, data messages are created with a priority band of 0. When writing
 53775 to a STREAM that is not a pipe or FIFO:

- 53776 • If `O_NONBLOCK` is clear, and the STREAM cannot accept data (the STREAM write queue
 53777 is full due to internal flow control conditions), *write()* shall block until data can be
 53778 accepted.
- 53779 • If `O_NONBLOCK` is set and the STREAM cannot accept data, *write()* shall return `-1` and
 53780 set *errno* to [EAGAIN].
- 53781 • If `O_NONBLOCK` is set and part of the buffer has been written while a condition in which
 53782 the STREAM cannot accept additional data occurs, *write()* shall terminate and return the
 53783 number of bytes written.

53784 In addition, *write()* shall fail if the STREAM head has processed an asynchronous error before
 53785 the call. In this case, the value of *errno* does not reflect the result of *write()*, but reflects the prior
 53786 error.

53787 The *pwrite()* function shall be equivalent to *write()*, except that it writes into a given position
 53788 and does not change the file offset (regardless of whether `O_APPEND` is set). The first three
 53789 arguments to *pwrite()* are the same as *write()* with the addition of a fourth argument *offset* for
 53790 the desired position inside the file.

53791 RETURN VALUE

53792 Upon successful completion, these functions shall return the number of bytes actually written to
 53793 the file associated with *fildev*. This number shall never be greater than *nbyte*. Otherwise, `-1` shall
 53794 be returned and *errno* set to indicate the error.

53795 ERRORS

53796 These functions shall fail if:

- | | | |
|----------------------------------|----------|---|
| 53797
53798 | [EAGAIN] | The <code>O_NONBLOCK</code> flag is set for the file descriptor and the thread would be delayed in the <i>write()</i> operation. |
| 53799 | [EBADF] | The <i>fildev</i> argument is not a valid file descriptor open for writing. |
| 53800
53801
53802 | [EFBIG] | An attempt was made to write a file that exceeds the implementation-defined maximum file size or the file size limit of the process, and there was no room for any bytes to be written. |
| 53803
53804
53805 | [EFBIG] | The file is a regular file, <i>nbyte</i> is greater than 0, and the starting position is greater than or equal to the offset maximum established in the open file description associated with <i>fildev</i> . |
| 53806
53807 | [EINTR] | The write operation was terminated due to the receipt of a signal, and no data was transferred. |
| 53808
53809
53810
53811 | [EIO] | The process is a member of a background process group attempting to write to its controlling terminal, <code>TOSTOP</code> is set, the process is neither ignoring nor blocking <code>SIGTTOU</code> , and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions. |

- 53812 [ENOSPC] There was no free space remaining on the device containing the file.
- 53813 [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
53814 any process, or that only has one end open. A SIGPIPE signal shall also be sent
53815 to the thread.
- 53816 OB XSR [ERANGE] The transfer request size was outside the range supported by the STREAMS
53817 file associated with *fildev*.
- 53818 The *write()* function shall fail if:
- 53819 [EAGAIN] or [EWOULDBLOCK]
53820 The file descriptor is for a socket, is marked O_NONBLOCK, and write would
53821 block.
- 53822 [ECONNRESET] A write was attempted on a socket that is not connected.
- 53823 [EPIPE] A write was attempted on a socket that is shut down for writing, or is no
53824 longer connected. In the latter case, if the socket is of type SOCK_STREAM, a
53825 SIGPIPE signal shall also be sent to the thread.
- 53826 These functions may fail if:
- 53827 OB XSR [EINVAL] The STREAM or multiplexer referenced by *fildev* is linked (directly or
53828 indirectly) downstream from a multiplexer.
- 53829 [EIO] A physical I/O error has occurred.
- 53830 [ENOBUFS] Insufficient resources were available in the system to perform the operation.
- 53831 [ENXIO] A request was made of a nonexistent device, or the request was outside the
53832 capabilities of the device.
- 53833 OB XSR [ENXIO] A hangup occurred on the STREAM being written to.
- 53834 OB XSR A write to a STREAMS file may fail if an error message has been received at the STREAM head.
53835 In this case, *errno* is set to the value included in the error message.
- 53836 The *write()* function may fail if:
- 53837 [EACCES] A write was attempted on a socket and the calling process does not have
53838 appropriate privileges.
- 53839 [ENETDOWN] A write was attempted on a socket and the local network interface used to
53840 reach the destination is down.
- 53841 [ENETUNREACH]
53842 A write was attempted on a socket and no route to the network is present.
- 53843 The *pwrite()* function shall fail and the file pointer remain unchanged if:
- 53844 XSI [EINVAL] The *offset* argument is invalid. The value is negative.
- 53845 XSI [ESPIPE] *fildev* is associated with a pipe or FIFO.

EXAMPLES

53847

Writing from a Buffer

53848

The following example writes data from the buffer pointed to by *buf* to the file associated with the file descriptor *fd*.

53849

53850

```
#include <sys/types.h>
```

53851

```
#include <string.h>
```

53852

```
...
```

53853

```
char buf[20];
```

53854

```
size_t nbytes;
```

53855

```
ssize_t bytes_written;
```

53856

```
int fd;
```

53857

```
...
```

53858

```
strcpy(buf, "This is a test\n");
```

53859

```
nbytes = strlen(buf);
```

53860

```
bytes_written = write(fd, buf, nbytes);
```

53861

```
...
```

53862

APPLICATION USAGE

53863

None.

53864

RATIONALE

53865

See also the RATIONALE section in [read\(\)](#).

53866

An attempt to write to a pipe or FIFO has several major characteristics:

53867

- *Atomic/non-atomic*: A write is atomic if the whole amount written in one operation is not interleaved with data from any other process. This is useful when there are multiple writers sending data to a single reader. Applications need to know how large a write request can be expected to be performed atomically. This maximum is called {PIPE_BUF}. This volume of IEEE Std 1003.1-200x does not say whether write requests for more than {PIPE_BUF} bytes are atomic, but requires that writes of {PIPE_BUF} or fewer bytes shall be atomic.

53874

- *Blocking/immediate*: Blocking is only possible with O_NONBLOCK clear. If there is enough space for all the data requested to be written immediately, the implementation should do so. Otherwise, the calling thread may block; that is, pause until enough space is available for writing. The effective size of a pipe or FIFO (the maximum amount that can be written in one operation without blocking) may vary dynamically, depending on the implementation, so it is not possible to specify a fixed value for it.

53875

53876

53877

53878

53879

53880

- *Complete/partial/deferred*: A write request:

53881

```
int fildes;
```

53882

```
size_t nbyte;
```

53883

```
ssize_t ret;
```

53884

```
char *buf;
```

53885

```
ret = write(fildes, buf, nbyte);
```

53886

may return:

53887

Complete *ret*=*nbyte*

53888

Partial *ret*<*nbyte*

53889

This shall never happen if *nbyte*≤{PIPE_BUF}. If it does happen (with *nbyte*>{PIPE_BUF}), this volume of IEEE Std 1003.1-200x does not guarantee atomicity, even if *ret*≤{PIPE_BUF}, because atomicity is guaranteed according to the amount *requested*, not the amount *written*.

53890

53891

53892

Deferred: *ret* = -1, *errno* = [EAGAIN]

This error indicates that a later request may succeed. It does not indicate that it *shall* succeed, even if *nbyte* ≤ {PIPE_BUF}, because if no process reads from the pipe or FIFO, the write never succeeds. An application could usefully count the number of times [EAGAIN] is caused by a particular value of *nbyte* > {PIPE_BUF} and perhaps do later writes with a smaller value, on the assumption that the effective size of the pipe may have decreased.

Partial and deferred writes are only possible with O_NONBLOCK set.

The relations of these properties are shown in the following tables:

Write to a Pipe or FIFO with O_NONBLOCK clear			
Immediately Writable:	None	Some	<i>nbyte</i>
<i>nbyte</i> ≤ {PIPE_BUF}	Atomic blocking <i>nbyte</i>	Atomic blocking <i>nbyte</i>	Atomic immediate <i>nbyte</i>
<i>nbyte</i> > {PIPE_BUF}	Blocking <i>nbyte</i>	Blocking <i>nbyte</i>	Blocking <i>nbyte</i>

If the O_NONBLOCK flag is clear, a write request shall block if the amount writable immediately is less than that requested. If the flag is set (by *fcntl()*), a write request shall never block.

Write to a Pipe or FIFO with O_NONBLOCK set			
Immediately Writable:	None	Some	<i>nbyte</i>
<i>nbyte</i> ≤ {PIPE_BUF}	-1, [EAGAIN]	-1, [EAGAIN]	Atomic <i>nbyte</i>
<i>nbyte</i> > {PIPE_BUF}	-1, [EAGAIN]	< <i>nbyte</i> or -1, [EAGAIN]	≤ <i>nbyte</i> or -1, [EAGAIN]

There is no exception regarding partial writes when O_NONBLOCK is set. With the exception of writing to an empty pipe, this volume of IEEE Std 1003.1-200x does not specify exactly when a partial write is performed since that would require specifying internal details of the implementation. Every application should be prepared to handle partial writes when O_NONBLOCK is set and the requested amount is greater than {PIPE_BUF}, just as every application should be prepared to handle partial writes on other kinds of file descriptors.

The intent of forcing writing at least one byte if any can be written is to assure that each write makes progress if there is any room in the pipe. If the pipe is empty, {PIPE_BUF} bytes must be written; if not, at least some progress must have been made.

Where this volume of IEEE Std 1003.1-200x requires -1 to be returned and *errno* set to [EAGAIN], most historical implementations return zero (with the O_NDELAY flag set, which is the historical predecessor of O_NONBLOCK, but is not itself in this volume of IEEE Std 1003.1-200x). The error indications in this volume of IEEE Std 1003.1-200x were chosen so that an application can distinguish these cases from end-of-file. While *write()* cannot receive an indication of end-of-file, *read()* can, and the two functions have similar return values. Also, some existing systems (for example, Eighth Edition) permit a write of zero bytes to mean that the reader should get an end-of-file indication; for those systems, a return value of zero from *write()* indicates a successful write of an end-of-file indication.

Implementations are allowed, but not required, to perform error checking for *write()* requests of zero bytes.

The concept of a {PIPE_MAX} limit (indicating the maximum number of bytes that can be written to a pipe in a single operation) was considered, but rejected, because this concept would unnecessarily limit application writing.

See also the discussion of O_NONBLOCK in *read()*.

53939 Writes can be serialized with respect to other reads and writes. If a *read()* of file data can be
 53940 proven (by any means) to occur after a *write()* of the data, it must reflect that *write()*, even if the
 53941 calls are made by different processes. A similar requirement applies to multiple write operations
 53942 to the same file position. This is needed to guarantee the propagation of data from *write()* calls
 53943 to subsequent *read()* calls. This requirement is particularly significant for networked file
 53944 systems, where some caching schemes violate these semantics.

53945 Note that this is specified in terms of *read()* and *write()*. The XSI extensions *readv()* and *writew()*
 53946 also obey these semantics. A new “high-performance” write analog that did not follow these
 53947 serialization requirements would also be permitted by this wording. This volume of
 53948 IEEE Std 1003.1-200x is also silent about any effects of application-level caching (such as that
 53949 done by *stdio*).

53950 This volume of IEEE Std 1003.1-200x does not specify the value of the file offset after an error is
 53951 returned; there are too many cases. For programming errors, such as [EBADF], the concept is
 53952 meaningless since no file is involved. For errors that are detected immediately, such as
 53953 [EAGAIN], clearly the pointer should not change. After an interrupt or hardware error, however,
 53954 an updated value would be very useful and is the behavior of many implementations.

53955 This volume of IEEE Std 1003.1-200x does not specify behavior of concurrent writes to a file from
 53956 multiple processes. Applications should use some form of concurrency control.

53957 FUTURE DIRECTIONS

53958 None.

53959 SEE ALSO

53960 *chmod()*, *creat()*, *dup()*, *fcntl()*, *getrlimit()*, *lseek()*, *open()*, *pipe()*, *ulimit()*, *writew()*, the Base
 53961 Definitions volume of IEEE Std 1003.1-200x, <limits.h>, <stropts.h>, <sys/uio.h>, <unistd.h>

53962 CHANGE HISTORY

53963 First released in Issue 1. Derived from Issue 1 of the SVID.

53964 Issue 5

53965 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
 53966 Threads Extension.

53967 Large File Summit extensions are added.

53968 The *pwrite()* function is added.

53969 Issue 6

53970 The DESCRIPTION states that the *write()* function does not block the thread. Previously this
 53971 said “process” rather than “thread”.

53972 The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are
 53973 marked as part of the XSI STREAMS Option Group.

53974 The following new requirements on POSIX implementations derive from alignment with the
 53975 Single UNIX Specification:

- 53976 • The DESCRIPTION now states that if *write()* is interrupted by a signal after it has
 53977 successfully written some data, it returns the number of bytes written. In the POSIX.1-1988
 53978 standard, it was optional whether *write()* returned the number of bytes written, or whether
 53979 it returned -1 with *errno* set to [EINTR]. This is a FIPS requirement.
- 53980 • The following changes are made to support large files:
 - 53981 — For regular files, no data transfer occurs past the offset maximum established in the
 53982 open file description associated with the *fdes*.

— A second [EFBIG] error condition is added.

- The [EIO] error condition is added.
- The [EPIPE] error condition is added for when a pipe has only one end open.
- The [ENXIO] optional error condition is added.

Text referring to sockets is added to the DESCRIPTION.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The effect of reading zero bytes is clarified.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that *write()* results are unspecified for typed memory objects.

The following error conditions are added for operations on sockets: [EAGAIN], [EWOULDBLOCK], [ECONNRESET], [ENOTCONN], and [EPIPE].

The [EIO] error is made optional.

The [ENOBUFS] error is added for sockets.

The following error conditions are added for operations on sockets: [EACCES], [ENETDOWN], and [ENETUNREACH].

The *writenv()* function is split out into a separate reference page.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/146 is applied, updating text in the ERRORS section from “a SIGPIPE signal is generated to the calling process” to “a SIGPIPE signal shall also be sent to the thread”.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/147 is applied, making a correction to the RATIONALE.

Issue 7

The *pwrite()* function is moved from the XSI option to the Base.

Functionality relating to the XSI STREAMS option is marked obsolescent.

SD5-XSH-ERN-160 is applied, updating the DESCRIPTION to clarify the requirements for the *pwrite()* function, and to change the use of the phrase “file pointer” to “file offset”.

54009 **NAME**

54010 writev — write a vector

54011 **SYNOPSIS**

```
54012 XSI #include <sys/uio.h>
54013 ssize_t writev(int fildes, const struct iovec *iov, int iovcnt);
```

54014 **DESCRIPTION**

54015 The *writev()* function shall be equivalent to *write()*, except as described below. The *writev()*
 54016 function shall gather output data from the *iovcnt* buffers specified by the members of the *iov*
 54017 array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt*-1]. The *iovcnt* argument is valid if greater than 0 and less than
 54018 or equal to {IOV_MAX}, as defined in <limits.h>.

54019 Each *iovec* entry specifies the base address and length of an area in memory from which data
 54020 should be written. The *writev()* function shall always write a complete area before proceeding to
 54021 the next.

54022 If *fildes* refers to a regular file and all of the *iov_len* members in the array pointed to by *iov* are 0,
 54023 *writev()* shall return 0 and have no other effect. For other file types, the behavior is unspecified.

54024 If the sum of the *iov_len* values is greater than {SSIZE_MAX}, the operation shall fail and no data
 54025 shall be transferred.

54026 **RETURN VALUE**

54027 Upon successful completion, *writev()* shall return the number of bytes actually written.
 54028 Otherwise, it shall return a value of -1, the file-pointer shall remain unchanged, and *errno* shall
 54029 be set to indicate an error.

54030 **ERRORS**54031 Refer to *write()*.54032 In addition, the *writev()* function shall fail if:54033 [EINVAL] The sum of the *iov_len* values in the *iov* array would overflow an *ssize_t*.54034 The *writev()* function may fail and set *errno* to:54035 [EINVAL] The *iovcnt* argument was less than or equal to 0, or greater than {IOV_MAX}.54036 **EXAMPLES**54037 **Writing Data from an Array**

54038 The following example writes data from the buffers specified by members of the *iov* array to the
 54039 file associated with the file descriptor *fd*.

```
54040 #include <sys/types.h>
54041 #include <sys/uio.h>
54042 #include <unistd.h>
54043 ...
54044 ssize_t bytes_written;
54045 int fd;
54046 char *buf0 = "short string\n";
54047 char *buf1 = "This is a longer string\n";
54048 char *buf2 = "This is the longest string in this example\n";
54049 int iocnt;
54050 struct iovec iov[3];
```

```
54051     iov[0].iov_base = buf0;
54052     iov[0].iov_len = strlen(buf0);
54053     iov[1].iov_base = buf1;
54054     iov[1].iov_len = strlen(buf1);
54055     iov[2].iov_base = buf2;
54056     iov[2].iov_len = strlen(buf2);
54057     ...
54058     iovcnt = sizeof(iov) / sizeof(struct iovec);
54059     bytes_written = writev(fd, iov, iovcnt);
54060     ...
```

APPLICATION USAGE

None.

RATIONALE

Refer to *write()*.

FUTURE DIRECTIONS

None.

SEE ALSO

readv(), *write()*, the Base Definitions volume of IEEE Std 1003.1-200x, `<limits.h>`, `<sys/uio.h>`

CHANGE HISTORY

First released in Issue 4, Version 2.

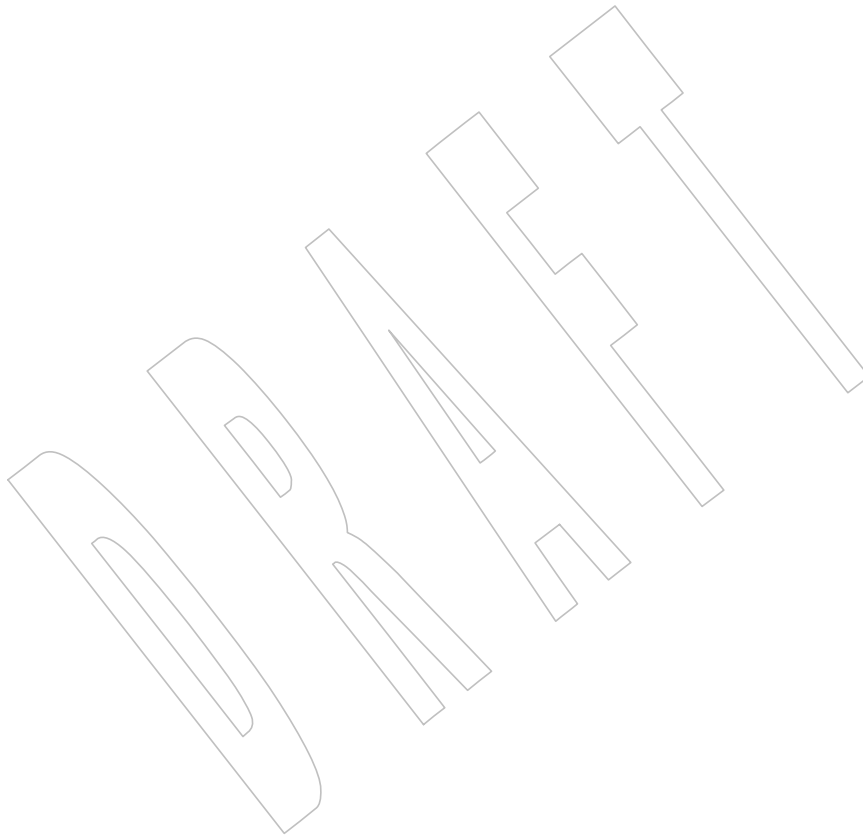
Issue 6

Split out from the *write()* reference page.

54073 **NAME**
54074 `wscanf` — convert formatted wide-character input

54075 **SYNOPSIS**
54076 `#include <stdio.h>`
54077 `#include <wchar.h>`
54078 `int wscanf(const wchar_t *restrict format, ...);`

54079 **DESCRIPTION**
54080 Refer to *fwscanf()*.



54081 **NAME**
 54082 `y0, y1, yn` — Bessel functions of the second kind

54083 **SYNOPSIS**

```
54084 XSI #include <math.h>
54085 double y0(double x);
54086 double y1(double x);
54087 double yn(int n, double x);
```

54088 **DESCRIPTION**

54089 The `y0()`, `y1()`, and `yn()` functions shall compute Bessel functions of x of the second kind of
 54090 orders 0, 1, and n , respectively.

54091 An application wishing to check for error situations should set `errno` to zero and call
 54092 `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `errno` is non-zero or
 54093 `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-
 54094 zero, an error has occurred.

54095 **RETURN VALUE**

54096 Upon successful completion, these functions shall return the relevant Bessel value of x of the
 54097 second kind.

54098 If x is NaN, NaN shall be returned.

54099 If the x argument to these functions is negative, `-HUGE_VAL` or NaN shall be returned, and a
 54100 domain error may occur.

54101 If x is 0.0, `-HUGE_VAL` shall be returned and a pole error may occur.

54102 If the correct result would cause underflow, 0.0 shall be returned and a range error may occur.

54103 If the correct result would cause overflow, `-HUGE_VAL` or 0.0 shall be returned and a range
 54104 error may occur.

54105 **ERRORS**

54106 These functions may fail if:

54107 **Domain Error** The value of x is negative.

54108 If the integer expression `(math_errhandling & MATH_ERRNO)` is non-zero,
 54109 then `errno` shall be set to [EDOM]. If the integer expression `(math_errhandling`
 54110 `& MATH_ERREXCEPT)` is non-zero, then the invalid floating-point exception
 54111 shall be raised.

54112 **Pole Error** The value of x is zero.

54113 If the integer expression `(math_errhandling & MATH_ERRNO)` is non-zero,
 54114 then `errno` shall be set to [ERANGE]. If the integer expression
 54115 `(math_errhandling & MATH_ERREXCEPT)` is non-zero, then the divide-by-zero
 54116 floating-point exception shall be raised.

54117 **Range Error** The correct result would cause overflow.

54118 If the integer expression `(math_errhandling & MATH_ERRNO)` is non-zero,
 54119 then `errno` shall be set to [ERANGE]. If the integer expression
 54120 `(math_errhandling & MATH_ERREXCEPT)` is non-zero, then the overflow
 54121 floating-point exception shall be raised.

54122 Range Error The value of x is too large in magnitude, or the correct result would cause
 54123 underflow.

54124 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 54125 then *errno* shall be set to [ERANGE]. If the integer expression
 54126 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 54127 floating-point exception shall be raised.

EXAMPLES

54128 None.
 54129

APPLICATION USAGE

54130 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
 54132 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

54133 None.
 54134

FUTURE DIRECTIONS

54135 None.
 54136

SEE ALSO

54137 *feclearexcept()*, *fetestexcept()*, *isnan()*, *j0()*, the Base Definitions volume of IEEE Std 1003.1-200x,
 54138 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>
 54139

CHANGE HISTORY

54140 First released in Issue 1. Derived from Issue 1 of the SVID.
 54141

Issue 5

54142 The DESCRIPTION is updated to indicate how an application should check for an error. This
 54143 text was previously published in the APPLICATION USAGE section.
 54144

Issue 6

54145 The normative text is updated to avoid use of the term “must” for application requirements.
 54146
 54147 The RETURN VALUE and ERRORS sections are reworked for alignment of the error handling
 54148 with the ISO/IEC 9899:1999 standard.
 54149
 54150 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/148 is applied, updating the RETURN
 54151 VALUE and ERRORS sections. The changes are made for consistency with the general rules
 54152 stated in “Treatment of Error Conditions for Mathematical Functions” in the Base Definitions
 volume of IEEE Std 1003.1-200x.

Index

<code>_exit</code>	87, 1680	<code>_POSIX2_PBS_TRACK</code>	1558
<code>_Exit()</code>	87	<code>_POSIX2_SW_DEV</code>	1558
<code>_FILE_</code>	138	<code>_POSIX2_UPE</code>	1558
<code>_IOFBF</code>	1359, 1400	<code>_POSIX2_VERSION</code>	1558
<code>_IOLBF</code>	392, 1400	<code>_POSIX_</code>	15, 17
<code>_IONBF</code>	1359, 1400	<code>_POSIX_ADVISORY_INFO</code>	420, 1556
<code>_LINE_</code>	138	<code>_POSIX_ASYNCHRONOUS_IO</code>	1556
<code>_longjmp()</code>	92	<code>_POSIX_ASYNC_IO</code>	418
<code>_LVL</code>	17	<code>_POSIX_BARRIERS</code>	1556
<code>_MAX</code>	16	<code>_POSIX_CHOWN_RESTRICTED</code>	197, 418, 420
<code>_MIN</code>	16	<code>_POSIX_CLOCK_SELECTION</code>	1556
PC constants		<code>_POSIX_CPUTIME</code>	1556
used in pathconf	418	<code>_POSIX_C_SOURCE</code>	14
<code>_PC_2_SYMLINKS</code>	418	<code>_POSIX_FSYNC</code>	1556
<code>_PC_ALLOC_SIZE_MIN</code>	418	<code>_POSIX_IPV6</code>	1556
<code>_PC_ASYNC_IO</code>	418	<code>_POSIX_JOB_CONTROL</code>	1556
<code>_PC_CHOWN_RESTRICTED</code>	418	<code>_POSIX_MAPPED_FILES</code>	1557
<code>_PC_FILESIZEBITS</code>	418	<code>_POSIX_MEMLOCK</code>	1557
<code>_PC_LINK_MAX</code>	418	<code>_POSIX_MEMLOCK_RANGE</code>	1557
<code>_PC_MAX_CANON</code>	418	<code>_POSIX_MEMORY_PROTECTION</code>	1557
<code>_PC_MAX_INPUT</code>	418	<code>_POSIX_MESSAGE_PASSING</code>	1557
<code>_PC_NAME_MAX</code>	418	<code>_POSIX_MONOTONIC_CLOCK</code>	1557
<code>_PC_NO_TRUNC</code>	418	<code>_POSIX_NAME_MAX</code>	844, 854, 1329, 1337, 1406
<code>_PC_PATH_MAX</code>	418	<code>_POSIX_NO_TRUNC</code>	418
<code>_PC_PIPE_BUF</code>	418	<code>_POSIX_OPEN_MAX</code>	583
<code>_PC_PRIO_IO</code>	418	<code>_POSIX_PATH_MAX</code>	844, 854, 1329, 1337, 1406
<code>_PC_REC_INCR_XFER_SIZE</code>	418	<code>_POSIX_PRIORITIZED_IO</code>	41-42, 1557
<code>_PC_REC_MAX_XFER_SIZE</code>	418	<code>_POSIX_PRIORITY_SCHEDULING</code>	41, 1557
<code>_PC_REC_MIN_XFER_SIZE</code>	418	<code>_POSIX_PRIO_IO</code>	418
<code>_PC_REC_XFER_ALIGN</code>	418	<code>_POSIX_RAW_SOCKETS</code>	1557
<code>_PC_SYMLINK_MAX</code>	418	<code>_POSIX_READER_WRITER_LOCKS</code>	1557
<code>_PC_SYNC_IO</code>	418	<code>_POSIX_REALTIME_SIGNALS</code>	1557
<code>_PC_VDISABLE</code>	418	<code>_POSIX_REGEXP</code>	1557
POSIX2 constants		<code>_POSIX_SAVED_IDS</code>	1557
in sysconf	1556	<code>_POSIX_SEMAPHORES</code>	1557
<code>_POSIX2_CHAR_TERM</code>	1558	<code>_POSIX_SHARED_MEMORY_OBJECTS</code>	1557
<code>_POSIX2_C_BIND</code>	1558	<code>_POSIX_SHELL</code>	1557
<code>_POSIX2_C_DEV</code>	1558	<code>_POSIX_SOURCE</code>	14
<code>_POSIX2_FORT_DEV</code>	1558	<code>_POSIX_SPAWN</code>	1557
<code>_POSIX2_FORT_RUN</code>	1558	<code>_POSIX_SPIN_LOCKS</code>	1557
<code>_POSIX2_LOCALEDEF</code>	1558	<code>_POSIX_SPORADIC_SERVER</code>	1557
<code>_POSIX2_PBS</code>	1558	<code>_POSIX_SS_REPL_MAX</code>	1557
<code>_POSIX2_PBS_ACCOUNTING</code>	1558	<code>_POSIX_SYNCHRONIZED_IO</code>	1557
<code>_POSIX2_PBS_CHECKPOINT</code>	1558	<code>_POSIX_SYNC_IO</code>	418
<code>_POSIX2_PBS_LOCATE</code>	1558	<code>_POSIX_THREADS</code>	249, 1557
<code>_POSIX2_PBS_MESSAGE</code>	1558	<code>_POSIX_THREAD_ATTR_STACKADDR</code>	1557
		<code>_POSIX_THREAD_ATTR_STACKSIZE</code>	1557

_POSIX_THREAD_CPUTIME.....	1557	_SC_ARG_MAX	1556
_POSIX_THREAD_PRIORITY_SCHEDULING.....	1557	_SC_ASYNCHRONOUS_IO.....	1556
_POSIX_THREAD_PRIO_INHERIT.....	1557	_SC_ATEXIT_MAX	1556
_POSIX_THREAD_PRIO_PROTECT	1557	_SC_BARRIERS	1556
_POSIX_THREAD_PROCESS_SHARED.....	1557	_SC_BC_BASE_MAX.....	1556
_POSIX_THREAD_ROBUST_PRIO_INHERIT.....	1557	_SC_BC_DIM_MAX.....	1556
_POSIX_THREAD_ROBUST_PRIO_PROTECT.....	1557	_SC_BC_SCALE_MAX	1556
_POSIX_THREAD_SAFE_FUNCTIONS.....	249, 1557	_SC_BC_STRING_MAX	1556
_POSIX_THREAD_SPORADIC_SERVER	1557	_SC_CHILD_MAX	1556
_POSIX_TIMEOUTS	1557	_SC_CLK_TCK	1556, 1612
_POSIX_TIMERS	1557	_SC_CLOCK_SELECTION	1556
_POSIX_TRACE.....	1557	_SC_COLL_WEIGHTS_MAX.....	1556
_POSIX_TRACE_EVENT_FILTER.....	1557	_SC_CPUTIME	1556
_POSIX_TRACE_EVENT_NAME_999, X1001, 1557		_SC_DELAYTIMER_MAX	1556
_POSIX_TRACE_INHERIT.....	1557	_SC_EXPR_NEST_MAX.....	1556
_POSIX_TRACE_LOG	1557	_SC_FSYNC.....	1556
_POSIX_TRACE_NAME_MAX	1557	_SC_GETGR_R_SIZE_MAX	534, 537, 1556
_POSIX_TRACE_SYS_MAX	996, 1557	_SC_GETPW_R_SIZE_MAX.....	575, 578, 1556
_POSIX_TRACE_USER_EVENT_MAX.....	1001, 1557	_SC_IOV_MAX	1556
_POSIX_TYPED_MEMORY_OBJECTS	1557	_SC_IPV6	1556
_POSIX_V6_ILP32_OFF32	1558	_SC_JOB_CONTROL.....	1556
_POSIX_V6_ILP32_OFFBIG.....	1558	_SC_LINE_MAX	1556
_POSIX_V6_LP64_OFF64.....	1558	_SC_LOGIN_NAME_MAX	1556
_POSIX_V6_LPBIG_OFFBIG	1558	_SC_MEMLOCK.....	1557
_POSIX_V7_ILP32_OFF32	1557	_SC_MEMLOCK_RANGE.....	1557
_POSIX_V7_ILP32_OFFBIG.....	1557	_SC_MEMORY_PROTECTION	1557
_POSIX_V7_LP64_OFF64.....	1557	_SC_MESSAGE_PASSING.....	1557
_POSIX_V7_LPBIG_OFFBIG	1557	_SC_MONOTONIC_CLOCK	1557
_POSIX_VDISABLE	418	_SC_MQ_OPEN_MAX	1556
_POSIX_VERSION	1557, 1642	_SC_MQ_PRIO_MAX.....	1556
PROCESS.....	17	_SC_NGROUPS_MAX	1556
PTHREAD_THREADS_MAX.....	1128	_SC_OPEN_MAX.....	1556
_SC constants		_SC_PAGESIZE.....	928, 1558
in sysconf.....	1556	_SC_PAGE_SIZE.....	1558
_SC_2_CHAR_TERM.....	1558	_SC_PRIORITIZED_IO.....	1557
_SC_2_C_BIND.....	1558	_SC_PRIORITY_SCHEDULING	1557
_SC_2_C_DEV.....	1558	_SC_RAW_SOCKETS	1557
_SC_2_FORT_DEV	1558	_SC_READER_WRITER_LOCKS	1557
_SC_2_FORT_RUN	1558	_SC_REALTIME_SIGNALS.....	1557
_SC_2_LOCALEDEF.....	1558	_SC_REGEXP	1557
_SC_2_PBS_ACCOUNTING	1558	_SC_RE_DUP_MAX.....	1558
_SC_2_PBS_CHECKPOINT.....	1558	_SC_RTSIG_MAX.....	1558
_SC_2_PBS_LOCATE.....	1558	_SC_SAVED_IDS	1557
_SC_2_PBS_MESSAGE.....	1558	_SC_SEMAPHORES	1557
_SC_2_PBS_TRACK.....	1558	_SC_SEM_NSEMS_MAX	1558
_SC_2_SW_DEV	1558	_SC_SEM_VALUE_MAX	1558
_SC_2_UPE.....	1558	_SC_SHARED_MEMORY_OBJECTS	1557
_SC_2_VERSION.....	922, 1558	_SC_SHELL	1557
_SC_ADVISORY_INFO	1556	_SC_SIGQUEUE_MAX.....	1558
_SC_AIO_LISTIO_MAX.....	1556	_SC_SPAWN.....	1557
_SC_AIO_MAX.....	1556	_SC_SPIN_LOCKS	1557
_SC_AIO_PRIO_DELTA_MAX	1556	_SC_SPORADIC_SERVER	1557
		_SC_SS_REPL_MAX	1557

Index

_SC_STREAM_MAX.....	1558	_TIME.....	17
_SC_SYMLOOP_MAX.....	1558	_tolower().....	94
_SC_SYNCHRONIZED_IO.....	1557	_toupper().....	95
_SC_THREADS.....	1557	_XOPEN_CRYPT.....	1558
_SC_THREAD_ATTR_STACKADDR.....	1557	_XOPEN_ENH_I18N.....	1558
_SC_THREAD_ATTR_STACKSIZE.....	1557	_XOPEN_NAME_MAX.844, 854, 1329, 1337, 1406	
_SC_THREAD_CPUTIME.....	1557	_XOPEN_PATH_MAX..844, 854, 1329, 1337, 1406	
_SC_THREAD_DESTRUCTOR_ITERATIONS.....	1558	_XOPEN_REALTIME.....	349, 1558
_SC_THREAD_KEYS_MAX.....	1558	_XOPEN_REALTIME_THREADS.....	1558
_SC_THREAD_PRIORITY_SCHEDULING.....	1557	_XOPEN_SHM.....	1558
_SC_THREAD_PRIO_INHERIT.....	1557	_XOPEN_SOURCE.....	14-15
_SC_THREAD_PRIO_PROTECT.....	1557	_XOPEN_STREAMS.....	1558
_SC_THREAD_PROCESS_SHARED.....	1557	_XOPEN_UNIX.....	1558
_SC_THREAD_ROBUST_PRIO_INHERIT.....	1557	_XOPEN_VERSION.....	1558
_SC_THREAD_ROBUST_PRIO_PROTECT.....	1557	a64l().....	96
_SC_THREAD_SAFE_FUNCTIONS.....	1557	ABDAY_1.....	890
_SC_THREAD_SPORADIC_SERVER.....	1557	abort().....	98
_SC_THREAD_STACK_MIN.....	1558	abs().....	100
_SC_THREAD_THREADS_MAX.....	1558	accept().....	101
_SC_TIMEOUTS.....	1557	access().....	103
_SC_TIMERS.....	1557	acos().....	106
_SC_TIMER_MAX.....	1558	acosf.....	106
_SC_TRACE.....	1557	acosh().....	108
_SC_TRACE_EVENT_FILTER.....	1557	acoshf.....	108
_SC_TRACE_EVENT_NAME_MAX.....	1557	acoshl.....	108
_SC_TRACE_INHERIT.....	1557	acosl.....	106
_SC_TRACE_LOG.....	1557	acosl().....	110
_SC_TRACE_NAME_MAX.....	1557	ACTION.....	610
_SC_TRACE_SYS_MAX.....	1557	address information.....	446
_SC_TRACE_USER_EVENT_MAX.....	1557	address string.....	446
_SC_TTY_NAME_MAX.....	1558	addrinfo structure.....	446
_SC_TZNAME_MAX.....	1558	ADV.....	3
_SC_V6_ILP32_OFF32.....	1558	ADVANCED REALTIME THREADS.....	1122
_SC_V6_ILP32_OFFBIG.....	1558	AF.....	17
_SC_V6_LP64_OFF64.....	1558	AIO.....	16
_SC_V6_LPBIG_OFFBIG.....	1558	aio.....	16
_SC_V7_ILP32_OFF32.....	1557	AIO_ALLDONE.....	111
_SC_V7_ILP32_OFFBIG.....	1557	aio_cancel().....	111
_SC_V7_LP64_OFF64.....	1557	AIO_CANCELED.....	111
_SC_V7_LPBIG_OFFBIG.....	1557	aio_error().....	113
_SC_VERSION.....	1557	aio_fsync().....	114
_SC_XOPEN_CRYPT.....	1558	AIO_LISTIO_MAX.....	737, 1556
_SC_XOPEN_ENH_I18N.....	1558	AIO_MAX.....	737, 1556
_SC_XOPEN_REALTIME.....	1558	AIO_NOTCANCELED.....	111
_SC_XOPEN_REALTIME_THREADS.....	1558	AIO_PRIO_DELTA_MAX.....	41, 1556
_SC_XOPEN_SHM.....	1558	aio_read().....	116
_SC_XOPEN_STREAMS.....	1558	aio_return().....	119
_SC_XOPEN_UNIX.....	1558	aio_suspend().....	121
_SC_XOPEN_VERSION.....	1558	aio_write().....	123
_setjmp.....	92	ai.....	16
_t.....	17	AI_ADDRCONFIG.....	447
		AI_ALL.....	447

Index

casinl.....	168	CLOCKS_PER_SEC.....	203
casinl().....	170	CLOCK_.....	17
catan.....	171	clock.....	17
catanf.....	171	clock_getcpuclockid().....	204
catanh().....	172	clock_getres().....	205
catanhf.....	172	clock_gettime.....	205
catanhl.....	172	CLOCK_MONOTONIC.....	49, 209
catanl.....	171	clock_nanosleep().....	208
catanl().....	173	CLOCK_PROCESS_CPUTIME_ID.....	49
catclose().....	174	CLOCK_REALTIME.....	49, 205, 209, 875, 1152, 1604
catgets().....	175	clock_settime.....	205
catopen().....	177	clock_settime().....	211
CBAUD.....	19	CLOCK_THREAD_CPUTIME_ID.....	50
cbrt().....	179	clog().....	212
cbrtf.....	179	clogf.....	212
cbrtl.....	179	clogl.....	212
ccos().....	180	close a file.....	215
ccosf.....	180	close().....	213
ccosh().....	181	closedir().....	216
ccoshf.....	181	closelog().....	218
ccoshl.....	181	cmsg.....	16
ccosl.....	180	CMSG.....	17
ccosl().....	182	COLL_WEIGHTS_MAX.....	1556
CD.....	3	command interpreter.....	
ceil().....	183	portable.....	1680
ceilf.....	183	compare thread IDs.....	1117
ceill.....	183	compilation environment.....	14
cexp().....	185	condition variable initialization attributes.....	1103
cexpf.....	185	conforming application.....	1464
cexpl.....	185	conforming application, strictly.....	126, 314
cfgetispeed().....	186	confstr().....	222
cfgetospeed().....	188	conj().....	225
cfsetispeed().....	189	conjf.....	225
cfsetospeed().....	190	conjl.....	225
change current working directory.....	192	connect().....	226
change file modes.....	195	control data.....	38
change owner and group of file.....	199	control-normal.....	1246
char.....	83	conversion descriptor.....	308, 313, 616-617, 619-620
CHAR_MAX.....	748, 750	conversion specifier.....	424, 458, 496, 505, 1501, 1505, 1521
chdir().....	191	modified.....	1507
CHILD_MAX.....	414, 1556	conversion specifier.....	
chmod().....	193	modified.....	1522
chown().....	197	copysign().....	229
cimag().....	201	copysignf.....	229
cimagf.....	201	copysignl.....	229
cimagl.....	201	core.....	1681
CLD_.....	17	core file.....	89
clearerr().....	202	cos().....	230
clock tick.....	126, 1559, 1612	cosf.....	230
clock ticks/second.....	1556	cosh().....	232
clock().....	203	coshf.....	232
clock-resolution attribute.....	77, 974	coshl.....	232
		cosl.....	230

cosl()	234	dbm_close	254
covert channel	716	dbm_delete	254
cpow()	235	dbm_error	254
cpowf	235	dbm_fetch	254
cpowl	235	dbm_firstkey	254
cproj()	236	DBM_INSERT	255
cprojf	236	dbm_nextkey	254
cprojl	236	dbm_open	254
CPT	4	DBM_REPLACE	255
creal()	237	dbm_store	254
crealf	237	DEAD_PROCESS	295-296
creall	237	DEFECHO	19
creat()	238	deferred cancelability	1129
create a per-process timer	1605	defined types	82
create an interprocess channel	916	delay process execution	1463
create session and set process group ID	1390	DELAYTIMER_MAX	1556, 1609
creation-time attribute	77, 974	dependency ordering	269
CRYPT	240, 281, 1371	descriptive name	446
crypt()	240	destroying a mutex	1140
csin()	242	destroying condition variables	1095
csinf	242	destructor functions	1132
csinh()	243	detaching a thread	1115
csinhf	243	difftime()	258
csinhl	243	DIR	82, 216, 1252, 1254, 1295, 1317, 1596
csinl	242	directive	424, 458, 496, 505, 1521
csinl()	244	directory operations	357
csqrt()	245	dirent structure	358
csqrtf	245	dirfd()	259
csqrtl	245	dirname()	261
ctan()	246	div()	263
ctanf	246	dlclose()	264
ctanh()	247	dlopen()	266
ctanhf	247	dlopen()	268
ctanhl	247	dlsym()	271
ctanl	246	dot	357, 1291
ctanl()	248	dot-dot	357, 1291
ctermid()	249	dprintf	424
ctime()	251	dprintf()	273
ctime_r	251	drand48()	274
CX	4	dup()	277
c_	17	dup2	277
data key creation	1133	duplocale()	279
data messages	38	dynamic package initialization	1178
data type	82	d_	16
DATMSK	520	E2BIG	22
daylight	253, 1636	EACCESS	22
DBL_MANT_DIG	183, 393	EADDRINUSE	22
DBL_MAX_EXP	183, 393	EADDRNOTAVAIL	22
DBM	254-255	EAFNOSUPPORT	22
dbm_	16	EAGAIN	22, 28
DBM_	17	EAI_AGAIN	511
dbm_clearerr()	254	EAI_BADFLAGS	511
		EAI_FAIL	511

Index

EAI_FAMILY	511	endservent()	293
EAI_MEMORY	511	endutxent()	295
EAI_NONAME	511	ENETDOWN	25
EAI_OVERFLOW	511	ENETRESET	25
EAI_SERVICE	511	ENETUNREACH	25
EAI_SOCKETYPE	511	ENFILE	25
EAI_SYSTEM	511	ENOBUFFS	25
EALREADY	22	ENODATA	25
EBADF	22	ENODEV	25
EBADMSG	22	ENOENT	25
EBUSY	23	ENOEXEC	25
ECANCELED	23	ENOLCK	25
ECHILD	23	ENOLINK	25
ECHOCTL	19	ENOMEM	25
ECHOKE	19	ENOMSG	26
ECHOPRT	19	ENOPROTOOPT	26
ECONNABORTED	23	ENOSPC	26
ECONNREFUSED	23	ENOSR	26
ECONNRESET	23	ENOSTR	26
EDEADLK	23	ENOSYS	26
EDESTADDRREQ	23	ENOTCONN	26
EDOM	23	ENOTDIR	26
EDQUOT	23	ENOTEMPTY	26
EEXIST	23	ENOTRECOVERABLE	1098, 1148
EFAULT	23	ENOTSOCK	26
EFBIG	24	ENOTSUP	26
effective group ID	199, 315, 540	ENOTTY	26
effective user ID	104, 315, 716	ENTRY	610
EHOSTUNREACH	24	environ	298, 315
EIDRM	24	envp	315
Eighth Edition UNIX	1761	ENXIO	26
EILSEQ	24, 37	EOPNOTSUPP	26
EINPROGRESS	24, 42	EOVERFLOW	26
EINTR	24, 57, 993	EOWNERDEAD	1098, 1148
EINVAL	24, 980, 993, 1137, 1140, 1168	EPERM	27, 1148
EIO	24	EPIPE	27
EISCONN	24	EPROTO	27
EISDIR	24	EPROTONOSUPPORT	27
ELOOP	24	EPROTOTYPE	27
ELSIZE	776	erand48	274
EMFILE	24	erand48()	299
EMLINK	24	ERANGE	27
EMPTY	296	erf()	300
EMSGSIZE	25	erfc()	302
EMULTIHOP	25	erfcf	302
ENAMETOOLONG	25	erfcl	302
encrypt()	281	erff	300
endgrent()	283	erff()	304
endhostent()	285	erfl	300, 304
endnetent()	287	EROFS	27
endprotoent()	289	errno	305
endpwent()	291	error descriptions	511
		error numbers	21

additional.....	28	fchmodat.....	193
ESPIPE.....	27	fchmodat().	335
ESRCH.....	27	fchown().	336
EST5EDT.....	1636	fchownat.....	197
establishing cancellation handlers.....	1088	fchownat().	338
ESTALE.....	27	fclose().	339
ETIME.....	27	fcntl().	341
ETIMEDOUT.....	27	FD.....	4
ETXTBSY.....	27	fdatasync().	349
EWOULDBLOCK.....	28	fdetach().	350
examine and change blocked signals.....	1212	fdim().	352
examine and change signal action.....	1421	fdimf.....	352
EXDEV.....	28	fdiml.....	352
exec.....	307	fdopen().	354
of shell scripts.....	314	fdopendir().	356
exec family104, 215, 346, 390, 416, 1042, 1379, 1680		fds.....	16
execl.....	307	FD.....	16
execle.....	307	fd.....	16
execlp.....	307	FD.....	17
execute a file.....	314	FE, CLOEXEC41, 357, 620, 894, 915, 936, 943, 1402	
execution time monitoring.....	49	FD_CLR.....	1035
execv.....	307	FD_CLR().	86
execve.....	307	FD_ISSET.....	86, 1035
execvp.....	307	FD_SET.....	86, 1035
exit().	319	FD_ZERO.....	86, 1035
EXIT_FAILURE.....	87, 319	feature test macro.....	14, 515
EXIT_SUCCESS.....	87, 90, 319	_POSIX_C_SOURCE.....	14
exp().	320	_XOPEN_SOURCE.....	14
exp2().	322	feclearexcept().	359
exp2f.....	322	fegetenv().	360
exp2l.....	322	fegetexceptflag().	361
expf.....	320	fegetround().	362
expl.....	320	fehldexcept().	364
expm1().	324	feof().	365
expm1f.....	324	feraiseexcept().	366
expm1l.....	324	ferror().	367
EXPR_NEST_MAX.....	1556	fesetenv.....	360
EXTA.....	19	fesetenv().	368
EXTB.....	19	fesetexceptflag.....	361
extension.....		fesetexceptflag().	369
CX.....	4	fesetround.....	362
OH.....	6	fesetround().	370
XSI.....	9	fetestexcept().	371
extensions to setlocale.....	1373	feupdateenv().	373
fabs().	326	fexecve.....	307, 375
fabsf.....	326	FE.....	16
fabsl.....	326	fflush().	376
faccessat.....	103	ffs().	379
faccessat().	328	fgetc().	380
fattach().	329	fgetpos().	382
fchdir().	332	fgets().	384
fchmod().	333	fgetwc().	386
		fgetws().	388

Index

FIFO.....	809-810, 812, 899, 1760	format of entries	10
FILE.....	1644, 1646, 1664, 1666, 1668, 1670, 1672, 1674	fpathconf()	418
file		fpclassify().....	423
locking.....	346	FPE_.....	17
file accessibility	104	fprintf()	424
file control.....	346	fputc()	436
FILE object.....	34	fputs()	438
file permission bits	105	fputwc().....	440
file permissions.....	104, 420, 476	fputws().....	442
file position indicator.....	34	FP_ILOGB0.....	626
fileno()	390	FP_ILOGBNAN	626
FILESIZEBITS	418	FQDN	554
FIND.....	610	FR.....	4
find string token	1536	fread()	443
flockfile()	391	free().....	445
floor().....	393	freeaddrinfo()	446
floorf.....	393	freelocale().....	450
floorl.....	393	freopen().....	452
FLT_RADIX.....	766	frexp()	456
FLT_ROUNDSD.....	395	frexpf.....	456
FLUSH.....	17	frexpl.....	456
FLUSHO	19	FSC.....	4
FLUSHR.....	640	fscanf()	458
FLUSHRW	640	fseek().....	465
FLUSHW.....	640	fseeko.....	465
fma().....	395	fsetpos().....	468
fmaf.....	395	fstat().....	470
fmal.....	395	fstatat().....	473
fmax().....	397	fstatvfs().....	478
fmaxf.....	397	fsync().....	481
fmaxl.....	397	ftell().....	483
fmemopen()	398	ftello.....	483
fmin().....	401	ftok().....	485
fminf.....	401	ftuncate().....	487
fminl.....	401	ftrylockfile	391
FMNAMESZ	639	ftrylockfile()	489
fmod().....	402	FTW.....	17, 884-885
fmodf.....	402	ftw().....	490
fmodl.....	402	FTW_CHDIR.....	884
fmtmsg().....	404	FTW_D.....	490, 884
fnmatch().....	407	FTW_DEPTH	884
FNM_.....	17	FTW_DNR.....	490, 884-885
FNM_NOESCAPE.....	407	FTW_DP.....	884
FNM_NOMATCH.....	407	FTW_F.....	490, 884
FNM_PATHNAME	407	FTW_MOUNT	884
FNM_PERIOD	407	FTW_NS.....	490, 885
fopen()	409	FTW_PHYS.....	884
FOPEN_MAX.....	354, 399, 410, 1614	FTW_SL.....	490, 884
foreground.....	1378	FTW_SLN	884
fork handler.....	1043	fully-qualified domain name.....	554
fork()	413	functions	13
forkall.....	416	implementation.....	13
		use.....	13

funlockfile.....	391	geteuid().....	531
funlockfile().....	493	getgid().....	532
futimesat.....	1660	getgrent.....	283
futimesat().....	494	getgrent().....	533
fwide().....	495	getgrgid().....	534
fwprintf().....	496	getgrgid_r.....	534
fwrite().....	503	getgrnam().....	537
fwscanf().....	505	getgrnam_r.....	537
f.....	16	getgroups().....	539
F.....	17	gethostent.....	285
F_DUPFD.....	277, 341, 343-344	gethostent().....	541
F_GETFD.....	341, 343, 346	gethostid().....	542
F_GETFL.....	341, 343, 346	gethostname().....	543
F_GETLK.....	342-344	getitimer().....	544
F_GETOWN.....	341, 343	getline.....	525
F_LOCK.....	755	getline().....	546
F_RDLCK.....	344	getlogin().....	547
F_SETFD.....	341, 343, 346	getlogin_r.....	547
F_SETFL.....	341, 343, 346	getmsg().....	550
F_SETLK.....	342-344	getnameinfo().....	554
F_SETLKW.....	55, 342-344	GETNCNT.....	1340-1341
F_SETOWN.....	341, 343	getnetbyaddr.....	287
F_TEST.....	755	getnetbyaddr().....	557
F_TLOCK.....	755	getnetbyname.....	287, 557
F_ULOCK.....	755	getnetent.....	287, 557
F_UNLCK.....	342-343	getopt().....	558
F_WRLCK.....	344	getpeername().....	563
gai_strerror().....	511	getpgid().....	565
generation-version attribute.....	77, 974	getpgrp().....	566
get configurable pathname variables.....	420	GETPID.....	1340-1341
get configurable system variables.....	1559	getpid().....	567
get file status.....	476	getpmsg.....	550
get process times.....	1612	getpmsg().....	568
get supplementary group IDs.....	540	getppid().....	569
get system time.....	1603	getpriority().....	570
get thread ID.....	1202	getprotent.....	573
get user name.....	548	getprotobyname.....	289
getaddrinfo.....	446	getprotobyname().....	573
getaddrinfo().....	512	getprotobynumber.....	289, 573
GETALL.....	1340	getprotoent.....	289
getc().....	513	getpwent.....	291
getchar().....	516	getpwent().....	574
getchar_unlocked.....	514	getpwnam().....	575
getchar_unlocked().....	517	getpwnam_r.....	575
getcwd().....	518	getpwuid().....	578
getc_unlocked().....	514	getpwuid_r.....	578
getdate().....	520	getrlimit().....	581
getdate_err.....	520	getrusage().....	584
getdelim().....	525	gets().....	586
getegid().....	527	getservbyname.....	293
getenv.....	315	getservbyname().....	588
getenv().....	528	getservbyport.....	293, 588
		getservent.....	293, 588

Index

getsid()	589	iconv_open()	620
getsockname()	590	ic_	16
getsockopt()	591	IEEE Std 754-1985	3
getsubopt()	594	IEEE Std 854-1987	3
gettimeofday()	598	ifc_	16
getuid()	599	ifra_	16
getutxent	295	ifru_	16
getutxent()	600	if_	16
getutxid	295, 600	IF_	17
getutxline	295, 600	if_freenameindex()	622
GETVAL	1340-1341	if_indextoname()	623
getwc()	601	if_nameindex()	624
getwchar()	602	if_nametoindex()	625
GETZCNT	1340-1341	ILL_	17
glob()	603	ilogb()	626
globfree	603	ilogbf	626
GLOB_	17	ilogbl	626
GLOB_ constants		imaxabs()	628
error returns of glob	605	imaxdiv()	629
used in glob	603	implementation-defined	1
GLOB_ABORTED	605	IMPLINK_	17
GLOB_APPEND	603-604	in6_	16
GLOB_DOOFFS	603-604	IN6_	17
GLOB_ERR	603, 605	INADDR_	17
GLOB_MARK	604	inet	16
GLOB_NOCHECK	604-605	inet_addr()	630
GLOB_NOESCAPE	604	inet_ntoa	630
GLOB_NOMATCH	605	inet_ntop()	632
GLOB_NOSORT	604	inet_pton	632
GLOB_NOSPACE	605	Inf	133
gl_	16	INF	427, 499
GMT0	1636	INFINITY	427, 499
gmtime()	607	INFO	405
gmtime_r	607	infu_	16
grantpt()	609	inheritance attribute	77, 976
HALT	405	init	90, 716
hcreate()	610	initialize a named semaphore	1329
hdestroy	610	initialize an unnamed semaphore	1326
high resolution sleep	875	initializing a mutex	1140
host name	446	initializing condition variables	1095
hsearch	610	initstate()	634
htonl()	613	INIT_PROCESS	295-296
htons	613	input and output rationale	1248
IEEE 754-1985	3	insque()	636
IEEE 854-1987	3	INT	18
IEEE Std 754-1985	3	international environment	1373
IEEE Std 854-1987	3	Internet Protocols	67
ifc_	16	INT_MAX	626
ifra_	16	INT_MIN	100
ifru_	16	in_	16
if_	16	IN_	17
IF_	17	ioctl()	639
if_freenameindex()	622	IOV_	17
if_indextoname()	623		
if_nameindex()	624		
if_nametoindex()	625		
ILL_	17		
ilogb()	626		
ilogbf	626		
ilogbl	626		
imaxabs()	628		
imaxdiv()	629		
implementation-defined	1		
IMPLINK_	17		
in6_	16		
IN6_	17		
INADDR_	17		
inet	16		
inet_addr()	630		
inet_ntoa	630		
inet_ntop()	632		
inet_pton	632		
Inf	133		
INF	427, 499		
INFINITY	427, 499		
INFO	405		
infu_	16		
inheritance attribute	77, 976		
init	90, 716		
initialize a named semaphore	1329		
initialize an unnamed semaphore	1326		
initializing a mutex	1140		
initializing condition variables	1095		
initstate()	634		
INIT_PROCESS	295-296		
input and output rationale	1248		
insque()	636		
INT	18		
international environment	1373		
Internet Protocols	67		
INT_MAX	626		
INT_MIN	100		
in_	16		
IN_	17		
ioctl()	639		
IOV_	17		

iov_	17	isdigit()	660
IOV_MAX	1260, 1556, 1764	isdigit_l	660
IP6	4	isfinite()	662
IPC	39, 858, 860, 863, 865, 1345, 1349, 1412, 1414	isgraph()	663
ipc_	16	isgraph_l	663
IPC_	17	isgreater()	665
IPC_constants		isgreaterequal()	666
used in semctl	1340	isinf()	667
used in shmctl	1410	isless()	668
IPC_CREAT	859, 1343, 1413	islessequal()	669
IPC_EXCL	859, 1343	islessgreater()	670
IPC_NOWAIT	861-862, 864-865, 1346	islower()	671
IPC_PRIVATE	859, 1343, 1413	islower_l	671
IPC_RMID	857, 1341, 1410	isnan()	673
IPC_SET	857, 1341, 1410, 126, 319, 346, 515, 778, 1242, 1291, 1373, 1422, 1447, 1603	isnormal()	674
IPC_STAT	857, 1340, 1410	isprint()	675
IPPORT_	17	isprint_l	675
IPPROTO_	17	ispunct()	677
IPv4	68	ispunct_l	677
IPv4-compatible address	69	isspace()	679
IPv4-mapped address	69	isspace_l	679
IPv6	68	isunordered()	681
compatibility with IPv4	69	isupper()	682
interface identification	70	isupper_l	682
options	70	iswalnum()	684
IPv6 address		iswalnum_l	684
anycast	68	iswalpha()	686
loopback	69	iswalpha_l	686
multicast	68	iswblank()	688
unicast	68	iswblank_l	688
unspecified	69	iswcntrl()	690
IPV6_	17	iswcntrl_l	690
IPV6_JOIN_GROUP	70	iswctype()	692
IPV6_LEAVE_GROUP	70	iswctype_l	692
IPV6_MULTICAST_HOPS	70	iswdigit()	694
IPV6_MULTICAST_IF	70	iswdigit_l	694
IPV6_MULTICAST_LOOP	70	iswgraph()	696
IPV6_UNICAST_HOPS	70	iswgraph_l	696
IPV6_V6ONLY	71	iswlower()	698
ip_	16	iswlower_l	698
IP_	17	iswprint()	700
isalnum()	650	iswprint_l	700
isalnum_l	650	iswpunct()	702
isalpha()	652	iswpunct_l	702
isalpha_l	652	iswspace()	704
isascii()	654	iswspace_l	704
isastream()	655	iswupper()	706
isatty()	656	iswupper_l	706
isblank()	657	iswxdigit()	708
isblank_l	657	iswxdigit_l	708
iscntrl()	658	isxdigit()	710
iscntrl_l	658	isxdigit_l	710

Index

ITIMER_PROF	544	LC_ALL.....	308, 750, 890, 1372, 1374
ITIMER_VIRTUAL.....	544	LC_COLLATE.....	603-604, 1372, 1374, 1491, 1546, 1695, 1737
it_.....	16-17	LC_CTYPE.....	1624, 1626, 1688, 1712, 1729, 1738-1739, 1741, 1743
I_.....	17	LC_MESSAGES.....	177, 1372-1374, 1499
I_ATMARK.....	645-646	LC_MONETARY.....	750, 1372, 1374, 1503
I_CANPUT.....	646	LC_MONETARY_C05.....	750, 1372, 1374, 1503, 1531, 1717
I_CKBAND.....	646	LC_TIME.....	521, 890, 1372, 1374
I_FDINSERT.....	642	ldexp().....	725
I_FIND.....	641	ldexpf.....	725
I_FLUSH.....	640	ldexpl.....	725
I_FLUSHBAND.....	640	ldiv().....	727
I_GETBAND.....	646	legacy.....	2
I_GETCLTIME.....	646	lfind.....	776
I_GETSIG.....	641	lfind().....	728
I_GRDOPT.....	642, 1246	lgamma().....	729
I_GWROPT.....	644	lgammaf.....	729
I_LINK.....	646	lgammal.....	729
I_LIST.....	645	LINK_MAX.....	1556
I_LOOK.....	639	link to a file.....	733
I_NREAD.....	642	link().....	731
I_PEEK.....	641	linkat.....	731
I_PLINK.....	647	linkat().....	735
I_POP.....	639	LINK_MAX.....	24, 418, 732, 1289
I_PUNLINK.....	648	LIO.....	16
I_PUSH.....	639	lio_.....	16
I_RECVFD.....	22, 645	lio_listio().....	736
I_SENDFD.....	644-645	LIO_NOP.....	736
I_SETCLTIME.....	213, 646	LIO_NOWAIT.....	736
I_SETSIG.....	640-641	LIO_READ.....	736
I_SRDOPT.....	642, 1246	LIO_WAIT.....	736
I_STR.....	643	LIO_WRITE.....	736
I_SWROPT.....	644, 1758	list directed I/O.....	738
I_UNLINK.....	647	listen().....	740
j0().....	712	llabs.....	721
j1.....	712	llabs().....	742
jn.....	712	lldiv.....	727
job control.....	90, 566, 716, 1378, 1390, 1559, 1680	lldiv().....	743
rand48.....	274	LLONG_MAX.....	1539, 1725
rand48().....	714	LLONG_MIN.....	1539, 1725
JST-9.....	1636	llrint().....	744
kill().....	715	llrintf.....	744
killpg().....	718	llrintl.....	744
l64a.....	96	llround().....	746
l64a().....	720	llroundf.....	746
labs().....	721	llroundl.....	746
LANG.....	177	load ordering.....	269
last close.....	1406	LOBLK.....	19
LASTMARK.....	646	localeconv().....	748
lchown().....	722	localtime().....	752
lcong48.....	274	localtime_r.....	752
lcong48().....	724	lockf().....	755
		locking.....	346
		advisory.....	346

mandatory	346	lseek()	778
locking and unlocking a mutex	1149	lstat	473
log()	758	lstat()	780
log-full-policy attribute	75, 77, 976, 979, 996	l_	16
log-max-size attribute	77, 977, 979	L_ctermid	249
log10()	760	l_sysid	346
log10f	760	malloc()	781
log10l	760	manipulate signal sets	1428
log1p()	762	mappings	830
log1pf	762	MAP_	16-17
log1pl	762	MAP_FAILED	830
log2()	764	MAP_FIXED	826, 829
log2f	764	MAP_PRIVATE	413, 826, 830, 835, 867
log2l	764	MAP_SHARED	416, 826-827
logb()	766	max-data-size attribute	77, 979-980
logbf	766	MAX_CANON	418
logbl	766	MAX_INPUT	418
logf	758	may	2
logf()	768	mblen()	783
login shell	314	mbrlen()	785
LOGIN_NAME_MAX	547, 1556	mbrtowc()	787
LOGIN_PROCESS	295-296	mbsinit()	789
logl	758, 768	mbsnrto wcs	791
LOG_	17	mbsnrto wcs()	790
LOG_constants in syslog	218	mbsrtowcs()	791
LOG_ALERT	218	mbstowcs()	793
LOG_CONS	219	mbtowc()	795
LOG_CRIT	218	MB_CUR_MAX	783, 785, 787, 795, 1688, 1739
LOG_DEBUG	218	MC1	4
LOG_EMERG	218	MCL	16
LOG_ERR	218	MCL_CURRENT	823
LOG_INFO	218	MCL_FUTURE	823
LOG_LOCAL	219	memccpy()	797
LOG_NDELAY	219	memchr()	798
LOG_NOTICE	218	memcmp()	799
LOG_NOWAIT	219	memcpy()	800
LOG_ODELAY	219	MEMLOCK_FUTURE	830
LOG_PID	219	memmove()	801
LOG_USER	218-219	memory management	42
LOG_WARNING	218	memset()	802
longjmp()	769	message catalog descriptor	87, 308, 313
LONG_MAX	1539, 1725	message parts	39
LONG_MIN	1539, 1725	message priority	38
rand48	274	high-priority	38
rand48()	771	normal	38
rint()	772	priority	38
rintf	772	message-discard mode	1246
rintl	772	message-nondiscard mode	1246
round()	774	MET-1MEST	1636
roundf	774	MINSIGSTKSZ	1425
roundl	774	mkdir()	803
rsearch()	776	mkdirat	803
		mkdirat()	806

Index

mkdtemp()	807	mq_send()	848
mkfifo()	809	mq_setattr()	850
mkfifoat()	809	mq_timedreceive	845
mkfifoat()	812	mq_timedreceive()	852
mknod()	813	mq_timedsend	848
mknodat	813	mq_timedsend()	853
mknodat()	817	mq_unlink()	854
mkstemp	807	mrnd48	274
mkstemp()	818	mrnd48()	856
mktime()	819	MSG	5, 17
ML	5	msgctl()	857
mlock()	821	msgget()	859
mlockall()	823	msgrcv()	861
MLR	5	msgsnd()	864
mmap()	825	MSGVERB	405-406
MM_	17	msg_	16
MM_APPL	404	MSG_	17
MM_CONSOLE	404	MSG_ANY	550
MM_ERROR	405-406	MSG_BAND	550, 1227
mm_FIRM	404	MSG_EOR	1351, 1353, 1356, 1468, 1470
MM_HALT	405	MSG_HIPRI	550, 1227
MM_HARD	404	MSG_NOERROR	861-862
MM_INFO	405	MSG_NOSIGNAL	1351, 1353, 1356
MM_NOCON	405	MSG_OOB	1351, 1353, 1356
MM_NOMSG	405	msg_perm	40
MM_NOSEV	405	msqid	39
MM_NOTOK	405	MST7MDT	1636
MM_NRECOV	404	msync()	867
MM_NULLMC	404	MS	16-17
MM_OK	405	MS_ASYNC	827, 867
MM_OPSYS	404	MS_INVALIDATE	867-868
MM_PRINT	404, 406	MS_SYNC	827, 867
MM_RECOVER	404	multicast	68
MM_SOFT	404	munlock	821
MM_UTIL	404	munlock()	870
MM_WARNING	405	munlockall	823
modf()	833	munlockall()	871
modff	833	munmap()	872
modfl	833	mutex attributes	1157
MON	5	mutex initialization attributes	1156
MORECTL	551	mutex performance	1157
MOREDATA	551	MUXID_ALL	647-648
mprotect()	835	MUXID_R	17
MQ	16	MX	5
mq	16	M_	17
mq_close()	837	name information	554
mq_close(405)	837	name space	15
mq_notify(405)	840	NAN	133
mq_notify(722, 732, 804, 809, 814, 885, 897, 1038, 1252, 1261, 1264, 1289, 1298, 1551, 1629, 1649, 1658, 1660)	840	NaN	427
mq_open()	842	NaN	427
MQ_OPEN_MAX	1556	NaN	499
MQ_PRIO_MAX	848-849, 1556	NaN	499
mq_receive()	845	NaN	499

nan()	874	OPEN_MAX	291, 344, 356, 490, 842, 943, 946, 1556
nanf	874	open_memstream()	902
nanl	874	open_wmemstream	902
nanosleep()	875	open_wmemstream()	904
NDEBUG	21, 138	optarg	558, 908
nearbyint()	877	opterr	558, 908
nearbyintf	877	optind	558, 908
nearbyintl	877	option	
network interfaces	60	ADV	3
newlocale()	879	BE	3
NEW_TIME	295-296	CD	3
nextafter()	882	CPT	4
nextafterf	882	FD	4
nextafterl	882	FR	4
nexttoward	882	FSC	4
nexttowardf	882	IP6	4
nexttowardl	882	MC1	4
nftw()	884	ML	5
NGROUPS_MAX	540, 1556	MLR	5
nice()	888	MON	5
NLSPATH	177	MSG	5
NL	17	MX	5
NL_ARGMAX	424, 458, 496, 505	PIO	6
NL_CAT_LOCALE	177	PS	6
nl_langinfo()	890	RPI	6
nl_langinfo_l	890	RPP	6
nohup utility	315	RS	6
non-local jumps	1447	SD	6
non-volatile storage	481	SHM	7
rand48	274	SIO	7
rand48()	892	SPN	7
ntohl	613	SS	7
ntohl()	893	TCT	7
ntohs	613, 893	TEF	7
NULL	223, 249, 256, 266, 271, 830, 1254	TPI	7
NUM_EMPL	611	TPP	7
NZERO	570, 888	TPS	8
n_	16	TRC	8
OB	5	TRI	8
OF	5	TRL	8
OH	6	TSA	8
OLD_TIME	295-296	TSH	8
open a file	899	TSP	8
open a named semaphore	1329	TSS	9
open a shared memory object	1404	TYM	9
open()	894	UP	9
openat	894	UU	9
openat()	905	XSR	9
opendir	356	optopt	558, 561, 908
opendir()	906	optstring	561
openlog	218	orphaned process group	90
openlog()	907	O_	17

Index

O_ constants	
used in open()	894
used in posix_openpt()	933
O_ACCMODE	341
O_APPEND	41, 123, 254, 355, 894, 1756
O_CREAT	843, 854, 894, 1320, 1328, 1402-1403, 1405
O_DIRECTORY	895
O_DSYNC	114, 895, 1246, 1757
O_EXCL	843, 895, 1328, 1402-1403
O_EXEC	894
O_NDELAY	1761
O_NOCTTY	895, 933
O_NOFOLLOW	895
O_RDONLY	645, 843, 845, 848, 850, 895, 915, 918, 1228, 1245, 1757, 1760
O_RDWR	261, 842, 894, 1402, 1405
O_RSYNC	332, 755, 842, 894, 933, 1402, 1405
O_SYNC	895, 1246
O_TRUNC	114, 895, 1246, 1757
O_WRONLY	238, 332, 755, 842, 894
PAGESIZE	42-43, 821, 1049, 1558
PAGE_SIZE	1558
PATH	223
PATH environment variable	316
pathconf	418
pathconf()	909
pause()	0, 1256, 1261, 1289, 1298, 1551, 1560, 1629, 1649, 1658, 1660
pclose()	910
pd	911
PENDIN	16
pererror()	19
pererror()	913
persistent connection (I_PLINK)	647
PF_	17
physical write	481
ph	16
PIO	6
pipe	415, 899, 1760
pipe()	915
PIPE_BUF	418, 1757, 1760
PIPE_MAX	1761
plain characters	1501
pointer to a function	32
pointer types	83
POLL	17
poll()	917
POLLERR	917
POLLHUP	917
POLLIN	917
POLLNVAL	918
POLLOUT	917
POLLPRI	917
POLLRDBAND	917
POLLRDNORM	917
POLLWRBAND	917
POLLWRNORM	917
POLL_	17
popen()	921
portability	3
POSIX.1 symbols	14
POSIX2_SYMLINKS	418
POSIX_	15
posix_	15
POSIX_	17
posix_	17
POSIX_ALLOC_SIZE_MIN	418
posix_fadvise()	924
POSIX_FADV_DONTNEED	924
POSIX_FADV_NOREUSE	924
POSIX_FADV_NORMAL	924
POSIX_FADV_RANDOM	924
POSIX_FADV_SEQUENTIAL	924
POSIX_FADV_WILLNEED	924
posix_fallocate()	926
posix_madvise()	928
POSIX_MADV_DONTNEED	928
POSIX_MADV_NORMAL	928
POSIX_MADV_RANDOM	928
POSIX_MADV_SEQUENTIAL	928
POSIX_MADV_WILLNEED	928
posix_memalign()	932
posix_mem_offset()	930
posix_openpt()	933
POSIX_REC_INCR_XFER_SIZE	418
POSIX_REC_MAX_XFER_SIZE	418
POSIX_REC_MIN_XFER_SIZE	418
POSIX_REC_XFER_ALIGN	418
posix_spawn()	935
posix_spawnattr_destroy()	950
posix_spawnattr_getflags()	952
posix_spawnattr_getpgroup()	954
posix_spawnattr_getschedparam()	956
posix_spawnattr_getschedpolicy()	958
posix_spawnattr_getsigdefault()	960
posix_spawnattr_getsigmask()	962
posix_spawnattr_init	950
posix_spawnattr_init()	964
posix_spawnattr_setflags	952
posix_spawnattr_setflags()	965
posix_spawnattr_setpgroup	954
posix_spawnattr_setpgroup()	966
posix_spawnattr_setschedparam	956
posix_spawnattr_setschedparam()	967
posix_spawnattr_setschedpolicy	958
posix_spawnattr_setschedpolicy()	968

posix_spawnattr_setsigdefault.....	960	posix_trace_close().....	993
posix_spawnattr_setsigdefault()	969	POSIX_TRACE_CLOSE_FOR_CHILD.....	976
posix_spawnattr_setsigmask.....	962	posix_trace_create()	995
posix_spawnattr_setsigmask().....	970	posix_trace_create_withlog.....	995
posix_spawnnp	935	POSIX_TRACE_ERROR trace event.....	78
posix_spawnnp().....	971	posix_trace_event().....	999
posix_spawn_file_actions_addclose().....	943	posix_trace_eventid_equal()	1001
posix_spawn_file_actions_adddup2()	946	posix_trace_eventid_get_name	1001
posix_spawn_file_actions_addopen.....	943	posix_trace_eventid_open.....	999
posix_spawn_file_actions_addopen()	948	posix_trace_eventid_open()	1003
posix_spawn_file_actions_destroy().....	949	posix_trace_eventset_add().....	1004
posix_spawn_file_actions_init.....	949	posix_trace_eventset_del.....	1004
POSIX_SPAWN_RESETIDS	936, 952	posix_trace_eventset_empty.....	1004
POSIX_SPAWN_SETPGROUP	936, 952, 954	posix_trace_eventset_fill	1004
POSIX_SPAWN_SETSCHEDPARAM	952, 956	posix_trace_eventset_ismember.....	1004
POSIX_SPAWN_SETSCHEDULETIME.....	936, 952, 958	posix_trace_eventtypelist_getnext_id().....	1006
POSIX_SPAWN_SETSIGDEF.....	937, 952, 960	posix_trace_eventtypelist_rewind	1006
POSIX_SPAWN_SETSIGMASK	952, 962	posix_trace_event_info structure	75
POSIX_TRACE_ADD_EVENTSET	1011	POSIX_TRACE_FILTER trace event	78, 1011
POSIX_TRACE_ALL_EVENTS.....	1004	POSIX_TRACE_FLUSH	977
POSIX_TRACE_APPEND.....	977, 997	posix_trace_flush.....	995
posix_trace_attr_destroy().....	972	posix_trace_flush()	1008
posix_trace_attr_getclockres().....	974	POSIX_TRACE_FLUSHING.....	74
posix_trace_attr_getcreatetime.....	974	POSIX_TRACE_FULL	73-75
posix_trace_attr_getgenversion	974	posix_trace_getnext_event().....	1014
posix_trace_attr_getinherited().....	976	posix_trace_get_attr().....	1009
posix_trace_attr_getlogfullpolicy.....	976	posix_trace_get_filter().....	1011
posix_trace_attr_getlogsize()	979	posix_trace_get_status.....	1009
posix_trace_attr_getmaxdatasize.....	979	posix_trace_get_status().....	1013
posix_trace_attr_getmaxsystemeventsz.....	979	POSIX_TRACE_INHERITED	976
posix_trace_attr_getmaxusereventsz.....	979	POSIX_TRACE_LOOP	74, 976-977, 996
posix_trace_attr_getname	974	POSIX_TRACE_NOT_FLUSHING.....	75
posix_trace_attr_getname().....	982	POSIX_TRACE_NOT_FULL	73-75
posix_trace_attr_getstreamfullpolicy	976	POSIX_TRACE_NOT_FULL	991
posix_trace_attr_getstreamfullpolicy().....	983	POSIX_TRACE_NOT_TRUNCATED	76, 1015
posix_trace_attr_getstreamsize	979	POSIX_TRACE_NO_OVERRUN.....	74-75, 1009
posix_trace_attr_getstreamsize().....	984	posix_trace_open.....	993
posix_trace_attr_init.....	972	posix_trace_open().....	1017
posix_trace_attr_init()	985	POSIX_TRACE_OVERFLOW trace event	78
posix_trace_attr_setinherited	976	POSIX_TRACE_OVERRUN	74-75
posix_trace_attr_setinherited()	986	POSIX_TRACE_RESUME trace event.....	78
posix_trace_attr_setlogfullpolicy	976, 986	posix_trace_rewind	993, 1017
posix_trace_attr_setlogsize	979	POSIX_TRACE_RUNNING	73-74, 1020
posix_trace_attr_setlogsize().....	987	POSIX_TRACE_SET_EVENTSET	1011
posix_trace_attr_setmaxdatasize	979, 987	posix_trace_set_filter.....	1011
posix_trace_attr_setname.....	974	posix_trace_set_filter()	1018
posix_trace_attr_setname().....	988	posix_trace_shutdown.....	995
posix_trace_attr_setstreamfullpolicy.....	976	posix_trace_shutdown()	1019
posix_trace_attr_setstreamfullpolicy()	989	POSIX_TRACE_START trace event.....	78, 1020
posix_trace_attr_setstreamsize.....	979	posix_trace_start().....	1020
posix_trace_attr_setstreamsize().....	990	posix_trace_status_info structure	73
posix_trace_clear().....	991	posix_trace_stop	1020
		POSIX_TRACE_STOP trace event	78, 1020

Index

POSIX_TRACE_SUB_EVENTSET	1011
POSIX_TRACE_SUSPENDED	73-74, 1020
POSIX_TRACE_SYSTEM_EVENTS	1004
posix_trace_timedgetnext_event.....	1014
posix_trace_timedgetnext_event()	1022
posix_trace_trid_eventid_open	1001
posix_trace_trid_eventid_open().....	1023
POSIX_TRACE_TRUNCATED_READ	76, 1015
POSIX_TRACE_TRUNCATED_RECORD.....	76, 1015
posix_trace_trygetnext_event.....	1014
posix_trace_trygetnext_event().....	1024
POSIX_TRACE_UNTIL_FULL	74, 976-977, 996
POSIX_TRACE_USER_EVENT_MAX	999
POSIX_TRACE_WOPID_EVENTS.....	1004
POSIX_TYPED_MEM_ALLOCATE	1025, 1027
POSIX_TYPED_MEM_ALLOCATE	1025, 1027
posix_typed_mem_get_info()	1025
POSIX_TYPED_MEM_MAP_ALLOCATE.....	1027
posix_typed_mem_open().....	1027
pow().....	1030
powf.....	1030
powl.....	1030
pread.....	1245
pread()	1033
predefined stream	
standard error	36
standard input	36
standard output	36
preempted thread.....	1099
PRI	17
printf.....	424
printf()	1034
priority	38
PRIO_	17
PRIO_INHERIT.....	1152
PRIO_PGRP.....	570
PRIO_PROCESS.....	570
PRIO_USER.....	570
process	
concurrent execution.....	415
setting real and effective user IDs.....	1386
single-threaded	415
process creation	415
process group	
orphaned.....	90
process group ID	566, 1378, 1390
process ID, 1	90
process lifetime	717
process scheduling	44
process shared memory	1157
process synchronization	1157
process termination.....	89
PROT_	16-17
PROT_EXEC.....	826, 835
PROT_NONE.....	43, 825-826, 835
PROT_READ.....	826, 835
PROT_WRITE.....	826-827, 830, 835
PS	6
pselect()	1035
pseudo-random sequence generation function.....	1242
psiginfo().....	1040
psignal.....	1040
psignal()	1041
PST8PDT	1636
ps	16
PTHREAD_	16
pthread	16
pthread_atfork().....	1042
pthread_attr_destroy()	1044
pthread_attr_getdetachstate().....	1047
pthread_attr_getguardsize().....	1049
pthread_attr_getinheritsched().....	1052
pthread_attr_getschedparam()	1054
pthread_attr_getschedpolicy().....	1056
pthread_attr_getscope().....	1058
pthread_attr_getstack().....	1060
pthread_attr_getstacksize().....	1062
pthread_attr_init.....	1044
pthread_attr_init()	1064
pthread_attr_setdetachstate.....	1047
pthread_attr_setdetachstate()	1065
pthread_attr_setguardsize	1049
pthread_attr_setguardsize()	1066
pthread_attr_setinheritsched.....	1052
pthread_attr_setinheritsched().....	1067
pthread_attr_setschedparam	1054
pthread_attr_setschedparam().....	1068
pthread_attr_setschedpolicy.....	1056
pthread_attr_setschedpolicy()	1069
pthread_attr_setscope.....	1058
pthread_attr_setscope()	1070
pthread_attr_setstack.....	1060
pthread_attr_setstack()	1071
pthread_attr_setstacksize	1062
pthread_attr_setstacksize().....	1072
pthread_barrierattr_destroy()	1077
pthread_barrierattr_getpshared().....	1079
pthread_barrierattr_init.....	1077
pthread_barrierattr_init()	1081
pthread_barrierattr_setpshared	1079
pthread_barrierattr_setpshared()	1082
pthread_barrier_destroy()	1073
pthread_barrier_init.....	1073
PTHREAD_BARRIER_SERIAL_THREAD.....	1075
pthread_barrier_wait().....	1075

<code>pthread_cancel()</code>	1083
<code>PTHREAD_CANCEL_CANCELED</code>	58, 1119
<code>PTHREAD_CANCEL_ASYNCHRONOUS</code>	54, 1203
<code>PTHREAD_CANCEL_DEFERRED</code>	310, 1097, 1203
<code>PTHREAD_CANCEL_DISABLE</code>	54, 57, 1203
<code>PTHREAD_CANCEL_ENABLE</code>	54, 58, 1203
<code>PTHREAD_CANCEL_ENABLED</code>	310
<code>pthread_cleanup_pop()</code>	1085
<code>pthread_cleanup_push</code>	1085
<code>pthread_condattr_destroy()</code>	1103
<code>pthread_condattr_getclock()</code>	1105
<code>pthread_condattr_getpshared()</code>	1107
<code>pthread_condattr_init</code>	1103
<code>pthread_condattr_init()</code>	1109
<code>pthread_condattr_setclock</code>	1105
<code>pthread_condattr_setclock()</code>	1110
<code>pthread_condattr_setpshared</code>	1107
<code>pthread_condattr_setpshared()</code>	1111
<code>pthread_cond_broadcast()</code>	1090
<code>pthread_cond_destroy()</code>	1093
<code>pthread_cond_init</code>	1093
<code>PTHREAD_COND_INITIALIZER</code>	1093
<code>pthread_cond_signal</code>	1090
<code>pthread_cond_signal()</code>	1096
<code>pthread_cond_timedwait()</code>	1097
<code>pthread_cond_wait</code>	1097
<code>pthread_create()</code>	1112
<code>PTHREAD_CREATE_DETACHED</code>	30, 1047
<code>PTHREAD_CREATE_JOINABLE</code>	30, 1047, 1129
<code>PTHREAD_DESTRUCTOR_ITERATIONS</code>	1120, 1158
<code>pthread_detach()</code>	1115
<code>pthread_equal()</code>	1117
<code>pthread_exit()</code>	1118
<code>PTHREAD_EXPLICIT_SCHED</code>	1052
<code>pthread_getconcurrency()</code>	1120
<code>pthread_getcpuclockid()</code>	1122
<code>pthread_getschedparam()</code>	1123
<code>pthread_getspecific()</code>	1126
<code>PTHREAD_INHERIT_SCHED</code>	1052
<code>pthread_join()</code>	1128
<code>PTHREAD_KEYS_MAX</code>	1131, 1158
<code>pthread_key_create()</code>	1131
<code>pthread_key_delete()</code>	1134
<code>pthread_kill()</code>	1136
<code>pthread_mutexattr_destroy()</code>	1156
<code>pthread_mutexattr_getprioceiling()</code>	1161
<code>pthread_mutexattr_getprotocol()</code>	1163
<code>pthread_mutexattr_getpshared()</code>	1166
<code>pthread_mutexattr_getrobust()</code>	1168
<code>pthread_mutexattr_gettype()</code>	1170
<code>pthread_mutexattr_init</code>	1156
<code>pthread_mutexattr_init()</code>	1172
<code>pthread_mutexattr_setprioceiling</code>	1161
<code>pthread_mutexattr_setprioceiling()</code>	1173
<code>pthread_mutexattr_setprotocol</code>	1163
<code>pthread_mutexattr_setprotocol()</code>	1174
<code>pthread_mutexattr_setpshared</code>	1166
<code>pthread_mutexattr_setpshared()</code>	1175
<code>pthread_mutexattr_setrobust</code>	1168
<code>pthread_mutexattr_setrobust()</code>	1176
<code>pthread_mutexattr_settype</code>	1170
<code>pthread_mutexattr_settype()</code>	1177
<code>pthread_mutex_consistent()</code>	1137
<code>PTHREAD_MUTEX_DEFAULT</code>	1147, 1170
<code>pthread_mutex_destroy()</code>	1139
<code>PTHREAD_MUTEX_ERRORCHECK</code> ..	1147, 1170
<code>pthread_mutex_getprioceiling()</code>	1144
<code>pthread_mutex_init</code>	1139
<code>pthread_mutex_init()</code>	1146
<code>PTHREAD_MUTEX_INITIALIZER</code>	1139
<code>pthread_mutex_lock()</code>	1147
<code>PTHREAD_MUTEX_NORMAL</code>	1147, 1170
<code>PTHREAD_MUTEX_RECURSIVE</code>	1147, 1170-1171
<code>PTHREAD_MUTEX_ROBUST</code>	1168
<code>pthread_mutex_setprioceiling</code>	1144
<code>pthread_mutex_setprioceiling()</code>	1151
<code>PTHREAD_MUTEX_STALLED</code>	1168
<code>pthread_mutex_timedlock()</code>	1152
<code>pthread_mutex_trylock</code>	1147
<code>pthread_mutex_trylock()</code>	1155
<code>pthread_mutex_unlock</code>	1147, 1155
<code>pthread_once()</code>	1178
<code>PTHREAD_ONCE_INIT</code>	1178
<code>PTHREAD_PRIO_INHERIT</code>	1163
<code>PTHREAD_PRIO_NONE</code>	1144, 1163
<code>PTHREAD_PRIO_PROTECT</code>	1148, 1163
<code>PTHREAD_PROCESS_PRIVATE</code>	1166, 1198, 1214
<code>PTHREAD_PROCESS_SHARED</code>	1166, 1198, 1214
<code>pthread_rwlockattr_destroy()</code>	1196
<code>pthread_rwlockattr_getpshared()</code>	1198
<code>pthread_rwlockattr_init</code>	1196
<code>pthread_rwlockattr_init()</code>	1200
<code>pthread_rwlockattr_setpshared</code>	1198
<code>pthread_rwlockattr_setpshared()</code>	1201
<code>pthread_rwlock_destroy()</code>	1180
<code>pthread_rwlock_init</code>	1180
<code>pthread_rwlock_rdlock()</code>	1183
<code>pthread_rwlock_timedrdlock()</code>	1186
<code>pthread_rwlock_timedwrlock()</code>	1188
<code>pthread_rwlock_tryrdlock</code>	1183
<code>pthread_rwlock_tryrdlock()</code>	1190
<code>pthread_rwlock_trywrlock()</code>	1191
<code>pthread_rwlock_unlock()</code>	1193
<code>pthread_rwlock_wrlck</code>	1191
<code>pthread_rwlock_wrlck()</code>	1195

Index

- PTHREAD_SCOPE_PROCESS52-53, 1058
 PTHREAD_SCOPE_SYSTEM.....52, 1058
 pthread_self()**1202**
 pthread_setcancelstate()**1203**
 pthread_setcanceltype1203
 pthread_setconcurrency1120
 pthread_setconcurrency().....**1205**
 pthread_setschedparam.....1123
 pthread_setschedparam().....**1206**
 pthread_setschedprio().....**1207**
 pthread_setspecific1126
 pthread_setspecific()349, 821, 823, 832, 838, 840, 842, 843, 848, 850, 854, 1307-1311, 1313, 1402, 1406
 pthread_sigmask().....1052, 1056, 1058, 1067, 1109-1070
 pthread_spin_destroy().....**1214**
 pthread_spin_init1214
 pthread_spin_lock().....**1216**
 pthread_spin_trylock.....1216
 pthread_spin_unlock().....**1218**
 PTHREAD_STACK_MIN.....1060, 1062, 1558
 pthread_testcancel.....1203
 pthread_testcancel()**1219**
 PTHREAD_THREADS_MAX.....1112, 1558
 ptsname()**1220**
 putc().....**1221**
 putchar().....**1223**
 putchar_unlocked.....514
 putchar_unlocked()**1224**
 putc_unlocked514
 putc_unlocked()**1222**
 putenv().....**1225**
 putmsg().....**1227**
 putpmsg.....1227
 puts().....**1231**
 pututxline295
 pututxline().....**1233**
 putwc()**1234**
 putwchar()**1235**
 pwrite1756
 pwrite().....**1236**
 pw_16
 P_16
 P_17
 P_ALL1683
 P_PGID1683
 P_PID.....1683
 qsort()**1237**
 queue a signal to a process.....1445
 raise()**1239**
 rand()**1241**
 random.....634
 random()**1244**
 RAND_MAX1241
 rand_r.....1241
 read from a file.....1248
 read().....**1245**
 readdir()**1252**
 readdir_r1252
 readlink().....**1256**
 readlinkat.....1256
 readlinkat()**1259**
 readv()**1260**
 real user ID104, 716
 realloc().....**1262**
 realpath().....**1264**
 REALTIME_THREADS1163, 1173-1174, 1206-1207
 recv().....**1266**
 recvfrom()**1268**
 recvmsg().....**1271**
 regcomp().....**1274**
 regerror1274
 regexexec1274
 regfree1274
 register fork handlers.....1042
 REG17
 REG_constants
 error return values of regcomp1276
 used in regcomp1274
 REG_BADBR1276
 REG_BADPAT.....1276
 REG_BADRPT1276
 REG_EBRACE1276
 REG_EBRACK1276
 REG_ECOLLATE.....1276
 REG_ECTYPE1276
 REG_EESCAPE1276
 REG_EPAREN.....1276
 REG_ERANGE.....1276
 REG_ESPACE.....1276
 REG_ESUBREG1276
 REG_EXTENDED.....1274
 REG_ICASE.....1274
 REG_NEWLINE1274
 REG_NOMATCH1276
 REG_NOSUB.....1274
 REG_NOTBOL.....1275
 REG_NOTEOL.....1275
 remainder()**1281**
 remainderf1281
 remainderl1281
 remove a directory1299
 remove directory entries1651
 remove()**1283**
 remque636
 remque()**1285**

Index

SEEK_SET	41, 116, 123, 342, 465, 778
SEGV_	17
select	1035
select()	1319
semctl()	1340
semget()	1343
semid	39
semop()	1346
SEM_	16
sem_	16
SEM_	17
sem_close()	1320
sem_destroy()	1322
SEM_FAILED	1329
sem_getvalue()	1324
sem_init()	1326
SEM_NSEMS_MAX	1326, 1558
sem_open()	1328
sem_perm	40
sem_post()	1331
sem_timedwait()	1333
sem_trywait()	1335
SEM_UNDO	1346
sem_unlink()	1337
SEM_VALUE_MAX	1326, 1328, 1558
sem_wait	1335
sem_wait()	1339
send()	1351
sendmsg()	1353
sendto()	1356
service name	446
session	90, 716, 1378, 1390
set cancelability state	1203
set file creation mask	1640
set process group ID for job control	1378
set-group-ID	89, 195, 315, 347
set-user-ID	89, 315, 519, 716
SETALL	1340, 1343
setbuf()	1359
setegid()	1360
setenv()	1361
seteuid()	1363
setgid()	1364
setgrent	283
setgrent()	1366
sethostent	285
sethostent()	1367
setitimer	544
setitimer()	1368
setjmp()	1369
setkey()	1371
setlocale()	1372
setlogmask	218
setlogmask()	1376
setnetent	287
setnetent()	1377
setpgid()	1378
setpgrp()	1380
setpriority	570
setpriority()	1381
setprotoent	289
setprotoent()	1382
setpwent	291
setpwent()	1383
setregid()	1384
setreuid()	1386
setrlimit	581
setrlimit()	1388
setservent	293
setservent()	1389
setsid()	1390
setsockopt()	1392
setstate	634
setstate()	1395
setuid()	1396
setuxent	295
setuxent()	1399
SETVAL	1340, 1343
setvbuf()	1400
shall	2
shell	90, 314, 548, 566, 716, 1379, 1680
job	716
login	548
shell scripts	
exec	314
shell, login	314
SHM	7, 17
shmat()	1408
shmctl()	1410
shmdt()	1412
shmget()	1413
shmid	39
SHMLBA	1408
shm_	16
SHM_	17
shm_open()	1402
shm_perm	40
SHM_RDONLY	1408
SHM_RND	1408
shm_unlink()	1406
should	2
shutdown()	1415
SHUT_	17
SIGABRT	98
sigaction()	1417

sigaddset()	1424	SIGSTOP	29, 1417, 1422, 1431
SIGALRM	126, 544, 1463	sigsuspend()	1449
sigaltstack()	1425	sigtimedwait()	1451
SIGBUS	43, 827, 830, 1210	SIGTSTP	29
SIGCANCEL	29, 1089, 376, 436, 440, 466, 468, 1574, 1576, 1578, 1585, 1588, 1758	SIGTTIN	29, 380, 386, 1247
87-88, 29, 584, 609, 1417, 1421, 1431, 1564, 1676, 1683		SIGURG	641
SIGCLD	1421	SIGVTALRM	544
SIGCONT	33, 88, 90, 715-716	sigwait()	1455
sigdelset()	1427	sigwaitinfo()	1451
sigemptyset()	1428	sigwaitinfo()	1457
SIGEV_	16	SIGXCPU	581
sigev_	16	SIGXFSZ	581, 1629
SIGEV_NONE	29, 42	SIG_	16-17
SIGEV_SIGNAL	29, 1604	SIG_BLOCK	1210
SIGEV_THREAD	30, 737	SIG_DFL	31, 308, 582, 1417, 1419, 1438
sigfillset()	1430	SIG_ERR	1439
SIGFPE	1210, 1438	SIG_HOLD	1431
sighold()	1431	SIG_IGN	31, 87-88, 308, 315, 584, 1417, 1438, 1676
SIGHUP	87-88, 90, 213	SIG_SETMASK	1210
sigignore	1431	SIG_UNBLOCK	1210
SIGILL	1210, 1438	sin()	1458
SIGINT	415, 1564	sin6	16
siginterrupt()	1434	sinf	1458
sigismember()	1436	sinh()	1460
SIGKILL	716, 1417, 1421-1422, 1431	sinhf	1460
siglongjmp()	1437	sinhl	1460
signal generation and delivery	28	sinl	1458
realtime	29	sinl()	1462
signal handler	1438	sin_	16
signal()	1438	SIO	7
signaling a condition	1091	SIOCATMARK	1466
signals	28	sival_	16
signbit()	1440	SI_	16
sigpause	1431	si_	16
sigpause()	1441	SI_	17
sigpending()	1442	si_	17
SIGPIPE	340, 376, 436, 441, 466, 469, 1228, 1759	SI_ASYNCIO	32
SIGPOLL	213, 640-641	SI_MESGQ	32
sigprocmask	1210	SI_QUEUE	32
sigprocmask()	1443	SI_TIMER	32
SIGPROF	544	SI_USER	32
sigqueue()	1444	sleep()	1463
SIGQUEUE_MAX	1444, 1558	sl_	16
SIGQUIT	1564	SND	17
sigrelse	1431	SNDZERO	644
sigrelse()	1446	snprintf	424
SIGRTMAX	29-30, 1420, 1444, 1451, 1455	snprintf()	1465
SIGRTMIN	29-30, 1420, 1444, 1451, 1455	SO	17
SIGSEGV	43, 582, 872, 1049, 1210, 1438	socketatmark()	1466
sigset	1431, 1446	socket I/O mode	62
sigsetjmp()	1447	socket out-of-band data	63
SIGSTKSZ	1425	socket owner	62

Index

socket queue limits	62
socket receive queue	62
socket types	61
socket()	1468
socketpair()	1470
sockets	60
address families	60
addressing	60
asynchronous errors	64
connection indication queue	63
Internet Protocols	67
IPv4	68
IPv6	68
local UNIX connections	67
options	64
pending error	62
protocols	60
signals	63
SOCK_	17
SOCK_DGRAM	67, 1468, 1470
SOCK_RAW	67
SOCK_SEQPACKET	68, 1468, 1470
SOCK_STREAM	67, 1468, 1470
SPN	7
sporadic server policy	
execution capacity	46
replenishment period	46
sprintf	424
sprintf()	1472
spurious wakeup	1091
sqrt()	1473
sqrtf	1473
sqrtl	1473
srand	1241
srand()	1475
srand48	274
srand48()	1476
random	634
random()	1477
SS	7
sscanf	458
sscanf()	1478
SSIZE_MAX	845, 861, 1245, 1256, 1503, 1756
ss_	16
SS_	17
SS_DISABLE	1425-1426
SS_ONSTACK	1425
stack size	1044
stat	473
stat()	1479
statvfs	478
statvfs()	1480
stderr	1481
STDERR_FILENO	1481
stdin	1481
STDIN_FILENO	921, 1481
stdio locking functions	391
stdio with explicit client locking	514
stdout	1481
STDOUT_FILENO	921, 1481
stpcpy	1493
stpcpy()	1483
stpncpy	1516
stpncpy()	1484
STR	17
strcasecmp()	1485
strcasecmp_l	1485
strcat()	1487
strchr()	1488
strcmp()	1489
strcoll()	1491
strcoll_l	1491
strcpy()	1493
strncpy()	1496
strdup()	1497
STREAM	643, 645, 1227, 1246, 1757
stream	
byte-oriented	36
wide-oriented	36
STREAM head/tail	38
stream-full-policy attribute	74-75, 77, 977
stream-min-size attribute	77, 980
STREAMS	22
streams	34
STREAMS	213, 329, 350, 550, 639, 655, 896, 917, 1035
access	39
streams	
interaction with file descriptors	35
STREAMS	
multiplexed	646
overview	38
streams	
stream orientation	36
STREAM_MAX	354, 410, 1558
strerror()	1499
strerror_l	1499
strerror_r	1499
strfmon()	1501
strfmon_l	1501
strftime()	1505
strftime_l	1505
strlen()	1511
strncasecmp	1485
strncasecmp()	1513
strncasecmp_l	1485, 1513

156, 228, 330, 350, 410, 453, 479, 485, 491, 512, 513, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 847, 848, 849, 850, 851, 852, 853, 854, 855, 856, 857, 858, 859, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 870, 871, 872, 873, 874, 875, 876, 877, 878, 879, 880, 881, 882, 883, 884, 885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 896, 897, 898, 899, 900, 901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915, 916, 917, 918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000	SYMLINK_MAX.....418, 1551	sync()1555	synchronously accept a signal.....1452
strncat().....1514		sysconf().....1556	syslog.....218
strncmp().....1515		syslog().....1563	system crash.....481
strncpy().....1516		System III.....199, 1642	system interfaces.....85
strndup.....1497		system name.....1642	system trace event type definitions.....77
strndup().....1518		system().....1564	s.....16
strnlen().....1519		S.....17	S_BANDURG.....641
strpbrk().....1520		S_ERROR.....641	S_HANGUP.....641
strptime().....1521		S_HIPRI.....640	S_IFBLK.....813
strrchr().....1526		S_IFCHR.....813	S_IFDIR.....813
strsignal().....1527		S_IFIFO.....813	S_IFREG.....813
strspn().....90, 126, 199, 316, 328, 420, 566, 571, 804, 1299, 1390, 1421-1422, 1447, 1580, 1642		S_INPUT.....640	S_IRGRP.....333, 470, 473, 813
strstr().....1529		S_IROTH.....333, 470, 473, 813	S_IRUSR.....333, 470, 473, 813
strtod().....1530		S_IRWXG.....813	S_IRWXO.....813
strtof.....1530		S_IRWXU.....813	S_ISGID.....193, 195, 813, 1629, 1757
strtoimax().....1534		S_ISUID.....193, 195, 813, 1629, 1757	S_ISVTX.....193, 813, 1290, 1299, 1649
strtok().....1535		S_IWGRP.....333, 470, 473, 813	S_IWOTH.....333, 470, 473, 813
strtok_r.....1535		S_IWUSR.....333, 470, 473, 813	S_IXGRP.....813
strtol().....1538		S_IXOTH.....813	S_IXUSR.....813
strtold.....1530		S_MSG.....640	S_OUTPUT.....640
strtold().....1540		S_OUTPUT.....640	S_RDBAND.....640-641
strtoll.....1538		S_RDNORM.....640	S_WRBAND.....640
strtoll().....1541		S_WRNORM.....640	TABSIZE.....157, 776
strtol().....1542		TABSIZE.....157, 776	tan().....1569
strtoull.....1542		tanf.....1569	tanh().....1571
strtoumax.....1534		tanhf.....1571	tanh.....1571
strtoumax().....1545		tanl.....1571	tanl.....1569
strxfrm().....1546			
strxfrm_l.....1546			
str.....16			
st.....16			
ST.....17			
ST_NOSUID.....309, 478			
ST_RDONLY.....478			
sun.....17			
superuser.....104, 199, 733, 1651			
supplementary groups.....199, 540			
SVID.....1447			
SVR4.....829, 875			
sv.....16			
SV.....17			
swab().....1548			
swprintf.....496			
swprintf().....1549			
swscanf.....505			
swscanf().....1550			
SWTCH.....19			
symbols			
POSIX.1.....14			
symlink().....1551			
symlinkat.....1551			
symlinkat().....1554			

Index

tanl()	1573
tcdrain()	1574
tcflow()	1576
tcflush()	1578
tcgetattr()	1580
tcgetpgrp()	1582
tcgetsid()	1584
TCIFLUSH	1578
TCIOFF	1576
TCIOFLUSH	1578
TCION	1576
TCOFLUSH	1578
TCOOFF	1576
TCOON	1576
TCP_	17
TCSADRAIN	1587
TCSAFLUSH	1587
TCSANOW	1587
tcsendbreak()	1585
tcsetattr()	1587
tcsetpgrp()	1590
TCT	7
tdelete()	1592
TEF	7
telldir()	1596
tempnam()	1597
terminal access control	1580, 1588
terminal device name	1633
terminate a process	89
terminology	1
termios structure	1580
tfind	1592
tfind()	1599
tgamma()	1600
tgammaf	1600
tgammaL	1600
thread cancellation	
cleanup handlers	57
thread creation	1113
thread creation attributes	1044
thread ID	51, 1117
972, 974, 976, 978, 982, 991, 993, 995, 999, 1001, 1003, 1004, 1006, 1008, 1009, 1011, 1013-1014, 1017-1020, 1022-1024	
thread scheduling	52
thread termination	1118
thread-safety	50, 391
thread-specific data key creation	1132
thread-specific data key deletion	1134
thread-specific data management	1127
threads	50
regular file operations	59
time()	1602
timer ID	1606
TIMER_	17
timer_	17
TIMER_ABSTIME	49, 208, 1608
timer_create()	1604
timer_delete()	1607
timer_getoverrun()	1608
timer_gettime	1608
TIMER_MAX	1558
timer_settime	1608
times()	1611
timezone()	1613
tmpfile()	1614
tmpnam()	1616
TMP_MAX	1597, 1615-1616
tms_	16
tm_	17
toascii()	1618
tolower()	1619
tolower_l	1619
TOSTOP	339, 376, 436, 440, 466, 468, 1758
toupper()	1621
toupper_l	1621
towctrans()	1622
towctrans_l	1622
towlower()	1624
towlower_l	1624
towupper()	1626
towupper_l	1626
TPI	7
TPP	7
TPS	8
trace event, POSIX_TRACE_ERROR	78
trace event, POSIX_TRACE_FILTER	78, 1011
trace event, POSIX_TRACE_OVERFLOW	78
trace event, POSIX_TRACE_RESUME	78
trace event, POSIX_TRACE_START	78, 1020
trace event, POSIX_TRACE_STOP	78, 1020
trace functions	81
trace-name attribute	77, 974
TRACE_EVENT_NAME_MAX	999, 1001
TRACE_SYS_MAX	996
TRACE_USER_EVENT_MAX	999, 1001
TRACENO	909, 1011, 1013-1014, 1017-1020, 1022-1024
TRAP_	17
TRC	8
TRI	8
TRL	8
trunc()	1628
truncate()	1629
truncation-status attribute	999
truncf	1628
truncf()	1631
truncL	1628, 1631

TSA	8	utim_	17
tsearch	1592	uts_	17
tsearch()	1632	ut_	17
TSH	8	UU	9
TSP	8	va_arg()	1663
TSS	9	va_copy	1663
ttyname()	1633	va_end	1663
ttyname_r	1633	va_start	1663
TTY_NAME_MAX	1558, 1633	VDISCARD	19
tv_	16-17	VDSUSP	19
twalk	1592	Version 7	126, 199, 716, 1642
twalk()	1635	vfprintf()	1664
TYM	9	vfscanf()	1665
tzname	1636	vfwprintf()	1666
TZNAME_MAX	1558	vfwscanf()	1667
tzset	1636	VISIT	1592, 1635
tzset()	1636	VLNEXT	19
t_uscalar_t	642	vprintf	1664
uc_	16	vprintf()	1668
UINT	18	VREPRINT	19
UINT_MAX	126, 1464	vscanf	1665
UIO_MAXIOV	17	vscanf()	1669
ulimit()	1638	vsnprintf	1664
ULLONG_MAX	1543	vsnprintf()	1670
ULONG_MAX	1543, 1732	vsprintf	1664, 1670
UL_	17	vsscanf	1665
UL_GETFSIZE	1638	vsscanf()	1671
UL_SETFSIZE	1638	VSTATUS	19
umask()	1640	vswprintf	1666
uname()	1642	vswprintf()	1672
undefined	2	vswscanf	1667
underlying function	36	vswscanf()	1673
ungetc()	1644	VWERASE	19
ungetwc()	1646	vwprintf	1666
unicast	68	vwprintf()	1674
unlink()	1648	vwscanf	1667
unlinkat	1648	vwscanf()	1675
unlinkat()	1653	wait for process termination	1680
unlockpt()	1654	wait for thread termination	1129
unsetenv()	1655	wait()	1676
unspecified	2	waitid()	1683
UP	9	waiting on a condition	1099
US-ASCII	654	waitpid	1676
uselocale()	1656	waitpid()	1685
user ID		WARNING	405
real and effective	1386	warning	
setting real and effective	1386	OB	5
user trace event type definitions	80	OF	5
USER_PROCESS	295-296	WCONTINUED	1676, 1683
UTC	1636	wcpcpy	1697
utime()	1658	wcpcpy()	1686
utimes()	1660	wcpcpy	1706
		wcpcpy()	1687

Index

	83, 684, 686, 690, 692, 694, 696, 698, 700, 702, 704, 706, 708, 1624, 1626, 1646	
wcrtomb()	1688	WEXITED
wscasecmp()	1690	WEXITSTATUS
wscasecmp_l	1690	we_
wscat()	1692	wide-oriented stream
wchr()	1693	WIFCONTINUED
wscmp()	1694	WIFEXITED
wscoll()	1695	WIFSIGNALED
wscoll_l	1695	WIFSTOPPED
wscopy()	1697	wmemchr()
wscspn()	1698	wmemcmp()
wscdup()	1699	wmemcpy()
wcsftime()	1700	wmemmove()
wcslen()	1702	wmemset()
wcscasecmp	1690	WNOHANG
wcscasecmp_l	1703	WNOWAIT
wcscat()	1690, 1703	wordexp()
wcscat_l	1704	wordfree
wcscmp()	1705	wprintf
wcscpy()	1706	wprintf()
wcsnlen	1702	WRDE_
wcsnlen_l	1708	WRDE_APPEND
wcsnrtombs	1712	WRDE_BADCHAR
wcsnrtombs_l	1709	WRDE_BADVAL
wcsprbrk()	1710	WRDE_CMDSUB
wcsrchr()	1711	WRDE_DOOFFS
wcsrtombs()	1712	WRDE_NOCMD
wcsspn()	1714	WRDE_NOSPACE
wcsstr()	1715	WRDE_REUSE
wcstod()	1716	WRDE_SHOWERR
wcstof	1716	WRDE_SYNTAX
wcstoimax()	1720	WRDE_UNDEF
wcstok()	1722	write to a file
wcstol()	1724	write()
wcstold	1716	writew()
wcstold_l	1727	wscanf
wcstoll	1724	wscanf()
wcstoll_l	1728	WSTOPPED
wcstombs()	1729	WSTOPSIG
wcstoul()	1731	WTERMSIG
wcstoull	1731	WUNTRACED
wcstoumax	1720	XSI
wcstoumax_l	1734	XSI interprocess communication
wcswidth()	1735	XSR
wcsxfrm()	1736	X_OK
wcsxfrm_l	1736	y0()
wctob()	1738	y1
wctomb()	1739	yn
wctrans()	1741	zombie process
wctrans_l	1741	
wctype()	1743	
wctype_l	1743	
wcwidth()	1745	

