Chapter 1

Introduction

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 Changes from Issue 4

Notes to Reviewers

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

The change history is subject to revision. The intent is to document changes from Issue 4 thru Issue 6, with the latest change history also documenting changes from the ISO POSIX-1:1996 standard.

The following sections describe changes made to this volume of IEEE Std. 1003.1-200x since Issue 4. The CHANGE HISTORY section for each entry details the technical changes that have been made to that entry since Issue 4. Changes made between Issue 2 and Issue 4 are not included.

1.4.1 Changes from Issue 4 to Issue 4, Version 2

The following list summarizes the major changes that were made in this volume of IEEE Std. 1003.1-200x from Issue 4 to Issue 4, Version 2:

- The X/Open UNIX extension was added. This specifies the common core APIs of 4.3 Berkeley Software Distribution (BSD 4.3), the OSF AES, and SVID Issue 3.
- STREAMS were added as part of the X/Open UNIX extension.
- Existing Issue 4 functions were clarified as a result of industry feedback.

1.4.2 Changes from Issue 4, Version 2 to Issue 5

The following list summarizes the major changes that were made in this volume of IEEE Std. 1003.1-200x from Issue 4, Version 2 to Issue 5:

- Functions previously defined in the ISO POSIX-2 standard C-language Binding, Shared Memory, Enhanced Internationalization, and X/Open UNIX Extension Feature Groups were moved to the BASE.
- Threads were added to the BASE for alignment with the POSIX Threads Extension.
- The Realtime Threads Feature Group was added.
- The Realtime Feature Group was added for alignment with the POSIX Realtime Extension.
- Multibyte Support Extensions (MSE) were added to the BASE for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).
- Large File Summit (LFS) Extensions were added to the BASE for support of 64-bit or larger files and file systems.
- X/Open-specific threads extensions were added to the BASE.
- X/Open-specific dynamic linking functions were added to the BASE.
- A new category Legacy was added.
- The categories TO BE WITHDRAWN and WITHDRAWN were removed.

1.4.3 Changes from Issue 5 to Issue 6 (IEEE Std. 1003.1-200x)

The following list summarizes the major changes that were made in this volume of IEEE Std. 1003.1-200x from Issue 5 to Issue 6:

- This volume of IEEE Std. 1003.1-200x is extensively revised so it can be both an IEEE POSIX Standard and an Open Group Technical Standard.
- The POSIX System Interfaces requirements incorporate support of FIPS 151-2.
- The POSIX System Interfaces requirements are updated to align with some features of the Single UNIX Specification.
- A RATIONALE section is added to each reference page.

1.5 New Features

1.5.1 New Features in Issue 4, Version 2

The functions, headers, and external variables first introduced in Issue 4, Version 2 are listed in the table below.

New Functions, Headers, and External Variables in Issue 4, Version 2				
FD_CLR( )	*endutxent*( )	*gettimeofday*( )	*ptsname*( )	*sigaltstack*( )
FD_ISSET( )	*expm1*( )	*getutxent*( )	*putmsg*( )	*sighold*( )
FD_SET( )	*fattach*( )	*getutxid*( )	*putpmsg*( )	*sigignore*( )
FD_ZERO( )	*fchdir*( )	*getutxline*( )	*pututxline*( )	*siginterrupt*( )
_longjmp( )	*fchmod*( )	*getwd*( )	*random*( )	*sigpause*( )
_setjmp( )	*fchown*( )	*grantpt*( )	*re_comp*( )	*sigrelse*( )
a64l( )	*fcvt*( )	*ilogb*( )	*re_exec*( )	*sigset*( )
acosh( )	*fdetach*( )	*index*( )	*readlink*( )	*sigstack*( )
asinh( )	*ffs*( )	*initstate*( )	*readv*( )	*srandom*( )
atanh( )	*fmtmsg*( )	*insque*( )	*realpath*( )	*statvfs*( )
basename( )	*fstatvfs*( )	*ioctl*( )	*regcmp*( )	*strcasecmp*( )
bcmp( )	*ftime*( )	*isastream*( )	*regex*( )	*strdup*( )
bcopy( )	*ftok*( )	*killpg*( )	*remainder*( )	*strncasecmp*( )
brk( )	*ftruncate*( )	*l64a*( )	*remque*( )	*swapcontext*( )
bsd_signal( )	*gcvt*( )	*lchown*( )	*rindex*( )	*symlink*( )
bzero( )	*getcontext*( )	*lockf*( )	*rint*( )	*sync*( )
cbrt( )	*getdate*( )	*log1p*( )	*sbrk*( )	*syslog*( )
closelog( )	*getdtablesize*( )	*logb*( )	*scalb*( )	*tcgetsid*( )
dbm_clearerr( )	*getgrent*( )	*lstat*( )	*select*( )	*truncate*( )
dbm_close( )	*gethostid*( )	*makecontext*( )	*setcontext*( )	*ttyslot*( )
dbm_delete( )	*getitimer*( )	*mknod*( )	*setgrent*( )	*ualarm*( )
dbm_error( )	*getmsg*( )	*mkstemp*( )	*setitimer*( )	*unlockpt*( )
dbm_fetch( )	*getpagesize*( )	*mktemp*( )	*setlogmask*( )	*usleep*( )
dbm_firstkey( )	*getpgid*( )	*mmap*( )	*setpgrp*( )	*utimes*( )
dbm_nextkey( )	*getpmsg*( )	*mprotect*( )	*setpriority*( )	*valloc*( )
dbm_open( )	*getpriority*( )	*msync*( )	*setpwent*( )	*vfork*( )
dbm_store( )	*getpwent*( )	*munmap*( )	*setregid*( )	*wait3*( )
dirname( )	*getrlimit*( )	*nextafter*( )	*setreuid*( )	*waitid*( )
ecvt( )	*getrusage*( )	*nftw*( )	*setrlimit*( )	*writev*( )
endgrent( )	*getsid*( )	*openlog*( )	*setstate*( )	
endpwent( )	*getsubopt*( )	*poll*( )	*setutxent*( )	
<fmtmsg.h>	**<re_comp.h>**	**<sys/resource.h>**	**<sys/uio.h>**	**<utmpx.h>**
<libgen.h>	**<strings.h>**	**<sys/statvfs.h>**	**<sys/un.h>**	
<ndbm.h>	**<stropts.h>**	**<sys/time.h>**	**<syslog.h>**	
<poll.h>	**<sys/mman.h>**	**<sys/timeb.h>**	**<ucontext.h>**	
getdate_err	*__loc1*			

1.5.2 New Features in Issue 5

The functions and headers first introduced in Issue 5 are listed in the table below.

New Functions and Headers in Issue 5		
aio_cancel( )	*pthread_attr_getstacksize*( )	*pthread_self*( )
aio_error( )	*pthread_attr_init*( )	*pthread_setcancelstate*( )
aio_fsync( )	*pthread_attr_setdetachstate*( )	*pthread_setcanceltype*( )
aio_read( )	*pthread_attr_setguardsize*( )	*pthread_setconcurrency*( )
aio_return( )	*pthread_attr_setinheritsched*( )	*pthread_setschedparam*( )
aio_suspend( )	*pthread_attr_setschedparam*( )	*pthread_setspecific*( )
aio_write( )	*pthread_attr_setschedpolicy*( )	*pthread_sigmask*( )
asctime_r( )	*pthread_attr_setscope*( )	*pthread_testcancel*( )
btowc( )	*pthread_attr_setstackaddr*( )	*putc_unlocked*( )
clock_getres( )	*pthread_attr_setstacksize*( )	*putchar_unlocked*( )
clock_gettime( )	*pthread_cancel*( )	*pwrite*( )
clock_settime( )	*pthread_cleanup_pop*( )	*rand_r*( )
ctime_r( )	*pthread_cleanup_push*( )	*readdir_r*( )
dlclose( )	*pthread_cond_broadcast*( )	*sched_get_priority_max*( )
dlerror( )	*pthread_cond_destroy*( )	*sched_get_priority_min*( )
dlopen( )	*pthread_cond_init*( )	*sched_getparam*( )
dlsym( )	*pthread_cond_signal*( )	*sched_getscheduler*( )
fdatasync( )	*pthread_cond_timedwait*( )	*sched_rr_get_interval*( )
flockfile( )	*pthread_cond_wait*( )	*sched_setparam*( )
fseeko( )	*pthread_condattr_destroy*( )	*sched_setscheduler*( )
ftello( )	*pthread_condattr_getpshared*( )	*sched_yield*( )
ftrylockfile( )	*pthread_condattr_init*( )	*sem_close*( )
funlockfile( )	*pthread_condattr_setpshared*( )	*sem_destroy*( )
fwide( )	*pthread_create*( )	*sem_getvalue*( )
fwprintf( )	*pthread_detach*( )	*sem_init*( )
fwscanf( )	*pthread_equal*( )	*sem_open*( )
getc_unlocked( )	*pthread_exit*( )	*sem_post*( )
getchar_unlocked( )	*pthread_getconcurrency*( )	*sem_trywait*( )
getgrgid_r( )	*pthread_getschedparam*( )	*sem_unlink*( )
getgrnam_r( )	*pthread_getspecific*( )	*sem_wait*( )
getlogin_r( )	*pthread_join*( )	*shm_open*( )
getpwnam_r( )	*pthread_key_create*( )	*shm_unlink*( )
getpwuid_r( )	*pthread_key_delete*( )	*sigqueue*( )
gmtime_r( )	*pthread_kill*( )	*sigtimedwait*( )
lio_listio( )	*pthread_mutex_destroy*( )	*sigwait*( )
localtime_r( )	*pthread_mutex_getprioceiling*( )	*sigwaitinfo*( )
mbrlen( )	*pthread_mutex_init*( )	*snprintf*( )
mbrtowc( )	*pthread_mutex_lock*( )	*strtok_r*( )
mbsinit( )	*pthread_mutex_setprioceiling*( )	*swprintf*( )
mbsrtowcs( )	*pthread_mutex_trylock*( )	*swscanf*( )
mlock( )	*pthread_mutex_unlock*( )	*timer_create*( )
mlockall( )	*pthread_mutexattr_destroy*( )	*timer_delete*( )
mq_close( )	*pthread_mutexattr_getprioceiling*( )	*timer_getoverrun*( )
mq_getattr( )	*pthread_mutexattr_getprotocol*( )	*timer_gettime*( )
mq_notify( )	*pthread_mutexattr_getpshared*( )	*timer_settime*( )
mq_open( )	*pthread_mutexattr_gettype*( )	*towctrans*( )

New Functions and Headers in Issue 5		
mq_receive( )	*pthread_mutexattr_init*( )	*ttyname_r*( )
mq_send( )	*pthread_mutexattr_setprioceiling*( )	*vfwprintf*( )
mq_setattr( )	*pthread_mutexattr_setprotocol*( )	*vsnprintf*( )
mq_unlink( )	*pthread_mutexattr_setpshared*( )	*vswprintf*( )
munlock( )	*pthread_mutexattr_settype*( )	*vwprintf*( )
munlockall( )	*pthread_once*( )	*wcrtomb*( )
nanosleep( )	*pthread_rwlock_destroy*( )	*wcsrtombs*( )
pread( )	*pthread_rwlock_init*( )	*wcsstr*( )
pthread_atfork( )	*pthread_rwlock_rdlock*( )	*wctob*( )
pthread_attr_destroy( )	*pthread_rwlock_tryrdlock*( )	*wctrans*( )
pthread_attr_getdetachstate( )	*pthread_rwlock_trywrlock*( )	*wmemchr*( )
pthread_attr_getguardsize( )	*pthread_rwlock_unlock*( )	*wmemcmp*( )
pthread_attr_getinheritsched( )	*pthread_rwlock_wrlock*( )	*wmemcpy*( )
pthread_attr_getschedparam( )	*pthread_rwlockattr_destroy*( )	*wmemmove*( )
pthread_attr_getschedpolicy( )	*pthread_rwlockattr_getpshared*( )	*wmemset*( )
pthread_attr_getscope( )	*pthread_rwlockattr_init*( )	*wprintf*( )
pthread_attr_getstackaddr( )	*pthread_rwlockattr_setpshared*( )	*wscanf*( )
<aio.h>	**<iso646.h>**	**<sched.h>**
<dlfcn.h>	**<mqueue.h>**	**<semaphore.h>**
<inttypes.h>	**<pthread.h>**	**<wctype.h>**

1.5.3 New Features in Issue 6

Notes to Reviewers

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

A table listing new functions, headers, etc. since the ISO POSIX-1:1996 standard will be added here in a future draft.

1.6 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.

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.

1.7 Definitions

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

1.8 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. 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.

1.9 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 1.9.1). For maximum portability, an application should avoid such functionality.

1.9.1 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.

AIO Asynchronous Input and Output
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

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

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

Where applicable, functions are marked with the BAR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the BAR 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
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

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

margin legend.

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

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

CX Extension to the ISO C standard
The functionality described is an extension to the ISO C standard. Application writers may make use of an extension as it is supported on all IEEE Std. 1003.1-200x-conforming systems.

FD FORTRAN Development Utilities
The functionality described is optional.

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

FR FORTRAN Runtime Utilities
The functionality described is optional.

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

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

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

IP6 IPV6
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the IP6 margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the IP6 margin legend.

MAN Mandatory in the Next Draft
This is an interim draft code used to aid reviewers during the development of IEEE Std. 1003.1-200x. It denotes a feature that was previously an option or extension that is being brought into the mandatory base functionality. This margin code will be removed from the final draft.

MF Memory Mapped Files
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

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

ML Process Memory Locking
The functionality described is optional. The functionality described is also an extension to the

ISO C standard.

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

MLR Range Memory Locking

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

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

MON Monotonic Clock

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

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

MPR Memory Protection

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

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

MSG Message Passing

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

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

OB Obsolescent

The functionality described may be withdrawn in a future version of this volume of IEEE Std. 1003.1-200x. Strictly Conforming POSIX Applications and Strictly Conforming XSI Applications shall not use obsolescent features.

OF Output Format Incompletely Specified

The functionality described is an XSI extension. The format of the output produced by the utility is not fully specified. It is therefore not possible to post-process this output in a consistent fashion. Typical problems include unknown length of strings and unspecified field delimiters.

OH Optional Header

In the SYNOPSIS section of some interfaces in the System Interfaces volume of IEEE Std. 1003.1-200x an included header is marked as in the following example:

OH
```
#include <sys/types.h>
#include <grp.h>
struct group *getgrnam(const char *name);
```

This indicates that the marked header is not required on XSI-conformant systems.

PIO Prioritized Input and Output
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

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

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

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

RTS Realtime Signals Extension
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

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

SD Software Development Utilities
The functionality described is optional.

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

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

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

SHM Shared Memory Objects
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

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

SIO Synchronized Input and Output
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

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

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

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

SPN Spawn

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

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

SS Process Sporadic Server

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

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

TCT Thread CPU-Time Clocks

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

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

THR Threads

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

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

TMO Timeouts

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

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

TMR Timers

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

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

TPI Threads Priority Inheritance

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

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

TPP Thread Priority Protection

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

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

TPS Thread Execution Scheduling

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

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

TRC Trace

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

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

TEF Trace Event Filter

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

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

TRL Trace Log

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

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

TRI Trace Inherit

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

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

TSA Thread Stack Address Attribute

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

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

TSF Thread-Safe Functions

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

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

TSH Thread Process-Shared Synchronization

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

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

TSP Thread Sporadic Server

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

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

TSS Thread Stack Address Size

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

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

TYM Typed Memory Objects

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

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

UN Possibly Unsupportable Feature

The functionality described is an XSI extension. It need not be possible to implement the required functionality (as defined) on all conformant systems and the functionality need not be present. This may, for example, be the case where the conformant system is hosted and the underlying system provides the service in an alternative way.

UP User Portability Utilities

The functionality described is optional.

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

XSI Extension

The functionality described is an XSI extension. Functionality marked XSI is also an extension to the ISO C standard. Application writers may confidently make use of an extension on all systems supporting the X/Open System Interfaces Extension.

If an entire SYNOPSIS section is shaded and marked with one XSI, all the functionality described in that reference page is an extension. See the Base Definitions volume of IEEE Std. 1003.1-200x, Section 3.441, XSI.

XSR XSI STREAMS

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

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

1.10 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 values returned in *errno* if an error occurs.

''No errors are defined'' means that values and usage of *errno*, if any, depend on the implementation.

EXAMPLES

This section is non-normative.

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 non-normative.

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 non-normative.

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.

FUTURE DIRECTIONS

This section is non-normative.

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

601 **SEE ALSO**

602 This section is non-normative.

603 This section gives references to related information.

604 **CHANGE HISTORY**

605 This section is non-normative.

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

Chapter 2

General Information

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 library 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 library 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 library 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 library function can be declared without reference to any type defined in a header, it is also permissible to declare the function, either explicitly or implicitly, 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.

2.2 The Compilation Environment

2.2.1 POSIX.1 Symbols

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

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

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

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

2.2.1.1 The _POSIX_C_SOURCE Feature Test Macro

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

When an application includes a header described by this volume of IEEE Std. 1003.1-200x, and when this feature test macro is defined to have at least the value 200xMML:

1. All symbols required by this volume of IEEE Std. 1003.1-200x to appear when the header is included shall be made visible.
2. Symbols that are explicitly permitted, but not required, by this volume of IEEE Std. 1003.1-200x to appear in that header (including those in reserved name spaces) may be made visible.
3. Additional symbols not required or explicitly permitted by this volume of IEEE Std. 1003.1-200x to be in that header shall not be made visible, except when enabled by another feature test macro or by having defined _POSIX_C_SOURCE with a value larger than 200xxxL.

Identifiers in this volume of IEEE Std. 1003.1-200x may only be undefined using the **#undef** directive as described in Section 2.1 (on page 511) or Section 2.2.2 (on page 513). These **#undef** directives shall follow all **#include** directives of any header in this volume of IEEE Std. 1003.1-200x.

2.2.1.2 The _XOPEN_SOURCE Feature Test Macro

XSI An XSI-conforming application should ensure that the feature test macro _XOPEN_SOURCE is defined with the value 600 before inclusion of any header. This is needed to enable the functionality described in Section 2.2.1.1 and in addition to enable the X/Open System Interfaces Extension.

Since this volume of IEEE Std. 1003.1-200x is aligned with the ISO C standard, and since all functionality enabled by _POSIX_C_SOURCE set greater than zero and less than or equal to 200xxxL should be enabled by _XOPEN_SOURCE set equal to 600, there should be no need to

define either _POSIX_SOURCE or _POSIX_C_SOURCE if _XOPEN_SOURCE is so defined. Therefore, if _XOPEN_SOURCE is set equal to 600 and _POSIX_SOURCE is defined, or _POSIX_C_SOURCE is set greater than zero and less than or equal to 200xxxL, the behavior is the same as if only _XOPEN_SOURCE is defined and set equal to 600. However, should _POSIX_C_SOURCE be set to a value greater than 200xxxL, the behavior is undefined.

2.2.2 The Name Space

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

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

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

The prefixes posix_, POSIX_, and _POSIX_ are reserved for use by IEEE Std. 1003.1-200x and other POSIX standards. Implementations may add symbols to the headers shown in the following table, provided the identifiers for those symbols begin with the corresponding reserved prefixes in the following table, and do not use the reserved prefixes posix_, POSIX_, or _POSIX_.

	Header	Prefix	Suffix	Complete Name
AIO	**<aio.h>**	aio_, lio_, AIO_, LIO_		
	<arpa/inet.h>	in_, inet_		
	<ctype.h>	to[a-z], is[a-z]		
	<dirent.h>	d_		
	<errno.h>	E[0-9], E[A-Z]		
	<fcntl.h>	l_		
	<glob.h>	gl_		
	<grp.h>	gr_		
	<inttypes.h>	int[0-9a-z_]_t, uint[0-9a-z_]_t		
	<limits.h>		_MAX, _MIN	
	<locale.h>	LC_[A-Z]		
MSG	**<mqueue.h>**	mq_, MQ_		
XSI	**<ndbm.h>**	dbm_		
	<netdb.h>	h_, n_, p_, s_		
	<net/if.h>	if_		
	<netinet/in.h>	in_, ip_, s_, sin_		
XSI	**<poll.h>**	pd_, ph_, ps_		
	<pthread.h>	pthread_, PTHREAD_		
	<pwd.h>	pw_		
	<regex.h>	re_, rm_		
PS	**<sched.h>**	sched_, SCHED_		
SEM	**<semaphore.h>**	sem_, SEM_		
	<signal.h>	sa_, uc_, SIG[A-Z], SIG_[A-Z]		
XSI		ss_, sv_		
RTS		si_, SI_, sigev_, SIGEV_, sival_		
XSI	**<stropts.h>**	bi_, ic_, l_, sl_, str_		
	<stdint.h>	int[0-9a-z_]_t, uint[0-9a-z_]_t		
	<stdlib.h>	str[a-z]		
	<string.h>	str[a-z], mem[a-z], wcs[a-z]		
XSI	**<sys/ipc.h>**	ipc_		key, pad, seq
MF	**<sys/mman.h>**	shm_, MAP_, MCL_, MS_, PROT_		
XSI	**<sys/msg.h>**	msg		msg
XSI	**<sys/resource.h>**	rlim_, ru_		
XSI	**<sys/sem.h>**	sem		sem
XSI	**<sys/shm.h>**	shm		
	<sys/socket.h>	_ss, sa_, if_, ifc_, ifru_, infu_, ifra_, msg_, cmsg_, l_		
	<sys/stat.h>	st_		
XSI	**<sys/statvfs.h>**	f_		
	<sys/time.h>	fds_, it_, tv_, FD_		
	<sys/times.h>	tms_		
XSI	**<sys/uio.h>**	iov_		
	<sys/un.h>	sun_		
	<sys/utsname.h>	uts_		
XSI	**<sys/wait.h>**	si_, W[A-Z], P_		
	<termios.h>	c_		

	Header	Prefix	Suffix	Complete Name
	<time.h>	tm_		
TMR		clock_, timer_, it_, tv_,		
TMR		CLOCK_, TIMER_		
XSI	**<ucontext.h>**	uc_, ss_		
XSI	**<ulimit.h>**	UL_		
	<utime.h>	utim_		
XSI	**<utmpx.h>**	ut_	_LVL, _TIME, _PROCESS	
	<wchar.h>	wcs[a-z]		
	<wctype.h>	is[a-z], to[a-z]		
	<wordexp.h>	we_		
	ANY header	POSIX_, _POSIX_, posix_	_t	

Note: The notation [A–Z] indicates any uppercase letter in the portable character set. The notation [a–z] indicates any lowercase letter in the portable character set. Commas and spaces in the lists of prefixes and complete names in the above table are not part of any prefix or complete name.

If any header in the following table is included, macros with the prefixes shown may be defined. After the last inclusion of a given header, an application may use identifiers with the corresponding prefixes for its own purpose, provided their use is preceded by an **#undef** of the corresponding macro.

Header	Prefix
<dflcn.h>	RTLD_
<fcntl.h>	F_, O_, S_
<fmtmsg.h>	MM_
<fnmatch.h>	FNM_
<ftw.h>	FTW
<glob.h>	GLOB_
<inttypes.h>	PRI[a-z], SCN[a-z]
<ndbm.h>	DBM_
<net/if.h>	IF_
<netinet/in.h>	IMPLINK_, IN_, INADDR_, IP_, IPPORT_, IPPROTO_, SOCK_
<netinet/tcp.h>	TCP_
<nl_types.h>	NL_
<poll.h>	POLL
<regex.h>	REG_
<signal.h>	SA_, SIG_[0-9a-z_],
	BUS_, CLD_, FPE_, ILL_, POLL_, SEGV_, SI_, SS_, SV_, TRAP_
stdint.h	INT[0-9A-Z_]_MIN, INT[0-9A-Z_]_MAX, INT[0-9A-Z_]_C UINT[0-9A-Z_]_MIN, UINT[0-9A-Z_]_MAX, UINT[0-9A-Z_]_C
<stropts.h>	FLUSH[A-Z], I_, M_, MUXID_R[A-Z], S_, SND[A-Z], STR
<syslog.h>	LOG_
<sys/ipc.h>	IPC_
<sys/mman.h>	PROT_, MAP_, MS_
<sys/msg.h>	MSG[A-Z]
<sys/resource.h>	PRIO_, RLIM_, RLIMIT_, RUSAGE_
<sys/sem.h>	SEM_
<sys/shm.h>	SHM[A-Z], SHM_[A-Z]
<sys/socket.h>	AF_, CMSG_, MSG_, PF_, SCM_, SHUT_, SO
<sys/stat.h>	S_
<sys/statvfs.h>	ST_
<sys/time.h>	FD_, ITIMER_
<sys/uio.h>	IOV_
<sys/wait.h>	BUS_, CLD_, FPE_, ILL_, POLL_, SEGV_, SI_, TRAP_
<termios.h>	V, I, O, TC, B[0-9] (See below.)
<wordexp.h>	WRDE_

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

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

CBAUD	EXTB	VDSUSP
DEFECHO	FLUSHO	VLNEXT
ECHOCTL	LOBLK	VREPRINT
ECHOKE	PENDIN	VSTATUS
ECHOPRT	SWTCH	VWERASE
EXTA	VDISCARD	

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

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

_Exit	cbrtf	exit	fsetpos	mbrtowc	sinhl	
abort	cbrtl	exp	ftell	mbsinit	sinl	
abs	ccos	exp2	fwide	mbsrtowcs	sprintf	
acos	ccosf	exp2f	fwprintf	mbstowcs	sqrt	
acosf	ccosh	exp2l	fwrite	mbtowc	sqrtf	
acosh	ccoshf	expf	fwscanf	mem[a-z]*	sqrtl	
acoshf	ccoshl	expl	getc	mktime	srand	
acoshl	ccosl	expm1	getchar	modf	sscanf	
acosl	ceil	expm1f	getenv	modff	str[a-z]*	
acosl	ceilf	expm1l	gets	modfl	strtof	
asctime	ceilf	fabs	getwc	nan	strtoimax	
asin	ceill	fabsf	getwchar	nanf	strtold	
asinf	ceill	fabsl	gmtime	nanl	strtoll	
asinh	cexp	fclose	hypotf	nearbyint	strtoull	
asinhf	cexpf	fdim	hypotl	nearbyintf	strtoumax	
asinhl	cexpl	fdimf	ilogb	nearbyintl	swprintf	
asinl	cimag	fdiml	ilogbf	nextafterf	swscanf	
asinl	cimagf	feclearexcept	ilogbl	nextafterl	system	
atan	cimagl	fegetenv	imaxabs	nexttoward	tan	
atan2	clearerr	fegetexceptflag	imaxdiv	nexttowardf	tanf	
atan2f	clock	fegetround	is[a-z]*	nexttowardl	tanh	
atan2l	clog	feholdexcept	isblank	perror	tanhf	
atanf	clogf	feof	iswblank	pow	tanhl	
atanf	clogl	feraiseexcept	labs	powf	tanl	
atanh	conj	ferror	ldexp	powl	tgamma	
atanh	conjf	fesetenv	ldexpf	printf	tgammaf	
atanhf	conjl	fesetexceptflag	ldexpl	putc	tgammal	
atanhl	copysign	fesetround	ldiv	putchar	time	
atanl	copysignf	fetestexcept	ldiv	puts	tmpfile	
atanl	copysignl	feupdateenv	lgammaf	putwc	tmpnam	
atexit	cos	fflush	lgammal	putwchar	to[a-z]*	
atof	cosf	fgetc	llabs	qsort	trunc	
atoi	cosh	fgetpos	lldiv	raise	truncf	
atol	coshf	fgets	llrint	rand	truncl	
atoll	coshl	fgetwc	llrintf	realloc	ungetc	
bsearch	cosl	fgetws	llrintl	remainderf	ungetwc	
cabs	cpow	floor	llround	remainderl	va_end	
cabsf	cpowf	floorf	llroundf	remove	vfprintf	
cabsl	cpowl	floorl	llroundl	remquo	vfscanf	
cacos	cproj	fma	localeconv	remquof	vfwprintf	

cacosf	cprojf	fmaf	ldg	remquol	vfwscanf		
cacosh	cprojl	fmal	ldg10	rename	vprintf		
cacoshf	creal	fmax	ldg10f	rewind	v$scanf		
cacoshl	crealf	fmaxf	ldg10l	rint	v$printf		
cacosl	creall	fmaxl	ldg1p	rintf	v$sscanf		
calloc	csin	fmin	ldg1pf	rintl	v$wprintf		
carg	csinf	fminf	ldg1pl	round	v$wscanf		
cargf	csinh	fminl	ldg2	roundf	vwprintf		
cargl	csinhf	fmod	ldg2f	roundl	vwscanf		
casin	csinhl	fmodf	ldg2l	scalbln	wcrtomb		
casinf	csinl	fmodl	ldgb	scalblnf	wcs[a-z]*		
casinh	csqrt	fopen	ldgbf	scalblnl	wcstof		
casinhf	csqrtf	fprintf	ldgbl	scalbn	wcstoimax		
casinhl	csqrtl	fputc	ldgf	scalbnf	wcstold		
casinl	ctan	fputs	ldgl	scalbnl	wcstoll		
catan	ctanf	fputwc	ldngjmp	scanf	wcstoull		
catanf	ctanl	fputws	lrint	setbuf	wcstoumax		
catanh	ctime	fread	lrintf	setjmp	wctob		
catanh	difftime	free	lrintl	setlocale	wctomb		
catanhf	div	freopen	lround	setvbuf	wctrans		
catanhf	erfcf	frexp	lroundf	signal	wctype		
catanhl	erfcl	frexpf	lroundl	sin	wcwidth		
catanhl	erff	frexpl	malloc	sinf	wmem[a-z]*		
catanl	erfl	fscanf	mblen	sinh	wprintf		
cbrt	errno	fseek	mbrlen	sinhf	wscanf		

Note: The notation [a–z] indicates any lowercase letter in the portable character set. The notation '*' indicates any combination of digits, letters in the portable character set, and underscore.

Notes to Reviewers

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

The following table should be made complete by including everything not in the previous table.

4. The following identifiers are also reserved for use as identifiers with external linkage:

Table 2-1 XSI Identifiers

a64l	fattach	getpriority	mknod	regex	sigelse	
basename	fchdir	getpwent	mkstemp	remainder	sigset	
bcmp	fchmod	getrlimit	mktemp	remque	sigstack	
bcopy	fchown	getrusage	mmap	rindex	srandom	
brk	fcvt	getsid	mprotect	sbrk	statvfs	
bsd_signal	fdetach	getsubopt	mrand48	scalb	strcasecmp	
bzero	ffs	gettimeofday	msync	select	strdup	
cbrt	fmtmsg	getutxent	munmap	setcontext	strncasecmp	
closelog	fstatvfs	getutxid	nextafter	setgrent	swapcontext	
dbm_clearerr	ftime	getutxline	nftw	setitimer	symlink	
dbm_close	ftok	getwd	nice	_setjmp	sync	
dbm_delete	ftruncate	grantpt	openlog	setlogmask	syslog	
dbm_error	gcvt	index	poll	setpgrp	tcgetsid	
dbm_fetch	getcontext	initstate	ptsname	setpriority	truncate	
dbm_firstkey	getdate	insque	putmsg	setpwent	ttyslot	
dbm_nextkey	getdtablesize	ioctl	putpmsg	setreuid	ualarm	
dbm_open	getgrent	isastream	pututxline	setrlimit	unlockpt	
dbm_store	getgrgid	killpg	random	setstate	usleep	
dirname	gethostid	l64a	readlink	setutxent	utimes	
ecvt	getitimer	lchown	readv	sigaltstack	valloc	
endgrent	getmsg	lockf	realpath	sighold	vfork	
endpwent	getpagesize	_longjmp	re_comp	sigignore	wait3	
endservent	getpgid	lstat	re_exec	siginterrupt	waitid	
endutxent	getpmsg	makecontext	regcmp	sigpause	writev	

Table 2-2 Sockets Identifiers

accept	if_freenameindex	recvfrom	shutdown	
bind	if_indextoname	recvmsg	socket	
connect	if_nameindex	send	socketpair	
getpeername	if_nametoindex	sendmsg		
getsockname	listen	sendto		
getsockopt	recv	setsockopt		

Table 2-3 IP Address Resolution Identifiers

endhostent	getipnodebyaddr	getservbyname	inet_netof	
endnetent	getipnodebyname	getservbyport	inet_network	
endprotoent	getnameinfo	getservent	inet_ntoa	
endservent	getnetbyaddr	h_errno	ntohl	
getaddrinfo	getnetbyname	htonl	ntohs	
gethostbyaddr	getnetent	htons	sethostent	
gethostbyname	getprotobyname	inet_addr	setnetent	
gethostent	getprotobynumber	inet_lnaof	setprotoent	
gethostname	getprotoent	inet_makeaddr	setservent	

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

No other identifiers are reserved.

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

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

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

2.3 Error Numbers

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

Some functions provide the error number in a variable accessed through the symbol *errno*. The symbol *errno*, defined by including the **<errno.h>** header, is a macro that expands to a modifiable **lvalue** of type **int**.

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

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

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

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

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

For functions under the Threads option for which [EINTR] is not listed as a possible error condition in this volume of IEEE Std. 1003.1-200x, an implementation shall not return an error code of [EINTR].

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

[E2BIG]

Argument list too long. The sum of 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.

or:

Lack of space in an output buffer.

or:

Argument is greater than the system-imposed maximum.

[EACCES]
Permission denied. An attempt was made to access a file in a way forbidden by its file access permissions.

[EADDRINUSE]
Address in use. The specified address is in use.

[EADDRNOTAVAIL]
Address not available. The specified address is not available from the local system.

[EAFNOSUPPORT]
Address family not supported. The implementation does not support the specified address family, or the specified address is not a valid address for the address family of the specified socket.

[EAGAIN]
Resource temporarily unavailable. This is a temporary condition and later calls to the same routine may complete normally.

[EALREADY]
Connection already in progress. A connection request is already in progress for the specified socket.

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

[EBADMSG]
XSR Bad message. During a *read*( ), *getmsg*( ), or *ioctl*( ) I_RECVFD request to a STREAMS device, a message arrived at the head of the STREAM that is inappropriate for the function receiving the message.

read( )	Message waiting to be read on a STREAM is not a data message.
getmsg( )	A file descriptor was received instead of a control message.
ioctl( )	Control or data information was received instead of a file descriptor when I_RECVFD was specified.

or:

Bad Message. The implementation has detected a corrupted message.

[EBUSY]
Resource busy. An attempt was made to make use of a system resource that is not currently available, as it is being used by another process in a manner that would have conflicted with the request being made by this process.

[ECANCELED]
Operation canceled. The associated asynchronous operation was canceled before completion.

[ECHILD]
No child process. A *wait*( ) or *waitpid*( ) function was executed by a process that had no existing or unwaited-for child process.

[ECONNABORTED]
Connection aborted. The connection has been aborted.

[ECONNREFUSED]
Connection refused. An attempt to connect to a socket was refused because there was no

process listening or because the queue of connection requests was full and the underlying protocol does not support retransmissions.

[ECONNRESET]
Connection reset. The connection was forcibly closed by the peer.

[EDEADLK]
Resource deadlock would occur. An attempt was made to lock a system resource that would have resulted in a deadlock situation.

[EDESTADDRREQ]
Destination address required. No bind address was established.

[EDOM]
Domain error. An input argument is outside the defined domain of the mathematical function (defined in the ISO C standard).

[EDQUOT]
Reserved.

[EEXIST]
File exists. An existing file was mentioned in an inappropriate context; for example, as a new link name in the *link*( ) function.

[EFAULT]
Bad address. The system detected an invalid address in attempting to use an argument of a call. The reliable detection of this error cannot be guaranteed, and when not detected may result in the generation of a signal, indicating an address violation, which is sent to the process.

[EFBIG]
File too large. The size of a file would exceed the maximum file size of an implementation or offset maximum established in the corresponding file description.

[EHOSTUNREACH]
Host is unreachable. The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).

[EIDRM]
Identifier removed. Returned during XSI interprocess communication if an identifier has been removed from the system.

[EILSEQ]
Illegal byte sequence. A wide-character code has been detected that does not correspond to a valid character, or a byte sequence does not form a valid wide-character code (defined in the ISO C standard).

[EINPROGRESS]
Operation in progress. This code is used to indicate that an asynchronous operation has not yet completed.

or:

O_NONBLOCK is set for the socket file descriptor and the connection cannot be immediately established.

[EINTR]
Interrupted function call. An asynchronous signal was caught by the process during the execution of an interruptible function. If the signal handler performs a normal return, the interrupted function call may return this condition (see the Base Definitions volume of

IEEE Std. 1003.1-200x, **<signal.h>**).

[EINVAL]
Invalid argument. Some invalid argument was supplied; for example, specifying an undefined signal in a *signal*( ) function or a *kill*( ) function.

[EIO]
Input/output error. Some physical input or output error has occurred. This error may be reported on a subsequent operation on the same file descriptor. Any other error-causing operation on the same file descriptor may cause the [EIO] error indication to be lost.

[EISCONN]
Socket is connected. The specified socket is already connected.

[EISDIR]
Is a directory. An attempt was made to open a directory with write mode specified.

[ELOOP]
Symbolic link loop. A loop exists in symbolic links encountered during path name resolution. This error may also be returned if more than {SYMLOOP_MAX} symbolic links are encountered during path name resolution.

[EMFILE]
Too many open files. An attempt was made to open more than the maximum number of {OPEN_MAX} file descriptors allowed in this process.

[EMLINK]
Too many links. An attempt was made to have the link count of a single file exceed {LINK_MAX}.

[EMSGSIZE]
Message too large. A message sent on a transport provider was larger than an internal message buffer or some other network limit.

or:

Inappropriate message buffer length.

[EMULTIHOP]
Reserved.

[ENAMETOOLONG]
File name too long. The length of a path name exceeds {PATH_MAX}, or a path name component is longer than {NAME_MAX}. This error may also occur when path name substitution, as a result of encountering a symbolic link during path name resolution, results in a path name string the size of which exceeds {PATH_MAX}.

[ENETDOWN]
Network is down. The local network interface used to reach the destination is down.

[ENETRESET]
The connection was aborted by the network.

[ENETUNREACH]
Network unreachable. No route to the network is present.

[ENFILE]
Too many files open in system. Too many files are currently open in the system. The system has reached its predefined limit for simultaneously open files and temporarily cannot accept requests to open another one.

[ENOBUFS]
No buffer space available. Insufficient buffer resources were available in the system to perform the socket operation.

XSR [ENODATA]
No message available. No message is available on the STREAM head read queue.

[ENODEV]
No such device. An attempt was made to apply an inappropriate function to a device; for example, trying to read a write-only device such as a printer.

[ENOENT]
No such file or directory. A component of a specified path name does not exist, or the path name is an empty string.

[ENOEXEC]
Executable file format error. A request is made to execute a file that, although it has the appropriate permissions, is not in the format required by the implementation for executable files.

[ENOLCK]
No locks available. A system-imposed limit on the number of simultaneous file and record locks has been reached and no more are currently available.

[ENOLINK]
Reserved.

[ENOMEM]
Not enough space. The new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.

[ENOMSG]
No message of the desired type. The message queue does not contain a message of the required type during XSI interprocess communication.

[ENOPROTOOPT]
Protocol not available. The protocol option specified to *setsockopt*( ) is not supported by the implementation.

[ENOSPC]
No space left on a device. During the *write*( ) function on a regular file or when extending a directory, there is no free space left on the device.

XSR [ENOSR]
No STREAM resources. Insufficient STREAMS memory resources are available to perform a STREAMS-related function. This is a temporary condition; it may be recovered from if other processes release resources.

XSR [ENOSTR]
Not a STREAM. A STREAM function was attempted on a file descriptor that was not associated with a STREAMS device.

[ENOSYS]
Function not implemented. An attempt was made to use a function that is not available in this implementation.

[ENOTCONN]
Socket not connected. The socket is not connected.

[ENOTDIR]
Not a directory. A component of the specified path name exists, but it is not a directory, when a directory was expected.

[ENOTEMPTY]
Directory not empty. A directory other than an empty directory was supplied when an empty directory was expected.

[ENOTSOCK]
Not a socket. The file descriptor does not refer to a socket.

[ENOTSUP]
Not supported. The implementation does not support this feature of the Realtime Option Group.

[ENOTTY]
Inappropriate I/O control operation. A control function has been attempted for a file or special file for which the operation is inappropriate.

[ENXIO]
No such device or address. Input or output on a special file refers to a device that does not exist, or makes a request beyond the capabilities of the device. It may also occur when, for example, a tape drive is not on-line.

[EOPNOTSUPP]
Operation not supported on socket. The type of socket (address family or protocol) does not support the requested operation.

[EOVERFLOW]
Value too large to be stored in data type. The user ID or group ID of an IPC or file system object was too large to be stored into the appropriate member of the caller-provided structure. This error shall only occur on implementations that support a larger range of user ID or group ID values than the declared structure member can support. This usually occurs because the IPC or file system object resides on a remote machine with a larger value of the type **uid_t**, **off_t**, or **gid_t** than the local system.

[EPERM]
Operation not permitted. An attempt was made to perform an operation limited to processes with appropriate privileges or to the owner of a file or other resource.

[EPIPE]
Broken pipe. A write was attempted on a socket, pipe, or FIFO for which there is no process to read the data.

[EPROTO]
Protocol error. Some protocol error occurred. This error is device-specific, but is generally not related to a hardware failure.

[EPROTONOSUPPORT]
Protocol not supported. The protocol is not supported by the address family, or the protocol is not supported by the implementation.

[EPROTOTYPE]
Socket type not supported. The socket type is not supported by the protocol.

[ERANGE]
Result too large or too small. The result of the function is too large (overflow) or too small (underflow) to be represented in the available space (defined in the ISO C standard).

[EROFS]
Read-only file system. An attempt was made to modify a file or directory on a file system that is read-only.

[ESPIPE]
Invalid seek. An attempt was made to access the file offset associated with a pipe or FIFO.

[ESRCH]
No such process. No process can be found corresponding to that specified by the given process ID.

[ESTALE]
Reserved.

XSR [ETIME]
STREAM *ioctl*( ) timeout. The timer set for a STREAMS *ioctl*( ) call has expired. The cause of this error is device-specific and could indicate either a hardware or software failure, or a timeout value that is too short for the specific operation. The status of the *ioctl*( ) operation is indeterminate.

[ETIMEDOUT]
Connection timed out. The connection to a remote machine has timed out. If the connection timed out during execution of the function that reported this error (as opposed to timing out prior to the function being called), it is unspecified whether the function has completed some or all of the documented behavior associated with a successful completion of the function.

or:

Operation timed out. The time limit associated with the operation was exceeded before the operation completed.

[ETXTBSY]
Text file busy. An attempt was made to execute a pure-procedure program that is currently open for writing, or an attempt has been made to open for writing a pure-procedure program that is being executed.

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

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

[EXDEV]
Improper link. A link to a file on another file system was attempted.

2.3.1 Additional Error Numbers

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

2.4 Signal Concepts

2.4.1 Signal Generation and Delivery

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

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

Each process has an action to be taken in response to each signal defined by the system (see Section 2.4.3 (on page 530)). A signal is said to be *delivered* to a process when the appropriate action for the process and signal is taken. A signal is said to be *accepted* by a process when the signal is selected and returned by one of the *sigwait*( ) functions.

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

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

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

When any stop signal (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) is generated for a process, any pending SIGCONT signals for that process shall be discarded. Conversely, when SIGCONT is generated for a process, all pending stop signals for that process shall be discarded. When

SIGCONT is generated for a process that is stopped, the process shall be continued, even if the SIGCONT signal is blocked or ignored. If SIGCONT is blocked and not ignored, it shall remain pending until it is either unblocked or a stop signal is generated for the process.

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

2.4.2 Realtime Signal Generation and Delivery

RTS This section describes extensions to support realtime signal generation and delivery. This functionality is dependent on support of the Realtime Signals Extension option (and the rest of this section is not further shaded for this option).

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

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

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

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

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

SIGEV_THREAD A notification function shall be called to perform notification.

An implementation may define additional notification mechanisms.

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

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

Member Type	Member Name	Description
int	*sival_int*	Integer signal value.
void*	*sival_ptr*	Pointer signal value.

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

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

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

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

If, when a pending signal is delivered, there are additional signals queued to that signal number, the signal remains pending. Otherwise, the pending indication is reset.

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

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

2.4.3 Signal Actions

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

SIG_DFL Signal-specific default action.

The default actions for the signals defined in this volume of IEEE Std. 1003.1-200x

RTS are specified under **<signal.h>**. If the Realtime Signals Extension option is supported, the default actions for the realtime signals in the range SIGRTMIN to SIGRTMAX are 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

1401 the SIGTSTP, SIGTTIN, or SIGTTOU signals. In cases where delivery of one of
1402 these signals would stop such a process, the signal shall be discarded.

1403 Setting a signal action to SIG_DFL for a signal that is pending, and whose default
1404 action is to ignore the signal (for example, SIGCHLD), shall cause the pending
1405 signal to be discarded, whether or not it is blocked.

1406 The default action for SIGCONT is to resume execution at the point where the
1407 RTS process was stopped, after first handling any pending unblocked signals. If the
1408 Realtime Signals Extension option is supported, any queued values pending shall
1409 be discarded and the resources used to queue them shall be released and returned
1410 to the system for other use.

1411 SIG_IGN Ignore signal.

1412 Delivery of the signal shall have no effect on the process. The behavior of a process
1413 RTS is undefined after it ignores a SIGFPE, SIGILL, SIGSEGV, or SIGBUS signal that
1414 RTS was not generated by *kill*( ), *sigqueue*( ), or *raise*( ).

1415 The system shall not allow the action for the signals SIGKILL or SIGSTOP to be set
1416 to SIG_IGN.

1417 Setting a signal action to SIG_IGN for a signal that is pending shall cause the
1418 pending signal to be discarded, whether or not it is blocked.

1419 If a process sets the action for the SIGCHLD signal to SIG_IGN, the behavior is
1420 XSI unspecified, except as specified below.

1421 If the action for the SIGCHLD signal is set to SIG_IGN, child processes of the
1422 calling processes shall not be transformed into zombie processes when they
1423 terminate. If the calling process subsequently waits for its children, and the process
1424 has no unwaited-for children that were transformed into zombie processes, it shall
1425 block until all of its children terminate, and *wait*( ), *waitid*( ), and *waitpid*( ) shall fail
1426 and set *errno* to [ECHILD].

1427 RTS If the Realtime Signals Extension option is supported, any queued values pending
1428 shall be discarded and the resources used to queue them shall be released and
1429 made available to queue other signals.

1430 *pointer to a function*
1431 Catch signal.

1432 On delivery of the signal, the receiving process is to execute the signal-catching
1433 function at the specified address. After returning from the signal-catching function,
1434 the receiving process shall resume execution at the point at which it was
1435 interrupted.

1436 If the SA_SIGINFO flag for the signal is cleared, the signal-catching function shall
1437 be entered as a C-language function call as follows:

1438 `void` *`func`*`(int` *`signo`*`);`

1439 XSI | RTS If the SA_SIGINFO flag for the signal is set, the signal-catching function shall be
1440 entered as a C-language function call as follows:

1441 `void` *`func`*`(int` *`signo`*`, siginfo_t *`*`info`*`, void *`*`context`*`);`

1442 where *func* is the specified signal-catching function, *signo* is the signal number of
1443 the signal being delivered, and *info* is a pointer to a **siginfo_t** structure defined in
1444 **<signal.h>** containing at least the following members:

Member Type	Member Name	Description
int	*si_signo*	Signal number
int	*si_code*	Cause of the signal
union sigval	*si_value*	Signal value

The *si_signo* member contains the signal number. This is the same as the *signo* parameter. The *si_code* member contains a code identifying the cause of the signal. The following values are defined for *si_code*:

Notes to Reviewers

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

The shading in this area needs some work.

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

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

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

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

RTS SI_MESGQ The signal was generated by the arrival of a message on an empty message queue.

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

RTS If the Realtime Signals Extension is supported, and *si_code* is one of SI_QUEUE, SI_TIMER, SI_ASYNCIO, or SI_MESGQ, then *si_value* contains the application-specified signal value. Otherwise, the contents of *si_value* are undefined.

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

XSI

RTS

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

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

When signal-catching functions are invoked asynchronously with process execution, the behavior of some of the functions defined by this volume of IEEE Std. 1003.1-200x is unspecified if they are called from a signal-catching function.

The following table defines a set of functions that are either reentrant or not interruptible by signals and are async-signal-safe. Therefore applications may invoke them, without restriction, from signal-catching functions:

Notes to Reviewers

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

The contents of the following tables need to be reviewed.

Base functions:

_Exit( )	*fpathconf*( )	*pipe*( )	*stat*( )
_exit( )	*fstat*( )	*raise*( )	*symlink*( )
access( )	*fsync*( )	*read*( )	*sysconf*( )
alarm( )	*ftruncate*( )	*readlink*( )	*tcdrain*( )
cfgetispeed( )	*getegid*( )	*rename*( )	*tcflow*( )
cfgetospeed( )	*geteuid*( )	*rmdir*( )	*tcflush*( )
cfsetispeed( )	*getgid*( )	*setgid*( )	*tcgetattr*( )
cfsetospeed( )	*getgroups*( )	*setpgid*( )	*tcgetpgrp*( )
chdir( )	*getpgrp*( )	*setsid*( )	*tcsendbreak*( )
chmod( )	*getpid*( )	*setuid*( )	*tcsetattr*( )
chown( )	*getppid*( )	*sigaction*( )	*tcsetpgrp*( )
close( )	*getuid*( )	*sigaddset*( )	*time*( )
creat( )	*kill*( )	*sigdelset*( )	*times*( )
dup( )	*link*( )	*sigemptyset*( )	*umask*( )
dup2( )	*lseek*( )	*sigfillset*( )	*uname*( )
execle( )	*lstat*( )	*sigismember*( )	*unlink*( )
execve( )	*mkdir*( )	*signal*( )	*utime*( )
fchmod( )	*mkfifo*( )	*sigpending*( )	*wait*( )
fchown( )	*open*( )	*sigprocmask*( )	*waitpid*( )
fcntl( )	*pathconf*( )	*sigsuspend*( )	*write*( )
fork( )	*pause*( )	*sleep*( )	

Realtime functions:

aio_error( )	*clock_gettime*( )	*sigpause*( )	*timer_getoverrun*( )
aio_return( )	*fdatasync*( )	*sigqueue*( )	*timer_gettime*( )
aio_suspend( )	*sem_post*( )	*sigset*( )	*timer_settime*( )

Tracing functions:

posix_trace_event( )

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

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

2.4.4 Signal Effects on Other Functions

Signals affect the behavior of certain functions defined by this volume of IEEE Std. 1003.1-200x if delivered to a process while it is executing such a function. If the action of the signal is to terminate the process, the process shall be terminated and the function shall not return. If the action of the signal is to stop the process, the process shall stop until continued or terminated. Generation of a SIGCONT signal for the process shall cause the process to be continued, and the original function shall continue at the point the process was stopped. If the action of the signal is to invoke a signal-catching function, the signal-catching function shall be invoked; in this case the original function is said to be *interrupted* by the signal.

Notes to Reviewers

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

D3, XSH, ERN 20 points out a discrepancy between the following sentence and the paragraph above beginning "All functions not in the above ...". An interpretation will be filed.

If the signal-catching function executes a **return** statement, the behavior of the interrupted function shall be as described individually for that function. Signals that are ignored shall not affect the behavior of any function; signals that are blocked shall not affect the behavior of any function until they are unblocked and then delivered, except as specified for the *sigpending*( ) and *sigwait*( ) functions.

2.5 Standard I/O Streams

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

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

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

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

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

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

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. This functionality 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*( ) or *abort*( ). 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.
- If it is a stream which is line buffered, and the last byte written to the stream was a newline (that is, as if a:

  ```
  putc('\n')
  ```

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

Otherwise, the result is undefined.

For the second handle:

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

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

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

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

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

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

2.5.2 Stream Orientation and Encoding Rules

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

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

Byte input/output functions cannot be applied to a wide-oriented stream, and wide-character input/output functions cannot be applied to a byte-oriented stream. The remaining stream operations shall not affect and shall not be affected by a stream's orientation, except for the following additional restrictions:

- Binary wide-oriented streams have the file positioning restrictions ascribed to both text and binary streams.
- For wide-oriented streams, after a successful call to a file-positioning function that leaves the file position indicator prior to the end-of-file, a wide-character output function can overwrite a partial character; any file contents beyond the byte(s) written are henceforth undefined.

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

Implementations that support multiple encoding rules associate an encoding rule with the stream. The encoding rule shall be determined by the setting of the *LC_CTYPE* category in the current locale at the time when the stream becomes wide-oriented. If a wide-character input/output function is applied to a byte-oriented stream, the encoding rule used is undefined. As with the stream's orientation, the encoding rule associated with a stream cannot be changed once it has been set, except by a successful call to *freopen*( ) which clears the encoding rule and resets the orientation to unoriented.

Although both text and binary wide-oriented streams are conceptually sequences of wide characters, the external file associated with a wide-oriented stream is a sequence of (possibly multibyte) characters generalized as follows:

- Multibyte encodings within files may contain embedded null bytes (unlike multibyte encodings valid for use internal to the program).
- A file need not begin nor end in the initial shift state.

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

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

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

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

2.6 STREAMS

XSR STREAMS functionality is provided on implementations supporting the XSI STREAMS Option Group. This functionality 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. 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. Messages flow in both directions in a STREAM. A given module need not understand and process every message in the STREAM, but every module in the STREAM handles every message. Each module accepts messages from one of its neighbor modules in the STREAM, and passes them to the other neighbor. For example, a line discipline module may transform the data. Data flow through the intermediate modules is bidirectional, with all modules handling, and optionally processing, all messages. There are three types of messages:

- *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 function 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*(), *putmsg*(), *putpmsg*(), or *ioctl*(), messages are sent down the STREAM, and *read*(), *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 is created that contains two modules: the STREAM head module and the STREAM end (driver) module. If pipes are STREAMS-based in an implementation, when a pipe is created, two STREAMS are created, each containing a STREAM head module. 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 are always 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 are discarded if there is already a high-priority message in the queue. Their priority band is 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 are not affected by flow control. Normal and priority

messages have separate flow controls.

Message Parts

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

2.6.1 Accessing STREAMS

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

Calls to *ioctl*( ) are used to perform control functions with the STREAMS-based device associated with the file descriptor *fildes*. The arguments *command* and *arg* are passed to the STREAMS file designated by *fildes* and are interpreted by the STREAM head. Certain combinations of these arguments may be passed to a module or driver in the STREAM.

Since these STREAMS requests are a subset of *ioctl*( ), they are subject to the errors described there.

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

2.7 XSI Interprocess Communication

XSI This section describes extensions to support interprocess communication. This functionality is dependent on support of the XSI Extension (and the rest of this section is not further shaded for this option).

The following message passing, semaphore, and shared memory services form an XSI interprocess communication facility. Certain aspects of their operation are common, and are described below.

IPC Functions		
msgctl( )	*semctl*( )	*shmctl*( )
msgget( )	*semget*( )	*shmdt*( )
msgrcv( )	*semop*( )	*shmget*( )
msgsnd( )	*shmat*( )	

Another interprocess communication facility is provided by functions in the Realtime Option Group; see Section 2.8 (on page 543).

2.7.1 IPC General Description

Each individual shared memory segment, message queue, and semaphore set is identified by a unique positive integer, called respectively a shared memory identifier, *shmid*, a semaphore identifier, *semid*, and a message queue identifier, *msqid*. The identifiers are returned by calls to *shmget*( ), *semget*( ), and *msgget*( ), respectively.

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

Each of the data structures contains both ownership information and an **ipc_perm** structure (see the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/ipc.h>**) which are used in conjunction to determine whether or not read/write (read/alter for semaphores) permissions should be granted to processes using the IPC facilities. The *mode* member of the **ipc_perm** structure acts as a bit field which determines the permissions.

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.

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

- The process has appropriate privileges.
- The effective user ID of the process matches *xxx_perm.cuid* or *xxx_perm.uid* in the data structure associated with the IPC identifier, and the appropriate bit of the *user* field in *xxx_perm.mode* is set.

- The effective user ID of the process does not match *xxx_perm.cuid* or *xxx_perm.uid* but the effective group ID of the process matches *xxx_perm.cgid* or *xxx_perm.gid* in the data structure associated with the IPC identifier, and the appropriate bit of the *group* field in *xxx_perm.mode* is set.
- The effective user ID of the process does not match *xxx_perm.cuid* or *xxx_perm.uid* and the effective group ID of the process does not match *xxx_perm.cgid* or *xxx_perm.gid* in the data structure associated with the IPC identifier, but the appropriate bit of the *other* field in *xxx_perm.mode* is set.

Otherwise, the permission is denied.

2.8 Realtime

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

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

- Semaphores
- Process Memory Locking
- Memory Mapped Files and Shared Memory Objects
- Priority Scheduling
- Realtime Signal Extension
- Timers
- Interprocess Communication
- Synchronized Input and Output
- Asynchronous Input and Output

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

2.8.1 Realtime Signals

RTS Realtime signal generation and delivery is dependent on support for the Realtime Signals Extension option.

See Section 2.4.2 (on page 529).

2.8.2 Asynchronous I/O

AIO The functionality described in this section is dependent on support of the Asynchronous Input and Output option (and the rest of this section is not further shaded for this option).

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

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

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

If O_APPEND is not set for the file descriptor *aio_fildes* and if *aio_fildes* is associated with a device that is capable of seeking, then the requested operation takes place at the absolute position in the file as given by *aio_offset*, as if *lseek*( ) were called immediately prior to the

operation with an *offset* argument equal to *aio_offset* and a *whence* argument equal to SEEK_SET. If O_APPEND is set for the file descriptor, or if *aio_fildes* is associated with a device that is incapable of seeking, write operations append to the file in the same order as the calls were made, with the following exception: under implementation-defined circumstances, such as operation on a multiprocessor or when requests of differing priorities are submitted at the same time, the ordering restriction may be relaxed. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified. The *aio_nbytes* and *aio_buf* elements are the same as the *nbyte* and *buf* arguments defined by *read*( ) and *write*( ), respectively.

If _POSIX_PRIORITIZED_IO and _POSIX_PRIORITY_SCHEDULING are defined, then asynchronous I/O is queued in priority order, with the priority of each asynchronous operation based on the current scheduling priority of the calling process. The *aio_reqprio* member can be used to lower (but not raise) the asynchronous I/O operation priority and is within the range zero through {AIO_PRIO_DELTA_MAX}, inclusive. Unless both _POSIX_PRIORITIZED_IO and _POSIX_PRIORITY_SCHEDULING are defined, the order of processing asynchronous I/O requests is unspecified. When both _POSIX_PRIORITIZED_IO and _POSIX_PRIORITY_SCHEDULING are defined, the order of processing of requests submitted by processes whose schedulers are not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC is unspecified. The priority of an asynchronous request is computed as (process scheduling priority) minus *aio_reqprio*. The priority assigned to each asynchronous I/O request is an indication of the desired order of execution of the request relative to other asynchronous I/O requests for this file. If _POSIX_PRIORITIZED_IO is defined, requests issued with the same priority to a character special file are processed by the underlying device in FIFO order; the order of processing of requests of the same priority issued to files that are not character special files is unspecified. Numerically higher priority values indicate requests of higher priority. The value of *aio_reqprio* has no effect on process scheduling priority. When prioritized asynchronous I/O requests to the same file are blocked waiting for a resource required for that I/O operation, the higher-priority I/O requests shall be granted the resource before lower-priority I/O requests are granted the resource. The relative priority of asynchronous I/O and synchronous I/O is implementation-defined. If _POSIX_PRIORITIZED_IO is defined, the implementation shall define for which files I/O prioritization is supported.

The *aio_sigevent* determines how the calling process shall be notified upon I/O completion, as specified in Section 2.4.1 (on page 528). If *aio_sigevent.sigev_notify* is SIGEV_NONE, then no signal shall be posted upon I/O completion, but the error status for the operation and the return status for the operation shall be set appropriately.

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

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

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

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

2.8.3 Memory Management

2.8.3.1 Memory Locking

ML The functionality described in this section is dependent on support of the Process Memory Locking option (and the rest of this section is not further shaded for this option).

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

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

2.8.3.2 Memory Mapped Files

MF The functionality described in this section is dependent on support of the Memory Mapped Files option (and the rest of this section is not further shaded for this option).

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

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

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

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

2.8.3.3 Memory Protection

MPR MF The functionality described in this section is dependent on support of the Memory Protection and Memory Mapped Files option (and the rest of this section is not further shaded for these options).

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

- A mapping may be restricted to disallow some types of access.

- Write attempts to memory that was mapped without write access, or any access to memory mapped PROT_NONE, shall result in a SIGSEGV signal.
- References to unmapped addresses shall result in a SIGSEGV signal.
- Reference to whole pages within the mapping, but beyond the current length of the object, shall result in a SIGBUS signal.
- The size of the object is unaffected by access beyond the end of the object (even if a SIGBUS is not generated).

2.8.3.4 *Typed Memory Objects*

TYM The functionality described in this section is dependent on support of the Typed Memory Objects option (and the rest of this section is not further shaded for this option).

Implementations may support the Typed Memory Objects option without supporting the Memory Mapped Files option or the Shared Memory Objects option. Typed memory objects are implementation-configurable named storage pools accessible from one or more processors in a system, each via one or more ports, such as backplane buses, LANs, I/O channels, and so on. Each valid combination of a storage pool and a port is identified through a name that is defined at system configuration time, in an implementation-defined manner; the name may be independent of the file system. Using this name, a typed memory object can be opened and mapped into process address space. For a given storage pool and port, it is necessary to support both dynamic allocation from the pool as well as mapping at an application-supplied offset within the pool; when dynamic allocation has been performed, subsequent deallocation must be supported. Lastly, accessing typed memory objects from different ports requires a method for obtaining the offset and length of contiguous storage of a region of typed memory (dynamically allocated or not); this allows typed memory to be shared among processes and/or processors while being accessed from the desired port.

2.8.4 Process Scheduling

Scheduling Policies

PS The functionality described in this section is dependent on support of the Process Scheduling option (and the rest of this section is not further shaded for this option).

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

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

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

Each thread shall be controlled by an associated scheduling policy and priority. These parameters may be specified by explicit application execution of the *pthread_setschedparam*( ) function.

Associated with each policy is a priority range. Each policy definition shall specify the minimum priority range for that policy. The priority ranges for each policy may but need not overlap the priority ranges of other policies.

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

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

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

SCHED_RR Round robin scheduling policy.

SS SCHED_SPORADIC Sporadic server scheduling policy.

SCHED_OTHER Another scheduling policy.

The values of these symbols shall be distinct.

SCHED_FIFO

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

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

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

1. When a running thread becomes a preempted thread, it becomes the head of the thread list for its priority.
2. When a blocked thread becomes a runnable thread, it becomes the tail of the thread list for its priority.
3. When a running thread calls the *sched_setscheduler*( ) function, the process specified in the function call is modified to the specified policy and the priority specified by the *param* argument.
4. When a running thread calls the *sched_setparam*( ) function, the priority of the process specified in the function call is modified to the priority specified by the *param* argument.
5. When a running thread calls the *pthread_setschedparam*( ) function, the thread specified in the function call is modified to the specified policy and the priority specified by the *param* argument.
6. If a thread whose policy or priority has been modified is a running thread or is runnable, it then becomes the tail of the thread list for its new priority.
7. When a running thread issues the *sched_yield*( ) function, the thread becomes the tail of the thread list for its priority.

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

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

SCHED_RR

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

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

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

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

SCHED_SPORADIC

SS | TSP The functionality described in this section is dependent on support of the Process Sporadic Server or Thread Sporadic Server options (and the rest of this section is not further shaded for these options).

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

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

The priority assigned to a thread using the sporadic server scheduling policy is determined in the following manner: if the available execution capacity is greater than zero and the number of pending replenishment operations is strictly less than *sched_ss_max_repl*, the thread is assigned the priority specified by *sched_priority*; otherwise, the assigned priority shall be *sched_ss_low_priority*. If the value of *sched_priority* is less than or equal to the value of *sched_ss_low_priority*, the results are undefined. When active, the thread shall belong to the thread list corresponding to its assigned priority level, according to the mentioned priority assignment. The modification of the available execution capacity and, consequently of the assigned priority, is done as follows:

1. When the thread at the head of the *sched_priority* list becomes a running thread, its execution time shall be limited to at most its available execution capacity, plus the resolution of the execution time clock used for this scheduling policy. This resolution shall be implementation-defined.

2. Each time the thread is inserted at the tail of the list associated with *sched_priority*—because as a blocked thread it became runnable with priority *sched_priority* or because a replenishment operation was performed—the time at which this operation is done is posted as the *activation_time*.

3. When the running thread with assigned priority equal to *sched_priority* becomes a preempted thread, it becomes the head of the thread list for its priority, and the execution time consumed is subtracted from the available execution capacity. If the available execution capacity would become negative by this operation, it shall be set to zero.

4. When the running thread with assigned priority equal to *sched_priority* becomes a blocked thread, the execution time consumed is subtracted from the available execution capacity, and a replenishment operation is scheduled, as described in 6 and 7. If the available execution capacity would become negative by this operation, it shall be set to zero.

5. When the running thread with assigned priority equal to *sched_priority* reaches the limit imposed on its execution time, it becomes the tail of the thread list for *sched_ss_low_priority*, the execution time consumed is subtracted from the available execution capacity (which becomes zero), and a replenishment operation is scheduled, as described in 6 and 7.

6. Each time a replenishment operation is scheduled, the amount of execution capacity to be replenished, *replenish_amount*, is set equal to the execution time consumed by the thread since the *activation_time*. The replenishment is scheduled to occur at *activation_time* plus *sched_ss_repl_period*. If the scheduled time obtained is before the current time, the replenishment operation is carried out immediately. Several replenishment operations may be pending at the same time, each of which will be serviced at its respective scheduled time. With the above rules, the number of replenishment operations simultaneously pending for a given thread that is scheduled under the sporadic server policy shall not be greater than *sched_ss_max_repl*.

7. A replenishment operation consists of adding the corresponding *replenish_amount* to the available execution capacity at the scheduled time. If, as a consequence of this operation, the execution capacity would become larger than *sched_ss_initial_budget*, it shall be rounded down to a value equal to *sched_ss_initial_budget*. Additionally, if the thread was runnable or running, and had assigned priority equal to *sched_ss_low_priority*, then it becomes the tail of the thread list for *sched_priority*.

Execution time is defined in Section 2.2.2 (on page 513).

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

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

SCHED_OTHER

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

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

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

2.8.5 Clocks and Timers

TMR The functionality described in this section is dependent on support of the Timers option (and the rest of this section is not further shaded for this option).

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

Time Value Specification Structures

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

Member Type	Member Name	Description
time_t	*tv_sec*	Seconds.
long	*tv_nsec*	Nanoseconds.

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

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

Member Type	Member Name	Description
struct timespec	*it_interval*	Timer period.
struct timespec	*it_value*	Timer expiration.

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

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

Timer Event Notification Control Block

RTS Per-process timers may be created that notify the process of timer expirations by queuing a realtime extended signal. The **sigevent** structure, defined in the Base Definitions volume of IEEE Std. 1003.1-200x, **<signal.h>**, is used in creating such a timer. The **sigevent** structure contains the signal number and an application-specific data value to be used when notifying the calling process of timer expiration events.

Manifest Constants

The following constants are defined in the Base Definitions volume of IEEE Std. 1003.1-200x, **<time.h>**:

CLOCK_REALTIME The identifier for the system-wide realtime clock.

TIMER_ABSTIME Flag indicating time is absolute with respect to the clock associated with a timer.

MON CLOCK_MONOTONIC The identifier for the system-wide monotonic clock, which is defined as a clock whose value cannot be set via *clock_settime*( ) and which cannot have backward clock jumps. The maximum possible clock jump is implementation-defined.

MON The maximum allowable resolution for the CLOCK_REALTIME and the CLOCK_MONOTONIC clocks and all time services based on these clocks is represented by {_POSIX_CLOCKRES_MIN} and is defined as 20ms (1/50 of a second). Implementations may support smaller values of resolution for these clocks to provide finer granularity time bases. The actual resolution supported by an implementation for a specific clock is obtained using the *clock_getres*( ) function. If the actual resolution supported for a time service based on one of these clocks differs from the resolution supported for that clock, the implementation shall document this difference.

MON The minimum allowable maximum value for the CLOCK_REALTIME and the CLOCK_MONOTONIC clocks and all absolute time services based on them is the same as that defined by the ISO C standard for the **time_t** type. If the maximum value supported by a time service based on one of these clocks differs from the maximum value supported by that clock, the implementation shall document this difference.

Execution Time Monitoring

CPT If _POSIX_CPUTIME is defined, process CPU-time clocks shall be supported in addition to the clocks described in **Manifest Constants**.

TCT If _POSIX_THREAD_CPUTIME is defined, thread CPU-time clocks shall be supported.

CPT|TCT CPU-time clocks measure execution or CPU time, which is defined in the Base Definitions volume of IEEE Std. 1003.1-200x, Section 3.120, CPU Time (Execution Time). The mechanism used to measure execution time is described in the Base Definitions volume of IEEE Std. 1003.1-200x, Section 4.7, Measurement of Execution Time.

CPT If _POSIX_CPUTIME is defined, the following constant of the type **clockid_t** shall be defined in **<time.h>**:

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

2219 TCT If _POSIX_THREAD_CPUTIME is defined, the following constant of the type **clockid_t** shall be
2220 defined in **<time.h>**:

2221 CLOCK_THREAD_CPUTIME_ID
2222 When this value of the type **clockid_t** is used in a *clock*() or *timer**() function call, it is
2223 interpreted as the identifier of the CPU-time clock associated with the thread making the
2224 function call.
2225

2.9 Threads

THR The functionality described in this section is dependent on support of the Threads option (and the rest of this section is not further shaded for this option).

This section defines functionality to support multiple flows of control, called *threads*, within a process. For the definition of *threads*, see the Base Definitions volume of IEEE Std. 1003.1-200x, Section 3.395, Thread.

The specific functional areas covered by threads and their scope includes:

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

2.9.1 Thread-Safety

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

asctime( )	*gethostbyname*( )	*getprotobynumber*( )	*inet_ntoa*( )	*ttyname*( )
ctime( )	*gethostent*( )	*getprotoent*( )	*localtime*( )	*unsetenv*( )
getc_unlocked( )	*getlogin*( )	*getpwnam*( )	*putc_unlocked*( )	*wcstombs*( )
getchar_unlocked( )	*getnetbyaddr*( )	*getpwuid*( )	*putchar_unlocked*( )	*wctomb*( )
getenv( )	*getnetbyname*( )	*getservbyname*( )	*rand*( )	
getgrgid( )	*getnetent*( )	*getservbyport*( )	*readdir*( )	
getgrnam( )	*getopt*( )	*getservent*( )	*setenv*( )	
gethostbyaddr( )	*getprotobyname*( )	*gmtime*( )	*strtok*( )	

XSI

basename( )	*dbm_open*( )	*fcvt*( )	*hdestroy*( )	*setgrent*( )
catgets( )	*dbm_store*( )	*gcvt*( )	*hsearch*( )	*setkey*( )
crypt( )	*dirname*( )	*getdate*( )	*l64a*( )	*setpwent*( )
dbm_clearerr( )	*dlerror*( )	*getenv*( )	*lgamma*( )	*setutxent*( )
dbm_close( )	*drand48*( )	*getgrent*( )	*lrand48*( )	*strerror*( )
dbm_delete( )	*ecvt*( )	*getpwent*( )	*mrand48*( )	
dbm_error( )	*encrypt*( )	*getutxent*( )	*nl_langinfo*( )	
dbm_fetch( )	*endgrent*( )	*getutxid*( )	*ptsname*( )	
dbm_firstkey( )	*endpwent*( )	*getutxline*( )	*putenv*( )	
dbm_nextkey( )	*endutxent*( )	*hcreate*( )	*pututxline*( )	

The *read*( ) function need not be thread-safe when reading from a pipe, FIFO, socket, or terminal device.

Note: While a read from a pipe of {PIPE_MAX}*2 bytes may not generate a single atomic and thread-safe stream of bytes, it should generate ‘‘several’’ (individually atomic) thread-safe streams of bytes. Similiarly, while reading from a terminal device may not generate a single atomic and thread-safe stream of bytes, it should generate some finite number of (individually atomic) and thread-safe streams of bytes. That is, concurrent calls to read for a pipe, FIFO, or terminal device are not allowed to result in corrupting the stream of bytes or other internal data. However, *read*( ), in these cases, is not required to return a single contiguous and atomic stream of bytes.

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

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

2.9.2 Thread IDs

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

2.9.3 Thread Mutexes

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

A thread becomes the owner of a mutex, *m*, when one of the following occurs:

- It returns successfully from *pthread_mutex_lock*( ) with *m* as the *mutex* argument.
- It returns successfully from *pthread_mutex_trylock*( ) with *m* as the *mutex* argument.
- TMO It returns successfully from *pthread_mutex_timedwait*( ) with *m* as the mutex argument.
- It returns (successfully or not) from *pthread_cond_wait*( ) with *m* as the *mutex* argument (except as explicitly indicated otherwise for certain errors).
- It returns (successfully or not) from *pthread_cond_timedwait*( ) with *m* as the *mutex* argument (except as explicitly indicated otherwise for certain errors).

The thread remains the owner of *m* until one of the following occurs:

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

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

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

2.9.4 Thread Scheduling

Thread Scheduling Attributes

Thread scheduling attributes are dependent on support of the Thread Execution Scheduling option.

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

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

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

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

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

TSP If _POSIX_THREAD_SPORADIC_SERVER is defined, the *schedparam* attribute supports four new members that are used for the sporadic server scheduling policy. These members are *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*, and *sched_ss_max_repl*. The meaning of these attributes is the same as in the definitions that appear under Section 2.8.4 (on page 546).

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

Thread Scheduling Contention Scope

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

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

- A thread created with PTHREAD_SCOPE_SYSTEM scheduling contention scope contends for resources with all other threads in the same scheduling allocation domain relative to their system scheduling attributes. The system scheduling attributes of a thread created with PTHREAD_SCOPE_SYSTEM scheduling contention scope are the scheduling attributes with which the thread was created. The system scheduling attributes of a thread created with PTHREAD_SCOPE_PROCESS scheduling contention scope are the implementation-defined mapping into system attribute space of the scheduling attributes with which the thread was

created.

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

Scheduling Allocation Domain

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

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

For application threads with scheduling allocation domains of size equal to one, the scheduling rules defined for SCHED_FIFO and SCHED_RR shall be used; see **Scheduling Policies** (on page 546). All threads with system scheduling contention scope, regardless of the processes in which they reside, compete for the processor according to their priorities. Threads with process scheduling contention scope compete only with other threads with process scheduling contention scope within their process.

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

TSP If _POSIX_THREAD_SPORADIC_SERVER is defined, the rules defined for SCHED_SPORADIC in **Scheduling Policies** (on page 546) shall be used in an implementation-defined manner for application threads whose scheduling allocation domain size is greater than one.

Scheduling Documentation

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

2.9.5 Thread Cancelation

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

Cancelation is controlled by the cancelation control functions. Each thread maintains its own cancelability state. Cancelation may only occur at cancelation points or when the thread is asynchronously cancelable.

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

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

2.9.5.1 Cancelability States

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

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

1. Cancelability-Enable: When cancelability is PTHREAD_CANCEL_DISABLE (as defined in the Base Definitions volume of IEEE Std. 1003.1-200x, **<pthread.h>**), cancelation requests against the target thread are held pending. By default, cancelability is set to PTHREAD_CANCEL_ENABLE (as defined in **<pthread.h>**).

2. Cancelability Type: When cancelability is enabled and the cancelability type is PTHREAD_CANCEL_ASYNCHRONOUS (as defined in **<pthread.h>**), new or pending cancelation requests may be acted upon at any time. When cancelability is enabled and the cancelability type is PTHREAD_CANCEL_DEFERRED (as defined in **<pthread.h>**), cancelation requests are held pending until a cancelation point (see below) is reached. If cancelability is disabled, the setting of the cancelability type has no immediate effect as all cancelation requests are held pending; however, once cancelability is enabled again the new type is in effect. The cancelability type is PTHREAD_CANCEL_DEFERRED in all newly created threads including the thread in which *main*( ) was first invoked.

2.9.5.2 Cancelation Points

Cancelation points occur when a thread is executing the following functions:

accept( )	*mq_timedsend*( )	*putpmsg*( )	*sigsuspend*( )
aio_suspend( )	*msgrcv*( )	*pwrite*( )	*sigtimedwait*( )
clock_nanosleep( )	*msgsnd*( )	*read*( )	*sigwait*( )
close( )	*msync*( )	*readv*( )	*sigwaitinfo*( )
connect( )	*nanosleep*( )	*recv*( )	*sleep*( )
creat( )	*open*( )	*recvfrom*( )	*system*( )
fcntl( )[1]	*pause*( )	*recvmsg*( )	*tcdrain*( )
fsync( )	*poll*( )	*select*( )	*usleep*( )
getmsg( )	*pread*( )	*sem_timedwait*( )	*wait*( )
getpmsg( )	*pthread_cond_timedwait*( )	*sem_wait*( )	*waitid*( )
lockf( )	*pthread_cond_wait*( )	*send*( )	*waitpid*( )
mq_receive( )	*pthread_join*( )	*sendmsg*( )	*write*( )
mq_send( )	*pthread_testcancel*( )	*sendto*( )	*writev*( )
mq_timedreceive( )	*putmsg*( )	*sigpause*( )	

1. When the *cmd* argument is F_SETLKW.

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

catclose( )
catgets( )
catopen( )
closedir( )
closelog( )
ctermid( )
dbm_close( )
dbm_delete( )
dbm_fetch( )
dbm_nextkey( )
dbm_open( )
dbm_store( )
dlclose( )
dlopen( )
endgrent( )
endhostent( )
endnetent( )
endprotoent( )
endpwent( )
endservent( )
endutxent( )
fclose( )
fcntl( )[2]
fflush( )
fgetc( )
fgetpos( )
fgets( )
fgetwc( )
fgetws( )
fopen( )
fprintf( )
fputc( )
fputs( )
fputwc( )
fputws( )
fread( )
freopen( )
fscanf( )
fseek( )
fseeko( )
fsetpos( )
ftell( )
ftello( )
ftw( )
fwprintf( )
fwrite( )
fwscanf( )
getc( )
getc_unlocked( )
getchar( )
getchar_unlocked( )
getcwd( )
getdate( )
getgrent( )
getgrgid( )
getgrgid_r( )
getgrnam( )
getgrnam_r( )
gethostbyaddr( )
gethostbyname( )
gethostent( )
gethostname( )
getlogin( )
getlogin_r( )
getnetbyaddr( )
getnetbyname( )
getnetent( )
getprotobyname( )
getprotobynumber( )
getprotoent( )
getpwent( )
getpwnam( )
getpwnam_r( )
getpwuid( )
getpwuid_r( )
gets( )
getservbyname( )
getservbyport( )
getservent( )
getutxent( )
getutxid( )
getutxline( )
getwc( )
getwchar( )
getwd( )
glob( )
iconv_close( )
iconv_open( )
ioctl( )
lseek( )
mkstemp( )
nftw( )
opendir( )
openlog( )
pclose( )
perror( )
popen( )
posix_fadvise( )
posix_fallocate( )
posix_madvise( )
posix_spawn( )
posix_spawnp( )
posix_trace_clear( )
posix_trace_close( )
posix_trace_create( )
posix_trace_create_withlog( )
posix_trace_eventtypelist_getnext_id( )
posix_trace_eventtypelist_rewind( )
posix_trace_flush( )
posix_trace_get_attr( )
posix_trace_get_filter( )
posix_trace_get_status( )
posix_trace_getnext_event( )
posix_trace_open( )
posix_trace_rewind( )
posix_trace_set_filter( )
posix_trace_shutdown( )
posix_trace_timedgetnext_event( )
posix_typed_mem_open( )
printf( )
pthread_rwlock_rdlock( )
pthread_rwlock_timedrdlock( )
pthread_rwlock_timedwrlock( )
pthread_rwlock_wrlock( )
putc( )
putc_unlocked( )
putchar( )
putchar_unlocked( )
puts( )
pututxline( )
putwc( )
putwchar( )
readdir( )
readdir_r( )
remove( )
rename( )
rewind( )
rewinddir( )
scanf( )
seekdir( )
semop( )
setgrent( )
sethostent( )
setnetent( )
setprotoent( )
setpwent( )
setservent( )
setutxent( )
strerror( )
syslog( )
tmpfile( )
tmpnam( )
ttyname( )
ttyname_r( )
ungetc( )
ungetwc( )
unlink( )
vfprintf( )
vfwprintf( )
vprintf( )
vwprintf( )
wprintf( )
wscanf( )

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

---

2. For any value of the *cmd* argument.

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

Whenever a thread has cancelability enabled and a cancelation request has been made with that thread as the target and the thread calls *pthread_testcancel*( ), then the cancelation request is acted upon before *pthread_testcancel*( ) returns. If a thread has cancelability enabled and the thread has a cancelation request pending and the thread is suspended at a cancelation point waiting for an event to occur, then the cancelation request shall be acted upon. However, if the thread is suspended at a cancelation point and the event that it is waiting for occurs before the cancelation request is acted upon, it is unspecified whether the cancelation request is acted upon or whether the request remains pending and the thread resumes normal execution.

2.9.5.3 Thread Cancelation Cleanup Handlers

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

When a cancelation request is acted upon, the routines in the list are invoked one by one in LIFO sequence; that is, the last routine pushed onto the list (Last In) is the first to be invoked (First Out). The thread invokes the cancelation cleanup handler with cancelation disabled until the last cancelation cleanup handler returns. When the cancelation cleanup handler for a scope is invoked, the storage for that scope remains valid. If the last cancelation cleanup handler returns, thread execution is terminated and a status of PTHREAD_CANCELED is made available to any threads joining with the target. The symbolic constant PTHREAD_CANCELED expands to a constant expression of type (**void***) whose value matches no pointer to an object in memory nor the value NULL.

The cancelation cleanup handlers are also invoked when the thread calls *pthread_exit*( ).

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

A cancelation cleanup handler cannot exit via *longjmp*( ) or *siglongjmp*( ).

2.9.5.4 Async-Cancel Safety

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

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

2.9.6 Thread Read-Write Locks

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

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

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

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

2.9.7 Thread Interactions with Regular File Operations

All of the functions *chmod*( ), *close*( ), *fchmod*( ), *fcntl*( ), *fstat*( ), *ftruncate*( ), *lseek*( ), *open*( ), *read*( ), *readlink*( ), *stat*( ), *symlink*( ), and *write*( ) shall be atomic with respect to each other in the effects specified in IEEE Std. 1003.1-200x when they operate on regular files. If two threads each call one of these functions, each call shall either see all of the specified effects of the other call, or none of them.

2.10 Sockets

A socket is an endpoint for communication using the facilities described in this section. A socket is created with a specific socket type, described in Section 2.10.6 (on page 563), and is associated with a specific protocol, detailed in Section 2.10.2. A socket is accessed via a file descriptor obtained when the socket is created.

2.10.1 Protocol Families

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

Section 2.10.17 (on page 569), Section 2.10.18 (on page 569), and Section 2.10.19 (on page 570), respectively, describe the use of sockets for local UNIX connections, for Internet protocols based on IPv4, and for Internet protocols based on IPv6.

2.10.2 Protocols

A protocol supports one of the socket abstractions detailed in Section 2.10.6 (on page 563). Selecting a protocol involves specifying the protocol family, socket type, and protocol number to the *socket*( ) function. Protocols normally accept only one type of address format, usually determined by the addressing structure inherent in the design of the protocol family/network architecture. Certain semantics of the basic socket abstractions are protocol-specific. All protocols are expected to support the basic model for their particular socket type, but may, in addition, provide non-standard facilities or extensions to a mechanism.

2.10.3 Addressing

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

2.10.4 Routing

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

2.10.5 Interfaces

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

2.10.6 Socket Types

A socket is created with a specific type, which defines the communication semantics and which allows the selection of an appropriate communication protocol. Three types are defined: SOCK_STREAM, SOCK_SEQPACKET, and SOCK_DGRAM. Implementations may specify additional socket types.

The SOCK_STREAM socket type provides reliable, sequenced, full-duplex octet streams between the socket and a peer to which the socket is connected. A socket of type SOCK_STREAM must be in a connected state before any data may be sent or received. Record boundaries are not maintained; data sent on a stream socket using output operations of one size may be received using input operations of smaller or larger sizes without loss of data. Data may be buffered; successful return from an output function does not imply that the data has been delivered to the peer or even transmitted from the local system. If data cannot be successfully transmitted within a given time then the connection is considered broken, and subsequent 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.

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

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

2.10.7 Socket I/O Mode

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

When the O_NONBLOCK flag is set, functions that would normally block until they are complete either return immediately with an error, or they complete asynchronously to the execution of the calling process. Data transfer operations (the *read*( ), *write*( ), *send*( ), and *recv*( ) functions) complete immediately, transfer only as much as is available, and then return without blocking, or return an error indicating that no transfer could be made without blocking. The

connect( ) function initiates a connection and returns without blocking when O_NONBLOCK is set; it returns the error [EINPROGRESS] to indicate that the connection was initiated successfully, but that it has not yet completed.

2.10.8 Socket Owner

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

2.10.9 Socket Queue Limits

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

2.10.10 Pending Error

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

2.10.11 Socket Receive Queue

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

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

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

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

2.10.12 Socket Out-of-Band Data State

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

2.10.13 Connection Indication Queue

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

2.10.14 Signals

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

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

If a socket has an owner, the SIGURG signal is sent to the owner of the socket when it is notified of expedited or out-of-band data. The socket state at this time is protocol-dependent, and the status of the socket is specified in Section 2.10.17 (on page 569), Section 2.10.18 (on page 569), and Section 2.10.19 (on page 570). Depending on the protocol, the expedited data may or may not have arrived at the time of signal generation.

2.10.15 Asynchronous Errors

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

[ECONNABORTED]
The connection was aborted locally.

[ECONNREFUSED]
For a connection-mode socket attempting a non-blocking connection, the attempt to connect was forcefully rejected. For a connectionless-mode socket, an attempt to deliver a datagram was forcefully rejected.

[ECONNRESET]
The peer has aborted the connection.

[EHOSTDOWN]
The destination host has been determined to be down or disconnected.

[EHOSTUNREACH]
The destination host is not reachable.

[EMSGSIZE]
For a connectionless-mode socket, the size of a previously sent datagram prevented delivery.

[ENETDOWN]
The local network connection is not operational.

[ENETRESET]
The connection was aborted by the network.

[ENETUNREACH]
The destination network is not reachable.

2.10.16 Use of Options

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

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

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

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

Table 2-4 shows the value for the socket level.

Table 2-4 Value of Level for Socket Options

Name	Description
SOL_SOCKET	Options are intended for the sockets level.

Table 2-5 (on page 567) 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.

Table 2-5 Socket-Level Options

Option	Parameter Type	Parameter Meaning
SO_BROADCAST	(**void** *) **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 (*getsockopt*( ) 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 *close*( ): linger on/off and linger time in seconds.
SO_OOBINLINE	**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 *bind*( ) (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 (*getsockopt*( ) only).

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.

SO_DEBUG 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.

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

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

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

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

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

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

The maximum value for the option for a socket may be obtained by the use of the *fpathconf*( ) function, using the value _PC_SOCK_MAXBUF.

SO_RCVLOWAT sets the minimum number of bytes to process for socket input operations. In general, receive calls block until any (non-zero) amount of data is received, then return the smaller of the amount available or the amount requested. The default value for SO_RCVLOWAT is 1, and does not affect the general case. 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. Receive calls may still 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 than that returned (for example, out-of-band data). As mentioned previously, the default value for SO_RCVLOWAT is 1 byte. It is implementation-defined whether the SO_RCVLOWAT option can be set.

SO_RCVTIMEO is an option to set a timeout value for input operations. 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 returns with a partial count or *errno* set to [EWOULDBLOCK] if no data were received. The default for this option is the value zero, which indicates that a receive operation will not timeout. It is implementation-defined whether the SO_RCVTIMEO option can be set.

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

SO_SNDBUF requests that the buffer space allocated for send operations on this socket be set to the value, in bytes, of the option value. The default value for the SO_SNDBUF option value is implementation-defined, and may vary by protocol. The maximum value for the option for a socket may be obtained by the use of the *fpathconf*( ) function, using the value _PC_SOCK_MAXBUF.

SO_SNDLOWAT sets the minimum number of bytes to process for socket output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Non-blocking output operations process as much data as permitted subject to flow control without blocking, but

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. A *select*( ) operation testing the ability to write to a socket returns true only if the send low water mark could be processed. The default value for SO_SNDLOWAT is implementation-defined and protocol-specific. It is implementation-defined whether the SO_SNDLOWAT option can be set.

SO_SNDTIMEO is an option to set a timeout value for the amount of time that an output function shall block because flow control prevents data from being sent. As noted in Table 2-5 (on page 567), the option value is a **timeval** structure with the number of seconds and microseconds specifying the limit on how long to wait for an output operation to complete. If a send operation has blocked for this much time, it returns with a partial count or *errno* set to [EWOULDBLOCK] if no data were sent. The default for this option is the value zero, which indicates that a send operation will not timeout. It is implementation-defined whether the SO_SNDTIMEO option can be set.

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

2.10.17 Use of Sockets for Local UNIX Connections

Support for UNIX domain sockets is mandatory.

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

2.10.17.1 Headers

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

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

2.10.18 Use of Sockets over Internet Protocols Based on IPv4

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

2.10.18.1 Headers

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

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

2.10.19 Use of Sockets over Internet Protocols Based on IPv6

IP6 This section describes extensions to support sockets over Internet protocols based on IPv6. This functionality is dependent on support of the IPV6 option (and the rest of this section is not further shaded for this option).

To enable smooth transition from IPv4 to IPv6, the features defined in this section may, in certain circumstances, also be used in connection with IPv4; see Section 2.10.19.2 (on page 571).

2.10.19.1 Addressing

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

There are three kinds of IPv6 address:

Unicast
: Identifies a single interface.

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

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

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

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

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

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

The following special IPv6 addresses are defined:

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

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

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

IPv4-compatible addresses
: These are assigned to nodes that support IPv6 and can be used when traffic is ‘‘tunneled’’ through IPv4.

IPv4-mapped addresses
: These are used to represent IPv4 addresses in IPv6 address format; see Section 2.10.19.2 (on

page 571).

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

2.10.19.2 Compatibility with IPv4

The API provides the ability for IPv6 applications to interoperate with applications using IPv4, by using IPv4-mapped IPv6 addresses. These addresses can be generated automatically by the *getipnodebyname*( ) function when the specified host has only IPv4 addresses (as described in *endhostent*( )).

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

An application may use AF_INET6 sockets to bind to a node's IPv4 address by specifying the address as an IPv4-mapped IPv6 address in a **sockaddr_in6** structure in the *bind*( ) function. For an AF_INET6 socket bound to a node's IPv4 address, the system returns the address in the *getsockname*( ) function as an IPv4-mapped IPv6 address in a **sockaddr_in6** structure.

2.10.19.3 Interface Identification

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

2.10.19.4 Options

The following options apply at the IPPROTO_IPV6 level:

IPV6_JOIN_GROUP
: When set via *setsockopt*( ), it joins the application to a multicast group on an interface (identified by its index) and addressed by a given multicast address, enabling packets sent to that address to be read via the socket. If the interface index is specified as zero, the system selects the interface (for example, by looking up the address in a routing table and using the resulting interface).

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

  The value of this option is an **ipv6_mreq** structure.

IPV6_LEAVE_GROUP
: When set via *setsockopt*( ), it removes the application from the multicast group on an interface (identified by its index) and addressed by a given multicast address.

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

2976 The value of this option is an **ipv6_mreq** structure.

2977 IPV6_MULTICAST_HOPS
2978 The value of this option is the hop limit for outgoing multicast IPv6 packets sent via the
2979 socket. Its possible values are the same as those of IPV6_UNICAST_HOPS. If the
2980 IPV6_MULTICAST_HOPS option is not set, a value of 1 is assumed. This option can be set
2981 via *setsockopt*( ) and read via *getsockopt*( ).

2982 IPV6_MULTICAST_IF
2983 The index of the interface to be used for outgoing multicast packets. It can be set via
2984 *setsockopt*( ) and read via *getsockopt*( ).

2985 IPV6_MULTICAST_LOOP
2986 This option controls whether outgoing multicast packets should be delivered back to the
2987 local application when the sending interface is itself a member of the destination multicast
2988 group. If it is set to 1 they are delivered. If it is set to 0 they are not. Other values result in an
2989 [EINVAL] error. This option can be set via *setsockopt*( ) and read via *getsockopt*( ).

2990 IPV6_UNICAST_HOPS
2991 The value of this option is the hop limit for outgoing unicast IPv6 packets sent via the
2992 socket. If the option is not set, or is set to –1, the system selects a default value. Attempts to
2993 set a value less than –1 or greater than 255 result in an [EINVAL] error. This option can be
2994 set via *setsockopt*( ) and read via *getsockopt*( ).

2995 An [EOPNOTSUPP] error results if IPV6_JOIN_GROUP or IPV6_LEAVE_GROUP is used with
2996 *getsockopt*( ).

2997 *2.10.19.5 Headers*

2998 Symbolic constant AF_INET6 is defined in the **<sys/socket.h>** header to identify the IPv6 |
2999 Internet address family. See the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 13, |
3000 Headers. |

3001 The **sockaddr_storage** structure defined in **<sys/socket.h>** is large enough to accommodate a
3002 **sockaddr_in6** structure (see the **<netinet/in.h>** header defined in the Base Definitions volume of |
3003 IEEE Std. 1003.1-200x, Chapter 13, Headers) and is aligned at an appropriate boundary so that |
3004 pointers to it can be cast as pointers to **sockaddr_in6** structures and used to access the fields of
3005 those structures without alignment problems. When a **sockaddr_storage** structure is cast as a
3006 **sockaddr_in6** structure, the *ss_family* field maps onto the *sin6_family* field.

3007 The **<netinet/in.h>**, **<arpa/inet.h>**, and **<netdb.h>** headers contain other definitions used in
3008 connection with IPv6 Internet sockets; see the Base Definitions volume of IEEE Std. 1003.1-200x, |
3009 Chapter 13, Headers. |

2.11 Tracing

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

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

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

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

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

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

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

The events to be traced belong to two classes:

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

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

Each traced process shall have a mapping of the trace event names to trace event type identifiers that have been defined for that process. Each active trace stream shall have a mapping that incorporates all the trace event type identifiers predefined by the trace system plus all the mappings of trace event names to trace event type identifiers of the processes that are being traced into that trace stream. These mappings are defined from the instrumented application by

calling the *posix_trace_eventid_open*( ) function and from the controller process by calling the *posix_trace_trid_eventid_open*( ) function. For a pre-recorded trace stream, the list of trace event types is obtained from the pre-recorded trace log.

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

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

Results are undefined if the application performs any operation on a file descriptor associated with an active or pre-recorded trace stream until *posix_trace_shutdown*( ) or *posix_trace_close*( ) is called for that trace stream.

The main purpose of this option is to define a complete set of functions and concepts that allow a portable application to be traced from birth to death, whatever its realtime constraints and properties.

2.11.1 Tracing Data Definitions

2.11.1.1 Structures

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

posix_trace_status_info Structure

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

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

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

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

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

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

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

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

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

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

POSIX_TRACE_NOT_FULL
There is still space available in the trace stream.

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

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

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

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

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

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

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

POSIX_TRACE_NO_OVERRUN
No trace events were lost.

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

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

An overrun occurs when:

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

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

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

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

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

POSIX_TRACE_NOT_FLUSHING
: No flush operation is in progress.

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

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

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

POSIX_TRACE_OVERRUN
: At least one trace event was lost.

POSIX_TRACE_NO_OVERRUN
: No trace events were lost.

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

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

POSIX_TRACE_FULL
: The space in the trace log is exhausted.

POSIX_TRACE_NOT_FULL
: There is still space available in the trace log.

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

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

posix_trace_event_info Structure

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

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

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

In addition, if the Trace option and the Threads option are both supported, the **posix_trace_event_info** structure defined in **<trace.h>** shall contain the following additional member:

Member Type	Member Name	Description
pthread_t	*posix_thread_id*	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.

If the Threads option is supported in addition to the Trace option:

- 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.

Otherwise, this member is undefined.

2.11.1.2 Trace Stream Attributes

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

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

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

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

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

- If the file type corresponding to the trace log supports the POSIX_TRACE_LOOP or the POSIX_TRACE_UNTIL_FULL policies, the *log-max-size* attribute defines the maximum size

in bytes of the trace log associated with an active trace stream. Other stream data—for example, trace attribute values—shall not be included in this size.

- The *log-full-policy* attribute defines the policy of a trace log associated with an active trace stream to be POSIX_TRACE_LOOP, POSIX_TRACE_UNTIL_FULL, or POSIX_TRACE_APPEND.

2.11.2 Trace Event Type Definitions

2.11.2.1 System Trace Event Type Definitions

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

- POSIX_TRACE_START shall be associated with a trace start operation.
- POSIX_TRACE_STOP shall be associated with a trace stop operation.
- if the Trace Event Filter option is supported, POSIX_TRACE_FILTER shall be associated with a trace event type filter change operation.

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

- POSIX_TRACE_OVERFLOW shall mark the beginning of a trace overflow condition.
- POSIX_TRACE_RESUME shall mark the end of a trace overflow condition.
- If the Trace Log option is supported, POSIX_TRACE_FLUSH_START shall mark the beginning of a flush operation.
- If the Trace Log option is supported, POSIX_TRACE_FLUSH_STOP shall mark the end of a flush operation.
- If an implementation-defined trace error condition is reported, it shall be marked POSIX_TRACE_ERROR.

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

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

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

The POSIX_TRACE_STOP trace event with an associated data that is different from zero indicates an automatic stop of the trace stream (see *posix_trace_attr_getstreamfullpolicy*( ) defined in the System Interfaces volume of IEEE Std. 1003.1-200x).

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

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

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

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

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

If the Trace option is supported and the Trace Event Filter option and the Trace Log option are not supported, the following predefined system trace events in Table 2-6 shall be defined:

Table 2-6 Trace Option: System Trace Events

Event Name	Constant	Associated 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.

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 2-7 shall be defined:

Table 2-7 Trace and Trace Event Filter Options: System Trace Events

Event Name	Constant	Associated 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.

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 2-8 (on page 581) shall be defined:

Table 2-8 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 2-9 shall be defined:

Table 2-9 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 shall be 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 2-10 (on page 582) shall be defined:

Table 2-10 Trace Option: User Trace Event

Event Name	Constant
"posix_trace_unnamed_userevent"	POSIX_TRACE_UNNAMED_USEREVENT

2.11.3 Trace Functions

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:

- A trace controller process requires functions to set up and customize all the resources needed to run a trace stream, including:
  - Attribute initialization and destruction (*posix_trace_attr_init*())
  - Identification information manipulation (*posix_trace_attr_getgenversion*())
  - Trace system behavior modification (*posix_trace_attr_getinherited*())
  - Trace stream and trace log size set (*posix_trace_attr_getmaxusereventsize*())
  - Trace stream creation, flush, and shutdown (*posix_trace_create*())
  - Trace stream and trace log clear (*posix_trace_clear*())
  - Trace event type identifier manipulation (*posix_trace_trid_eventid_open*())
  - Trace event type identifier list exploration (*posix_trace_eventtypelist_getnext_id*())
  - Trace event type set manipulation (*posix_trace_eventset_empty*())
  - Trace event type filter set (*posix_trace_set_filter*())
  - Trace stream start and stop (*posix_trace_start*())
  - Trace stream information and status read (*posix_trace_get_attr*())
- A traced process requires functions to instrument trace points:
  - Trace event type identifiers definition and trace points insertion (*posix_trace_event*())
- A trace analyzer process requires functions to retrieve information from a trace stream and trace log:
  - Identification information read (*posix_trace_attr_getgenversion*())
  - Trace system behavior information read (*posix_trace_attr_getinherited*())
  - Trace stream and trace log size get (*posix_trace_attr_getmaxusereventsize*())
  - Trace event type identifier manipulation (*posix_trace_trid_eventid_open*())
  - Trace event type identifier list exploration (*posix_trace_eventtypelist_getnext_id*())
  - Trace log open, rewind, and close (*posix_trace_open*())
  - Trace stream information and status read (*posix_trace_get_attr*())
  - trace event read (*posix_trace_getnext_event*())

2.12 Data Types

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

Defined Type	Description
cc_t	Type used for terminal special characters.
clock_t	Arithmetic type used for processor times, as defined in the ISO C standard.
clockid_t	Used for clock ID type in some timer functions.
dev_t	Arithmetic type used for device numbers.
DIR	Type representing a directory stream.
div_t	Structure type returned by the *div*( ) function.
FILE	Structure containing information about a file.
glob_t	Structure type used in path name pattern matching.
fpos_t	Type containing all information needed to specify uniquely every position within a file.
gid_t	Arithmetic type used for group IDs.
iconv_t	Type used for conversion descriptors.
id_t	Arithmetic type used as a general identifier; can be used to contain at least the largest of a **pid_t**, **uid_t**, or **gid_t**.
ino_t	Arithmetic type used for file serial numbers.
key_t	Arithmetic type used for XSI interprocess communication.
ldiv_t	Structure type returned by the *ldiv*( ) function.
mode_t	Arithmetic type used for file attributes.
mqd_t	Used for message queue descriptors.
nfds_t	Integer type used for the number of file descriptors.
nlink_t	Arithmetic type used for link counts.
off_t	Signed arithmetic type used for file sizes.
pid_t	Signed arithmetic type used for process and process group IDs.
pthread_attr_t	Used to identify a thread attribute object.
pthread_cond_t	Used for condition variables.
pthread_condattr_t	Used to identify a condition attribute object.
pthread_key_t	Used for thread-specific data keys.
pthread_mutex_t	Used for mutexes.
pthread_mutexattr_t	Used to identify a mutex attribute object.
pthread_once_t	Used for dynamic package initialization.
pthread_rwlock_t	Used for read-write locks.
pthread_rwlockattr_t	Used for read-write lock attributes.
pthread_t	Used to identify a thread.
ptrdiff_t	Signed integer type of the result of subtracting two pointers.
regex_t	Structure type used in regular expression matching.
regmatch_t	Structure type used in regular expression matching.
rlim_t	Unsigned arithmetic type used for limit values, to which objects of type **int** and **off_t** can be cast without loss of value.
sem_t	Type used in performing semaphore operations.
sig_atomic_t	Integer type of an object that can be accessed as an atomic

Defined Type	Description
	entity, even in the presence of asynchronous interrupts.
sigset_t	Integer or structure type of an object used to represent sets of signals.
size_t	Unsigned integer type used for size of objects.
speed_t	Type used for terminal baud rates.
ssize_t	Arithmetic type used for a count of bytes or an error indication.
suseconds_t	Signed arithmetic type used for time in microseconds.
tcflag_t	Type used for terminal modes.
time_t	Arithmetic type used for time in seconds, as defined in the ISO C standard.
timer_t	Used for timer ID returned by the *timer_create*( ) function.
uid_t	Arithmetic type used for user IDs.
useconds_t	Integer type used for time in microseconds.
va_list	Type used for traversing variable argument lists.
wchar_t	Integer type whose range of values can represent distinct codes for all members of the largest extended character set specified by the supported locales.
wctype_t	Scalar type which represents a character class descriptor.
wint_t	Integer type capable of storing any valid value of **wchar_t** or WEOF.
wordexp_t	Structure type used in word expansion.

Chapter 3

System Interfaces

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

NAME

FD_CLR — macros for synchronous I/O multiplexing

SYNOPSIS

```
#include <sys/time.h>

FD_CLR(int fd, fd_set *fdset);
FD_ISSET(int fd, fd_set *fdset);
FD_SET(int fd, fd_set *fdset);
FD_ZERO(fd_set *fdset);
```

DESCRIPTION

Refer to *select*( ).

NAME

_Exit, _exit — terminate a process

SYNOPSIS

```
#include <unistd.h>

void _Exit(int status);
void _exit(int status);
```

DESCRIPTION

Refer to *exit*( ).

NAME

_longjmp, _setjmp — non-local goto

SYNOPSIS

XSI

```
#include <setjmp.h>

void _longjmp(jmp_buf env, int val);
int _setjmp(jmp_buf env);
```

DESCRIPTION

The *_longjmp*( ) and *_setjmp*( ) functions are identical to *longjmp*( ) and *setjmp*( ), respectively, with the additional restriction that *_longjmp*( ) and *_setjmp*( ) do not manipulate the signal mask.

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

RETURN VALUE

Refer to *longjmp*( ) and *setjmp*( ).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

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

Passing *longjmp*( ) a pointer to a buffer not created by *setjmp*( ), passing *_longjmp*( ) a pointer to a buffer not created by *_setjmp*( ), passing *siglongjmp*( ) a pointer to a buffer not created by *sigsetjmp*( ), or passing any of these three functions a buffer that has been modified by the user can cause all the problems listed above, and more.

The *_longjmp*( ) and *_setjmp*( ) functions are included to support programs written to historical system interfaces. New applications should use *siglongjmp*( ) and *sigsetjmp*( ) respectively.

RATIONALE

None.

FUTURE DIRECTIONS

The *_longjmp*( ) and *_setjmp*( ) functions may be marked LEGACY in a future version.

SEE ALSO

longjmp( ), *setjmp*( ), *siglongjmp*( ), *sigsetjmp*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<setjmp.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

3555 **NAME**

3556 _setjmp — set jump point for a non-local goto

3557 **SYNOPSIS**

3558 XSI
```
#include <setjmp.h>
```

3559
```
int _setjmp(jmp_buf env);
```

3560

3561 **DESCRIPTION**

3562 Refer to *_longjmp*( ).

NAME

_tolower — transliterate uppercase characters to lowercase

SYNOPSIS

XSI
```
#include <ctype.h>

int _tolower(int c);
```

DESCRIPTION

The *_tolower*( ) macro shall be equivalent to *tolower*(*c*) except that the application shall ensure that the argument *c* is an uppercase letter.

RETURN VALUE

Upon successful completion, *_tolower*( ) shall return the lowercase letter corresponding to the argument passed.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

tolower( ), *isupper*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

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

Issue 4

The RETURN VALUE section is expanded.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

_toupper — transliterate lowercase characters to uppercase

SYNOPSIS

XSI
```
#include <ctype.h>

int _toupper(int c);
```

DESCRIPTION

The *_toupper*( ) macro shall be equivalent to *toupper*( ) except that the application shall ensure that the argument *c* is a lowercase letter.

RETURN VALUE

Upon successful completion, *_toupper*( ) shall return the uppercase letter corresponding to the argument passed.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

islower( ), *toupper*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

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

Issue 4

The RETURN VALUE section is expanded.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

a64l, l64a — convert between a 32-bit integer and a radix-64 ASCII string

SYNOPSIS

XSI
```
#include <stdlib.h>

long a64l(const char *s);
char *l64a(long value);
```

DESCRIPTION

These functions are used to maintain numbers stored in radix-64 ASCII characters. This is a notation by which 32-bit integers can be represented by up to six characters; each character represents a digit in radix-64 notation. If the type **long** contains more than 32 bits, only the low-order 32 bits shall be used for these operations.

The characters used to represent digits are `'.'` (dot) for 0, `'/'` for 1, `'0'` through `'9'` for 2-11, `'A'` through `'Z'` for 12-37, and `'a'` through `'z'` for 38-63.

The *a64l*( ) function shall take a pointer to a radix-64 representation, in which the first digit is the least significant, and return a corresponding **long** value. If the string pointed to by *s* contains more than six characters, *a64l*( ) shall use the first six. If the first six characters of the string contain a null terminator, *a64l*( ) shall use only characters preceding the null terminator. The *a64l*( ) function scans the character string from left to right with the least significant digit on the left, decoding each character as a 6-bit radix-64 number. If the type **long** contains more than 32 bits, the resulting value is sign-extended. The behavior of *a64l*( ) is unspecified if *s* is a null pointer or the string pointed to by *s* was not generated by a previous call to *l64a*( ).

The *l64a*( ) function shall take a **long** argument and return a pointer to the corresponding radix-64 representation. The behavior of *l64a*( ) is unspecified if *value* is negative.

The value returned by *l64a*( ) may be a pointer into a static buffer. Subsequent calls to *l64a*( ) may overwrite the buffer.

The *l64a*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *a64l*( ) shall return the **long** value resulting from conversion of the input string. If a string pointed to by *s* is an empty string, *a64l*( ) shall return 0L.

The *l64a*( ) function shall return a pointer to the radix-64 representation. If *value* is 0L, *l64a*( ) shall return a pointer to an empty string.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

If the type **long** contains more than 32 bits, the result of *a64l*(*l64a*(*x*)) is *x* in the low-order 32 bits.

RATIONALE

This is not the same encoding as used by either encoding variant of the *uuencode* utility.

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 moved to the DESCRIPTION.

A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

NAME

abort — generate an abnormal process abort

SYNOPSIS

```
#include <stdlib.h>

void abort(void);
```

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 *abort*( ) function shall cause abnormal process termination to occur, unless the signal SIGABRT is being caught and the signal handler does not return.

CX The abnormal termination processing shall include at least the effect of *fclose*( ) on all open streams and the default actions defined for SIGABRT.

XSI On XSI-conformant systems, in addition the abnormal termination processing shall include the effect of *fclose*( ) on message catalog descriptors.

The SIGABRT signal shall be sent to the calling process as if by means of *raise*( ) with the argument SIGABRT.

CX The status made available to *wait*( ) or *waitpid*( ) by *abort*( ) shall be that of a process terminated by the SIGABRT signal. The *abort*( ) function shall override blocking or ignoring the SIGABRT signal.

RETURN VALUE

The *abort*( ) function shall not return.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Catching the signal is intended to provide the application writer with a portable means to abort processing, free from possible interference from any implementation-defined library functions. If SIGABRT is neither caught nor ignored, then the actions associated with SIGABRT defined in Section 2.4.1 (on page 528) will be taken.

RATIONALE

None.

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 4

The following changes are incorporated in this issue for alignment with the ISO C standard and the ISO POSIX-1 standard:

- The argument list is explicitly defined as **void**.
- The DESCRIPTION is revised to identify the correct order in which operations occur. It also identifies:
  - How the calling process is signaled
  - How status information is made available to the host environment
  - That *abort*( ) overrides blocking or ignoring of the SIGABRT signal
- The APPLICATION USAGE section is replaced.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

abs — return an integer absolute value

SYNOPSIS

```
#include <stdlib.h>

int abs(int i);
```

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 *abs*( ) function shall compute the absolute value of its integer operand, *i*. If the result cannot be represented, the behavior is undefined.

RETURN VALUE

The *abs*( ) function shall return the absolute value of its integer operand.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

In two's-complement representation, the absolute value of the negative integer with largest magnitude {INT_MIN} might not be representable.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fabs( ), *labs*( ), 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 4

In the APPLICATION USAGE section, the phrase ''{INT_MIN} is undefined'' is replaced with ''{INT_MIN} might not be representable''.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

accept — accept a new connection on a socket

SYNOPSIS

```
#include <sys/socket.h>

int accept(int socket, struct sockaddr *restrict address,
    socklen_t *restrict address_len);
```

DESCRIPTION

The *accept*( ) function shall extract the first connection on the queue of pending connections, create a new socket with the same socket type protocol and address family as the specified socket, and allocate a new file descriptor for that socket.

The *accept*( ) function takes the following arguments:

socket Specifies a socket that was created with *socket*( ), has been bound to an address with *bind*( ), and has issued a successful call to *listen*( ).

address Either a null pointer, or a pointer to a **sockaddr** structure where the address of the connecting socket shall be returned.

address_len 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.

If *address* is not a null pointer, the address of the peer for the accepted connection shall be stored in the **sockaddr** structure pointed to by *address*, and the length of this address shall be stored in the object pointed to by *address_len*.

If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the stored address shall be truncated.

If the protocol permits connections by unbound clients, and the peer is not bound, then the value stored in the object pointed to by *address* is unspecified.

If the listen queue is empty of connection requests and O_NONBLOCK is not set on the file descriptor for the socket, *accept*( ) shall block until a connection is present. If the *listen*( ) queue is empty of connection requests and O_NONBLOCK is set on the file descriptor for the socket, *accept*( ) shall fail and set *errno* to [EAGAIN] or [EWOULDBLOCK].

The accepted socket cannot itself accept more connections. The original socket remains open and can accept more connections.

RETURN VALUE

Upon successful completion, *accept*( ) shall return the non-negative file descriptor of the accepted socket. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *accept*( ) function shall fail if:

[EAGAIN] or [EWOULDBLOCK]
O_NONBLOCK is set for the socket file descriptor and no connections are present to be accepted.

[EBADF] The *socket* argument is not a valid file descriptor.

[ECONNABORTED]
A connection has been aborted.

[EINTR] The *accept*( ) function was interrupted by a signal that was caught before a valid connection arrived.

[EINVAL] The *socket* is not accepting connections.

[EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

[ENFILE] The maximum number of file descriptors in the system are already open.

[ENOTSOCK] The *socket* argument does not refer to a socket.

[EOPNOTSUPP] The socket type of the specified socket does not support accepting connections.

The *accept*( ) function may fail if:

[ENOBUFS] No buffer space is available.

[ENOMEM] There was insufficient memory available to complete the operation. |

XSR [EPROTO] A protocol error has occurred; for example, the STREAMS protocol stack has not been initialized.

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. | |

3833 **NAME**

3834 access — determine accessibility of a file

3835 **SYNOPSIS**

3836 ```
#include <unistd.h>
```

3837 ```
int access(const char *path, int amode);
```

3838 **DESCRIPTION**

3839 The *access*( ) function shall check the file named by the path name pointed to by the *path*
3840 argument for accessibility according to the bit pattern contained in *amode*, using the real user ID
3841 in place of the effective user ID and the real group ID in place of the effective group ID.

3842 The value of *amode* is either the bitwise-inclusive OR of the access permissions to be checked
3843 (R_OK, W_OK, X_OK) or the existence test (F_OK).

3844 If any access permissions are checked, each shall be checked individually, as described in the |
3845 Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 3, Definitions. If the process has |
3846 appropriate privileges, an implementation may indicate success for X_OK even if none of the
3847 execute file permission bits are set.

3848 **RETURN VALUE**

3849 If the requested access is permitted, *access*( ) succeeds and shall return 0; otherwise, –1 shall be
3850 returned and *errno* shall be set to indicate the error.

3851 **ERRORS**

3852 The *access*( ) function shall fail if:

3853 [EACCES] Permission bits of the file mode do not permit the requested access, or search |
3854 permission is denied on a component of the path prefix. |

3855 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* |
3856 argument. |

3857 [ENAMETOOLONG] |
3858 The length of the *path* argument exceeds {PATH_MAX} or a path name |
3859 component is longer than {NAME_MAX}. |

3860 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string. |

3861 [ENOTDIR] A component of the path prefix is not a directory. |

3862 [EROFS] Write access is requested for a file on a read-only file system. |

3863 The *access*( ) function may fail if:

3864 [EINVAL] The value of the *amode* argument is invalid. |

3865 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
3866 resolution of the *path* argument. |

3867 [ENAMETOOLONG] |
3868 As a result of encountering a symbolic link in resolution of the *path* argument,
3869 the length of the substituted path name string exceeded {PATH_MAX}. |

3870 [ETXTBSY] Write access is requested for a pure procedure (shared text) file that is being |
3871 executed. |

EXAMPLES

Testing for the Existence of a File

The following example tests whether a file named **myfile** exists in the **/tmp** directory.

```
#include <unistd.h>
...
int result;
const char *filename = "/tmp/myfile";

result = access (filename, F_OK);
```

APPLICATION USAGE

Additional values of *amode* other than the set defined in the description may be valid; for example, if a system has extended access controls.

RATIONALE

In early proposals, some inadequacies in the *access*( ) function led to the creation of an *eaccess*( ) function because:

1. Historical implementations of *access*( ) do not test file access correctly when the process' real user ID is superuser. In particular, they always return zero when testing execute permissions without regard to whether the file is executable.
2. The superuser has complete access to all files on a system. As a consequence, programs started by the superuser and switched to the effective user ID with lesser privileges cannot use *access*( ) to test their file access permissions.

However, the historical model of *eaccess*( ) does not resolve problem (1), so this volume of IEEE Std. 1003.1-200x now allows *access*( ) to behave in the desired way because several implementations have corrected the problem. It was also argued that problem (2) is more easily solved by using *open*( ), *chdir*( ), or one of the *exec* functions as appropriate and responding to the error, rather than creating a new function that would not be as reliable. Therefore, *eaccess*( ) is not included in this volume of IEEE Std. 1003.1-200x.

The sentence concerning appropriate privileges and execute permission bits reflects the two possibilities implemented by historical implementations when checking superuser access for X_OK.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *stat*( ), 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 4

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument *path* is changed from **char*** to **const char***.

The following change is incorporated for alignment with the FIPS requirements:

- In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path name component is larger that {NAME_MAX} is now defined as mandatory and marked as an extension.

Issue 4, Version 2

The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- It states that [ELOOP] is returned if too many symbolic links are encountered during path name resolution.
- A second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of path name resolution of a symbolic link.

Issue 6

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

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 [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.

NAME

acos, acosf, acosl — arc cosine function

SYNOPSIS

```
#include <math.h>

double acos(double x);
float acosf(float x);
long double acosl(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.

The *acos* family of functions shall compute the principal value of the arc cosine of *x*. The value of *x* should be in the range [–1,1].

An application wishing to check for error situations should set *errno* to 0 before calling *acos*( ). If *errno* is non-zero on return, or the value NaN is returned, an error has occurred.

RETURN VALUE

Upon successful completion, the *acos* family of functions shall return the arc cosine of *x*, in the XSI range [0,π] radians. If the value of *x* is not in the range [–1,1], and is not ±Inf or NaN, either 0.0 or NaN shall be returned and *errno* shall be set to [EDOM].

XSI If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM]. If *x* is ±Inf, either 0.0 shall be returned and *errno* shall be set to [EDOM], or NaN shall be returned and *errno* may be set to [EDOM].

ERRORS

The *acos* family of functions shall fail if:

XSI [EDOM] The value *x* is not ±Inf or NaN and is not in the range [–1,1].

The *acos* family of functions may fail if:

XSI [EDOM] The value *x* is ±Inf or NaN.

XSI No other errors shall occur.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cos( ), *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 4

Removed references to *matherr*( ).

The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the ISO C standard and to rationalize error handling in the mathematics functions.

The return value specified for [EDOM] is marked as an extension.

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 *acosf*( ) and *acosl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

NAME

acosh, acoshf, acoshl, asinh, asinfh, asinfl, atanh, atanhf, atanhl — inverse hyperbolic functions

SYNOPSIS

```
#include <math.h>

double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);
double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);
double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);
```

DESCRIPTION

The *acosh*( ), *asinh*( ), and *atanh*( ) functions shall compute the inverse hyperbolic cosine, sine, and tangent of their argument, respectively.

RETURN VALUE

The *acosh*( ), *asinh*( ), and *atanh*( ) functions shall return the inverse hyperbolic cosine, sine, and tangent of their argument, respectively.

The *acosh*( ), *acoshf*( ), and *acoshl*( ) functions shall return an implementation-defined value (NaN or equivalent if available) and set *errno* to [EDOM] when its argument is less than 1.0.

The *atanh*( ), *atanhf*( ), and *atanhl*( ) functions shall return an implementation-defined value (NaN or equivalent if available) and set *errno* to [EDOM] when its argument has absolute value greater than 1.0.

If *x* is NaN, the *asinh*( ), *acosh*( ), and *atanh*( ) functions shall return NaN and may set *errno* to [EDOM].

ERRORS

The *acosh*( ), *acoshf*( ), and *acoshl*( ) functions shall fail if:

[EDOM] The *x* argument is less than 1.0.

The *atanh*( ), *atanhf*( ), and *atanhl*( ) functions shall fail if:

[EDOM] The *x* argument has an absolute value greater than 1.0.

The *atanh*( ), *atanhf*( ), and *atanhl*( ) functions shall fail if:

[ERANGE] The *x* argument has an absolute value equal to 1.0

The *asinh*( ), *acosh*( ), and *atanh*( ) functions may fail if:

[EDOM] The value of *x* is NaN.

4016 **EXAMPLES**

4017 None.

4018 **APPLICATION USAGE**

4019 None.

4020 **RATIONALE**

4021 None.

4022 **FUTURE DIRECTIONS**

4023 None.

4024 **SEE ALSO**

4025 *cosh*( ), *sinh*( ), *tanh*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>** |

4026 **CHANGE HISTORY**

4027 First released in Issue 4, Version 2.

4028 **Issue 5**

4029 Moved from X/OPEN UNIX extension to BASE. |

4030 **Issue 6** |

4031 The *acoshf*( ), *acoshl*( ), *asinhf*( ), *asinhl*( ), *atanhf*( ), and *atanhl*( ) functions are added for alignment |

4032 with the ISO/IEC 9899:1999 standard. |

NAME

aio_cancel — cancel an asynchronous I/O request (**REALTIME**)

SYNOPSIS

AIO
```
#include <aio.h>

int aio_cancel(int fildes, struct aiocb *aiocbp);
```

DESCRIPTION

The *aio_cancel*( ) function shall attempt to cancel one or more asynchronous I/O requests currently outstanding against file descriptor *fildes*. The *aiocbp* argument points to the asynchronous I/O control block for a particular request to be canceled. If *aiocbp* is NULL, then all outstanding cancelable asynchronous I/O requests against *fildes* shall be canceled.

Normal asynchronous notification shall occur for asynchronous I/O operations that are successfully canceled. If there are requests that cannot be canceled, then the normal asynchronous completion process shall take place for those requests when they are completed.

For requested operations that are successfully canceled, the associated error status shall be set to [ECANCELED] and the return status shall be −1. For requested operations that are not successfully canceled, the *aiocbp* shall not be modified by *aio_cancel*( ).

If *aiocbp* is not NULL, then if *fildes* does not have the same value as the file descriptor with which the asynchronous operation was initiated, unspecified results occur.

Which operations are cancelable is implementation-defined.

RETURN VALUE

The *aio_cancel*( ) function shall return the value AIO_CANCELED to the calling process if the requested operation(s) were canceled. The value AIO_NOTCANCELED shall be returned if at least one of the requested operation(s) cannot be canceled because it is in progress. In this case, the state of the other operations, if any, referenced in the call to *aio_cancel*( ) is not indicated by the return value of *aio_cancel*( ). The application may determine the state of affairs for these operations by using *aio_error*( ). The value AIO_ALLDONE is returned if all of the operations have already completed. Otherwise, the function shall return −1 and set *errno* to indicate the error.

ERRORS

The *aio_cancel*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid file descriptor.

EXAMPLES

None.

APPLICATION USAGE

The *aio_cancel*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

4074 **SEE ALSO**

4075 *aio_read*( ), *aio_write*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<aio.h>**

4076 **CHANGE HISTORY**

4077 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

4078 **Issue 6**

4079 The [ENOSYS] error condition has been removed as stubs need not be provided if an
4080 implementation does not support the Asynchronous Input and Output option.

4081 The APPLICATION USAGE section is added.

NAME

aio_error — retrieve errors status for an asynchronous I/O operation (**REALTIME**)

SYNOPSIS

AIO
```
#include <aio.h>

int aio_error(const struct aiocb *aiocbp);
```

DESCRIPTION

The *aio_error*() function shall return the error status associated with the **aiocb** structure referenced by the *aiocbp* argument. The error status for an asynchronous I/O operation is the SIO *errno* value that would be set by the corresponding *read*(), *write*(), *fdatasync*(), or *fsync*() operation. If the operation has not yet completed, then the error status shall be equal to [EINPROGRESS].

RETURN VALUE

If the asynchronous I/O operation has completed successfully, then 0 shall be returned. If the asynchronous operation has completed unsuccessfully, then the error status, as described for SIO *read*(), *write*(), *fdatasync*(), and *fsync*(), shall be returned. If the asynchronous I/O operation has not yet completed, then [EINPROGRESS] shall be returned.

ERRORS

The *aio_error*() function may fail if:

[EINVAL] The *aiocbp* argument does not refer to an asynchronous operation whose return status has not yet been retrieved.

EXAMPLES

None.

APPLICATION USAGE

The *aio_error*() function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_cancel(), *aio_fsync*(), *aio_read*(), *aio_return*(), *aio_write*(), *close*(), *exec*, *exit*(), *fork*(), *lio_listio*(), *lseek*(), *read*(), the Base Definitions volume of IEEE Std. 1003.1-200x, **<aio.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.

4121 **NAME**

4122 aio_fsync — asynchronous file synchronization (**REALTIME**)

4123 **SYNOPSIS**

4124 AIO
```
#include <aio.h>
```

4125
```
int aio_fsync(int op, struct aiocb *aiocbp);
```

4126

4127 **DESCRIPTION**

4128 The *aio_fsync*( ) function asynchronously forces all I/O operations associated with the file
4129 indicated by the file descriptor *aio_fildes* member of the **aiocb** structure referenced by the *aiocbp*
4130 argument and queued at the time of the call to *aio_fsync*( ) to the synchronized I/O completion
4131 state. The function call shall return when the synchronization request has been initiated or
4132 queued to the file or device (even when the data cannot be synchronized immediately).

4133 If *op* is O_DSYNC, all currently queued I/O operations shall be completed as if by a call to
4134 *fdatasync*( ); that is, as defined for synchronized I/O data integrity completion. If *op* is O_SYNC,
4135 all currently queued I/O operations shall be completed as if by a call to *fsync*( ); that is, as
4136 defined for synchronized I/O file integrity completion. If the *aio_fsync*( ) function fails, or if the
4137 operation queued by *aio_fsync*( ) fails, then, as for *fsync*( ) and *fdatasync*( ), outstanding I/O
4138 operations are not guaranteed to have been completed.

4139 If *aio_fsync*( ) succeeds, then it is only the I/O that was queued at the time of the call to
4140 *aio_fsync*( ) that is guaranteed to be forced to the relevant completion state. The completion of
4141 subsequent I/O on the file descriptor is not guaranteed to be completed in a synchronized
4142 fashion.

4143 The *aiocbp* argument refers to an asynchronous I/O control block. The *aiocbp* value may be used
4144 as an argument to *aio_error*( ) and *aio_return*( ) in order to determine the error status and return
4145 status, respectively, of the asynchronous operation while it is proceeding. When the request is
4146 queued, the error status for the operation is [EINPROGRESS]. When all data has been |
4147 successfully transferred, the error status shall be reset to reflect the success or failure of the
4148 operation. If the operation does not complete successfully, the error status for the operation shall
4149 be set to indicate the error. The *aio_sigevent* member determines the asynchronous notification to
4150 occur as specified in Section 2.4.1 (on page 528) when all operations have achieved synchronized
4151 I/O completion. All other members of the structure referenced by *aiocbp* are ignored. If the
4152 control block referenced by *aiocbp* becomes an illegal address prior to asynchronous I/O
4153 completion, then the behavior is undefined.

4154 If the *aio_fsync*( ) function fails or the *aiocbp* indicates an error condition, data is not guaranteed
4155 to have been successfully transferred.

4156 **RETURN VALUE**

4157 The *aio_fsync*( ) function shall return the value 0 to the calling process if the I/O operation is
4158 successfully queued; otherwise, the function shall return the value –1 and set *errno* to indicate
4159 the error.

4160 **ERRORS**

4161 The *aio_fsync*( ) function shall fail if:

4162 [EAGAIN] The requested asynchronous operation was not queued due to temporary |
4163 resource limitations.

4164 [EBADF] The *aio_fildes* member of the **aiocb** structure referenced by the *aiocbp* argument |
4165 is not a valid file descriptor open for writing.

[EINVAL] This implementation does not support synchronized I/O for this file.

[EINVAL] A value of *op* other than O_DSYNC or O_SYNC was specified.

In the event that any of the queued I/O operations fail, *aio_fsync*() shall return the error condition defined for *read*() and *write*(). The error is returned in the error status for the asynchronous *fsync*() operation, which can be retrieved using *aio_error*().

EXAMPLES

None.

APPLICATION USAGE

The *aio_fsync*() function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fcntl(), *fdatasync*(), *fsync*(), *open*(), *read*(), *write*(), the Base Definitions volume of IEEE Std. 1003.1-200x, **<aio.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.

4189 **NAME**

4190 aio_read — asynchronous read from a file (**REALTIME**)

4191 **SYNOPSIS**

4192 AIO
```
#include <aio.h>
```

4193
```
int aio_read(struct aiocb *aiocbp);
```

4194

4195 **DESCRIPTION**

4196 The *aio_read*( ) function allows the calling process to read *aiocbp->aio_nbytes* from the file
4197 associated with *aiocbp->aio_fildes* into the buffer pointed to by *aiocbp->aio_buf*. The function call
4198 shall return when the read request has been initiated or queued to the file or device (even when
4199 the data cannot be delivered immediately).

4200 PIO If prioritized I/O is supported for this file, then the asynchronous operation is submitted at a
4201 priority equal to the scheduling priority of the process minus *aiocbp->aio_reqprio*.

4202 The *aiocbp* value may be used as an argument to *aio_error*( ) and *aio_return*( ) in order to
4203 determine the error status and return status, respectively, of the asynchronous operation while it
4204 is proceeding. If an error condition is encountered during queuing, the function call shall return
4205 without having initiated or queued the request. The requested operation takes place at the
4206 absolute position in the file as given by *aio_offset*, as if *lseek*( ) were called immediately prior to
4207 the operation with an *offset* equal to *aio_offset* and a *whence* equal to {SEEK_SET}. After a
4208 successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file
4209 is unspecified.

4210 The *aiocbp->aio_lio_opcode* field shall be ignored by *aio_read*( ).

4211 The *aiocbp* argument points to an **aiocb** structure. If the buffer pointed to by *aiocbp->aio_buf* or
4212 the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O
4213 completion, then the behavior is undefined.

4214 Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

4215 SIO If synchronized I/O is enabled on the file associated with *aiocbp->aio_fildes*, the behavior of this
4216 function shall be according to the definitions of synchronized I/O data integrity completion and
4217 synchronized I/O file integrity completion.

4218 For any system action that changes the process memory space while an asynchronous I/O is
4219 outstanding to the address range being changed, the result of that action is undefined.

4220 For regular files, no data transfer shall occur past the offset maximum established in the open
4221 file description associated with *aiocbp->aio_fildes*.

4222 **RETURN VALUE**

4223 The *aio_read*( ) function shall return the value zero to the calling process if the I/O operation is
4224 successfully queued; otherwise, the function shall return the value –1 and set *errno* to indicate
4225 the error.

4226 **ERRORS**

4227 The *aio_read*( ) function shall fail if:

4228 [EAGAIN] The requested asynchronous I/O operation was not queued due to system
4229 resource limitations.

4230 Each of the following conditions may be detected synchronously at the time of the call to
4231 *aio_read*( ), or asynchronously. If any of the conditions below are detected synchronously, the
4232 *aio_read*( ) function shall return –1 and set *errno* to the corresponding value. If any of the
4233 conditions below are detected asynchronously, the return status of the asynchronous operation

is set to –1, and the error status of the asynchronous operation is set to the corresponding value.

[EBADF] The *aiocbp->aio_fildes* argument is not a valid file descriptor open for reading.

[EINVAL] The file offset value implied by *aiocbp->aio_offset* would be invalid, *aiocbp->aio_reqprio* is not a valid value, or *aiocbp->aio_nbytes* is an invalid value.

In the case that the *aio_read*( ) successfully queues the I/O operation but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operation is one of the values normally returned by the *read*( ) function call. In addition, the error status of the asynchronous operation is set to one of the error statuses normally set by the *read*( ) function call, or one of the following values:

[EBADF] The *aiocbp->aio_fildes* argument is not a valid file descriptor open for reading.

[ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit *aio_cancel*( ) request.

[EINVAL] The file offset value implied by *aiocbp->aio_offset* would be invalid.

The following condition may be detected synchronously or asynchronously:

[EOVERFLOW] The file is a regular file, *aiobcp->aio_nbytes* is greater than 0, and the starting offset in *aiobcp->aio_offset* is before the end-of-file and is at or beyond the offset maximum in the open file description associated with *aiocbp->aio_fildes*.

EXAMPLES

None.

APPLICATION USAGE

The *aio_read*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_cancel( ), *aio_error*( ), *lio_listio*( ), *aio_return*( ), *aio_write*( ), *close*( ), *exec*, *exit*( ), *fork*( ), *lseek*( ), *read*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<aio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Large File Summit extensions are added.

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 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.

NAME

aio_return — retrieve return status of an asynchronous I/O operation (**REALTIME**)

SYNOPSIS

AIO

```
#include <aio.h>

ssize_t aio_return(struct aiocb *aiocbp);
```

DESCRIPTION

The *aio_return*( ) function shall return the return status associated with the **aiocb** structure referenced by the *aiocbp* argument. The return status for an asynchronous I/O operation is the value that would be returned by the corresponding *read*( ), *write*( ), or *fsync*( ) function call. If the error status for the operation is equal to [EINPROGRESS], then the return status for the operation is undefined. The *aio_return*( ) function may be called exactly once to retrieve the return status of a given asynchronous operation; thereafter, if the same **aiocb** structure is used in a call to *aio_return*( ) or *aio_error*( ), an error may be returned. When the **aiocb** structure referred to by *aiocbp* is used to submit another asynchronous operation, then *aio_return*( ) may be successfully used to retrieve the return status of that operation.

RETURN VALUE

If the asynchronous I/O operation has completed, then the return status, as described for *read*( ), *write*( ), and *fsync*( ), shall be returned. If the asynchronous I/O operation has not yet completed, the results of *aio_return*( ) are undefined.

ERRORS

The *aio_return*( ) function may fail if:

[EINVAL] The *aiocbp* argument does not refer to an asynchronous operation whose return status has not yet been retrieved.

EXAMPLES

None.

APPLICATION USAGE

The *aio_return*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_cancel( ), *aio_error*( ), *aio_fsync*( ), *aio_read*( ), *aio_write*( ), *close*( ), *exec*, *exit*( ), *fork*( ), *lio_listio*( ), *lseek*( ), *read*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<aio.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 [EINVAL] error condition is updated as a ‘‘may fail’’. This is for consistency with the DESCRIPTION.

NAME

aio_suspend — wait for an asynchronous I/O request (**REALTIME**)

SYNOPSIS

AIO
```
#include <aio.h>

int aio_suspend(const struct aiocb * const list[], int nent,
    const struct timespec *timeout);
```

DESCRIPTION

The *aio_suspend*( ) function shall suspend the calling thread until at least one of the asynchronous I/O operations referenced by the *list* argument has completed, until a signal interrupts the function, or, if *timeout* is not NULL, until the time interval specified by *timeout* has passed. If any of the **aiocb** structures in the list correspond to completed asynchronous I/O operations (that is, the error status for the operation is not equal to [EINPROGRESS]) at the time of the call, the function shall return without suspending the calling thread. The *list* argument is an array of pointers to asynchronous I/O control blocks. The *nent* argument indicates the number of elements in the array. Each **aiocb** structure pointed to has been used in initiating an asynchronous I/O request via *aio_read*( ), *aio_write*( ), or *lio_listio*( ). This array may contain NULL pointers, which are ignored. If this array contains pointers that refer to **aiocb** structures that have not been used in submitting asynchronous I/O, the effect is undefined.

If the time interval indicated in the **timespec** structure pointed to by *timeout* passes before any of the I/O operations referenced by *list* are completed, then *aio_suspend*( ) shall return with an error. MON If the Monotonic Clock option is supported, the clock that shall be used to measure this time interval shall be the CLOCK_MONOTONIC clock.

RETURN VALUE

If the *aio_suspend*( ) function returns after one or more asynchronous I/O operations have completed, the function shall return zero. Otherwise, the function shall return a value of –1 and set *errno* to indicate the error.

The application may determine which asynchronous I/O completed by scanning the associated error and return status using *aio_error*( ) and *aio_return*( ), respectively.

ERRORS

The *aio_suspend*( ) function shall fail if:

[EAGAIN] No asynchronous I/O indicated in the list referenced by *list* completed in the time interval indicated by *timeout*.

[EINTR] A signal interrupted the *aio_suspend*( ) function. Note that, since each asynchronous I/O operation may possibly provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited.

EXAMPLES

None.

APPLICATION USAGE

The *aio_suspend*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_read( ), *aio_write*( ), *lio_listio*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<aio.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.

4376 **NAME**

4377 aio_write — asynchronous write to a file (**REALTIME**)

4378 **SYNOPSIS**

4379 AIO
```
#include <aio.h>
```

4380
```
int aio_write(struct aiocb *aiocbp);
```

4381

4382 **DESCRIPTION**

4383 The *aio_write*( ) function allows the calling process to write *aiocbp->aio_nbytes* to the file |
4384 associated with *aiocbp->aio_fildes* from the buffer pointed to by *aiocbp->aio_buf*. The function call |
4385 shall return when the write request has been initiated or, at a minimum, queued to the file or |
4386 device.

4387 PIO If prioritized I/O is supported for this file, then the asynchronous operation shall be submitted
4388 at a priority equal to the scheduling priority of the process minus *aiocbp->aio_reqprio*. |

4389 The *aiocbp* may be used as an argument to *aio_error*( ) and *aio_return*( ) in order to determine the
4390 error status and return status, respectively, of the asynchronous operation while it is proceeding.

4391 The *aiocbp* argument points to an **aiocb** structure. If the buffer pointed to by *aiocbp->aio_buf* or |
4392 the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O
4393 completion, then the behavior is undefined.

4394 If O_APPEND is not set for the file descriptor *aio_fildes*, then the requested operation takes place
4395 at the absolute position in the file as given by *aio_offset*, as if *lseek*( ) were called immediately
4396 prior to the operation with an *offset* equal to *aio_offset* and a *whence* equal to {SEEK_SET}. If
4397 O_APPEND is set for the file descriptor, write operations append to the file in the same order as
4398 the calls were made. After a successful call to enqueue an asynchronous I/O operation, the value
4399 of the file offset for the file is unspecified.

4400 The *aiocbp->aio_lio_opcode* field shall be ignored by *aio_write*( ). |

4401 Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

4402 SIO If synchronized I/O is enabled on the file associated with *aiocbp->aio_fildes*, the behavior of this |
4403 function shall be according to the definitions of synchronized I/O data integrity completion, and |
4404 synchronized I/O file integrity completion.

4405 For any system action that changes the process memory space while an asynchronous I/O is
4406 outstanding to the address range being changed, the result of that action is undefined.

4407 For regular files, no data transfer shall occur past the offset maximum established in the open |
4408 file description associated with *aiocbp->aio_fildes*. |

4409 **RETURN VALUE**

4410 The *aio_write*( ) function shall return the value zero to the calling process if the I/O operation is
4411 successfully queued; otherwise, the function shall return the value –1 and set *errno* to indicate
4412 the error.

4413 **ERRORS**

4414 The *aio_write*( ) function shall fail if:

4415 [EAGAIN] The requested asynchronous I/O operation was not queued due to system |
4416 resource limitations.

4417 Each of the following conditions may be detected synchronously at the time of the call to
4418 *aio_write*( ), or asynchronously. If any of the conditions below are detected synchronously, the
4419 *aio_write*( ) function shall return –1 and set *errno* to the corresponding value. If any of the

conditions below are detected asynchronously, the return status of the asynchronous operation shall be set to –1, and the error status of the asynchronous operation is set to the corresponding value.

[EBADF] The *aiocbp->aio_fildes* argument is not a valid file descriptor open for writing.

[EINVAL] The file offset value implied by *aiocbp->aio_offset* would be invalid, *aiocbp->aio_reqprio* is not a valid value, or *aiocbp->aio_nbytes* is an invalid value.

In the case that the *aio_write*( ) successfully queues the I/O operation, the return status of the asynchronous operation shall be one of the values normally returned by the *write*( ) function call. If the operation is successfully queued but is subsequently canceled or encounters an error, the error status for the asynchronous operation contains one of the values normally set by the *write*( ) function call, or one of the following:

[EBADF] The *aiocbp->aio_fildes* argument is not a valid file descriptor open for writing.

[EINVAL] The file offset value implied by *aiocbp->aio_offset* would be invalid.

[ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit *aio_cancel*( ) request.

The following condition may be detected synchronously or asynchronously:

[EFBIG] The file is a regular file, *aiobcp->aio_nbytes* is greater than 0, and the starting offset in *aiobcp->aio_offset* is at or beyond the offset maximum in the open file description associated with *aiocbp->aio_fildes*.

EXAMPLES

None.

APPLICATION USAGE

The *aio_write*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_cancel( ), *aio_error*( ), *aio_read*( ), *aio_return*( ), *close*( ), *exec*, *exit*( ), *fork*( ), *lio_listio*( ), *lseek*( ), *write*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<aio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Large File Summit extensions are added.

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 following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, text is added to indicate that for regular files no data transfer occurs past the offset maximum established in the open file description associated with *aiocbp->aio_fildes*.

4463 • The [EFBIG] error is added as part of the large file support extensions.

4464 **NAME**

4465 alarm — schedule an alarm signal

4466 **SYNOPSIS**

4467 `#include <unistd.h>`

4468 `unsigned alarm(unsigned seconds);`

4469 **DESCRIPTION**

4470 The *alarm*( ) function shall cause the system to generate a SIGALRM signal for the process after
4471 the number of realtime seconds specified by *seconds* have elapsed. Processor scheduling delays
4472 may prevent the process from handling the signal as soon as it is generated.

4473 If *seconds* is 0, a pending alarm request, if any, is canceled.

4474 Alarm requests are not stacked; only one SIGALRM generation can be scheduled in this manner.
4475 If the SIGALRM signal has not yet been generated, the call shall result in rescheduling the time
4476 at which the SIGALRM signal is generated.

4477 Interactions between *alarm*( ) and any of *setitimer*( ), *ualarm*( ), or *usleep*( ) are unspecified.

4478 **RETURN VALUE**

4479 If there is a previous *alarm*( ) request with time remaining, *alarm*( ) shall return a non-zero value
4480 that is the number of seconds until the previous request would have generated a SIGALRM
4481 signal. Otherwise, *alarm*( ) shall return 0.

4482 **ERRORS**

4483 The *alarm*( ) function is always successful, and no return value is reserved to indicate an error.

4484 **EXAMPLES**

4485 None.

4486 **APPLICATION USAGE**

4487 The *fork*( ) function clears pending alarms in the child process. A new process image created by
4488 one of the *exec* functions inherits the time left to an alarm signal in the old process' image.

4489 Application writers should note that the type of the argument *seconds* and the return value of
4490 *alarm*( ) is **unsigned**. That means that a Strictly Conforming POSIX System Interfaces
4491 Application cannot pass a value greater than the minimum guaranteed value for {UINT_MAX},
4492 which the ISO C standard sets as 65 535, and any application passing a larger value is restricting
4493 its portability. A different type was considered, but historical implementations, including those
4494 with a 16-bit **int** type, consistently use either **unsigned** or **int**.

4495 Application writers should be aware of possible interactions when the same process uses both
4496 the *alarm*( ) and *sleep*( ) functions.

4497 **RATIONALE**

4498 Many historical implementations (including Version 7 and System V) allow an alarm to occur up
4499 to a second early. Other implementations allow alarms up to half a second or one clock tick
4500 early or do not allow them to occur early at all. The latter is considered most appropriate, since it
4501 gives the most predictable behavior, especially since the signal can always be delayed for an
4502 indefinite amount of time due to scheduling. Applications can thus choose the *seconds* argument
4503 as the minimum amount of time they wish to have elapse before the signal.

4504 The term *realtime* here and elsewhere (*sleep*( ), *times*( )) is intended to mean ''wall clock'' time as
4505 common English usage, and has nothing to do with ''realtime operating systems''. It is in
4506 contrast to *virtual time,* which could be misinterpreted if just *time* were used.

4507 In some implementations, including 4.3 BSD, very large values of the *seconds* argument are
4508 silently rounded down to an implementation-defined maximum value. This maximum is large

enough (on the order of several months) that the effect is not noticeable.

There were two possible choices for alarm generation in multi-threaded applications: generation for the calling thread or generation for the process. The first option would not have been particularly useful since the alarm state is maintained on a per-process basis and the alarm that is established by the last invocation of *alarm*( ) is the only one that would be active.

Furthermore, allowing generation of an asynchronous signal for a thread would have introduced an exception to the overall signal model. This requires a compelling reason in order to be justified.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *exec*, *fork*( ), *getitimer*( ), *pause*( ), *sigaction*( ), *sleep*( ), *ualarm*( ), *usleep*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<signal.h>**, **<unistd.h>**

CHANGE HISTORY

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

Issue 4

The **<unistd.h>** header is included in the SYNOPSIS section.

Issue 4, Version 2

The DESCRIPTION is updated to indicate that interactions with the *setitimer*( ), *ualarm*( ), and *usleep*( ) functions are unspecified.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to indicate that interactions with the *setitimer*( ), *ualarm*( ), and *usleep*( ) functions are unspecified.

NAME

asctime, asctime_r — convert date and time to a string

SYNOPSIS

```
#include <time.h>

char *asctime(const struct tm *timeptr);
```

TSF
```
char *asctime_r(const struct tm *restrict tm, char *restrict buf);
```

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 *asctime*( ) function shall convert the broken-down time in the structure pointed to by *timeptr* into a string in the form:

```
Sun Sep 16 01:03:52 1973\n\0
```

using the equivalent of the following algorithm:

```
char *asctime(const struct tm *timeptr)
{
    static char wday_name[7][3] = {
        "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
    };
    static char mon_name[12][3] = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
    };
    static char result[26];

    sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
        wday_name[timeptr->tm_wday],
        mon_name[timeptr->tm_mon],
        timeptr->tm_mday, timeptr->tm_hour,
        timeptr->tm_min, timeptr->tm_sec,
        1900 + timeptr->tm_year);
    return result;
}
```

The **tm** structure is defined in the **<time.h>** header.

CX 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 *asctime*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *asctime_r*( ) function shall convert the broken-down time in the structure pointed to by *tm* into a string (of the same form as that returned by *asctime*( )) that is placed in the user-supplied buffer pointed to by *buf* (which contains at least 26 bytes) and then return *buf*.

RETURN VALUE

Upon successful completion, *asctime*( ) shall return a pointer to the string.

TSF Upon successful completion, *asctime_r*( ) shall return a pointer to a character string containing the date and time. This string is pointed to by the argument *buf*. If the function is unsuccessful, it shall return NULL.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Values for the broken-down time structure can be obtained by calling *gmtime*( ) or *localtime*( ). This function is included for compatibility with older implementations, and does not support localized date and time formats. Applications should use *strftime*( ) to achieve maximum portability.

The *asctime_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

clock( ), *ctime*( ), *difftime*( ), *gmtime*( ), *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 4

The location of the **tm** structure is now defined.

The APPLICATION USAGE section is expanded to describe the time-handling functions generally and to refer users to *strftime*( ), which is a locale-dependent time-handling function.

The following change is incorporated for alignment with the ISO C standard:

- The type of argument *timeptr* is changed from **struct tm*** to **const struct tm***.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

The *asctime_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *asctime*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *asctime_r*( ) function is marked as part of the Thread-Safe Functions option.

Extensions beyond the ISO C standard are now marked.

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.

The DESCRIPTION of *asctime_r*( ) is updated to describe the format of the string returned.

The **restrict** keyword is added to the *asctime_r*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

4622 **NAME**

4623 asin, asinf, asinl — arc sine function

4624 **SYNOPSIS**

4625
```
#include <math.h>
```

4626
```
double asin(double x);
```
4627
```
float asinf(float x);
```
4628
```
long double asinl(long double x);
```

4629 **DESCRIPTION**

4630 CX The functionality described on this reference page is aligned with the ISO C standard. Any
4631 conflict between the requirements described here and the ISO C standard is unintentional. This
4632 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

4633 The *asin*( ), *asinf*( ), and *asinl*( ) functions shall compute the principal value of the arc sine of *x*.
4634 The value of *x* should be in the range [–1,1].

4635 An application wishing to check for error situations should set *errno* to 0, then call *asin*( ). If *errno*
4636 is non-zero on return, or the return value is NaN, an error has occurred.

4637 **RETURN VALUE**

4638 Upon successful completion, the *asin*( ), *asinf*( ), and *asinl*( ) functions shall return the arc sine of
4639 XSI *x*, in the range [–π/2,π/2] radians. If the value of *x* is not in the range [–1,1], and is not ±Inf or
4640 NaN, either 0.0 or NaN shall be returned and *errno* shall be set to [EDOM].

4641 XSI If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

4642 If *x* is ±Inf, either 0.0 shall be returned and *errno* set to [EDOM], or NaN shall be returned and
4643 *errno* may be set to [EDOM].

4644 If the result underflows, 0.0 shallbe returned and *errno* may be set to [ERANGE].

4645 **ERRORS**

4646 The *asin*( ), *asinf*( ), and *asinl*( ) functions shall fail if:

4647 XSI [EDOM] The value *x* is not ±Inf or NaN and is not in the range [–1,1].

4648 The *asin*( ), *asinf*( ), and *asinl*( ) functions may fail if:

4649 XSI [EDOM] The value of *x* is ±Inf or NaN.

4650 [ERANGE] The result underflows

4651 XSI No other errors shall occur.

4652 **EXAMPLES**

4653 None.

4654 **APPLICATION USAGE**

4655 None.

4656 **RATIONALE**

4657 None.

4658 **FUTURE DIRECTIONS**

4659 None.

4660 **SEE ALSO**

4661 *isnan*( ), *sin*( ), 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 4

References to *matherr*( ) are removed.

The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the ISO C standard and to rationalize error handling in the mathematics functions.

The return value specified for [EDOM] is marked as an extension.

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.

4674 **NAME**

4675 asinh — hyperbolic arc sine

4676 **SYNOPSIS**

4677 XSI
```
#include <math.h>
```

4678
```
double asinh(double x);
```

4679

4680 **DESCRIPTION**

4681 Refer to *acosh*( ).

NAME

assert — insert program diagnostics

SYNOPSIS

```
#include <assert.h>

void assert(scalar expression);
```

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 *assert*( ) macro inserts diagnostics into programs; it expands to a **void** expression. When it is executed, if *expression* (which shall have a **scalar** type) is false (that is, compares equal to 0), *assert*( ) shall write information about the particular call that failed on *stderr* and shall call *abort*( ).

The information written about the call that failed shall include the text of the argument, the name of the source file, the source file line number, and the name of the enclosing function, the latter are, respectively, the values of the preprocessing macros __FILE__ and __LINE__ and of the identifier __func__.

Forcing a definition of the name NDEBUG, either from the compiler command line or with the preprocessor control statement **#define** NDEBUG ahead of the **#include <assert.h>** statement, shall stop assertions from being compiled into the program.

RETURN VALUE

The *assert*( ) macro shall return no value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

abort( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<assert.h>**, *stderr*

CHANGE HISTORY

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

Issue 4

The APPLICATION USAGE section is merged into the DESCRIPTION.

Issue 6

The prototype for the *expression* argument to *assert*( ) is changed from **int** to **scalar** for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION of *assert*( ) is updated for alignment with the ISO / IEC 9899: 1999 standard.

4723 **NAME**

4724 atan, atanf, atanl — arc tangent function

4725 **SYNOPSIS**

4726
```
#include <math.h>
```

4727
```
double atan(double x);
```
4728
```
float atanf(float x);
```
4729
```
long double atanl(long double x);
```

4730 **DESCRIPTION**

4731 CX The functionality described on this reference page is aligned with the ISO C standard. Any
4732 conflict between the requirements described here and the ISO C standard is unintentional. This
4733 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

4734 The *atan*( ), *atanf*( ), and *atanl*( ) functions shall compute the principal value of the arc tangent of
4735 *x*.

4736 An application wishing to check for error situations should set *errno* to 0 before calling *atan*( ). If
4737 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

4738 **RETURN VALUE**

4739 Upon successful completion, the *atan*( ), *atanf*( ), and *atanl*( ) functions shall return the arc
4740 tangent of *x* in the range $[-\pi/2,\pi/2]$ radians.

4741 XSI If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

4742 If the result underflows, 0.0 shall be returned and *errno* may be set to [ERANGE].

4743 **ERRORS**

4744 The *atan*( ), *atanf*( ), and *atanl*( ) functions may fail if:

4745 XSI [EDOM] The value of *x* is NaN.

4746 [ERANGE] The result underflows

4747 XSI No other errors shall occur.

4748 **EXAMPLES**

4749 None.

4750 **APPLICATION USAGE**

4751 None.

4752 **RATIONALE**

4753 None.

4754 **FUTURE DIRECTIONS**

4755 None.

4756 **SEE ALSO**

4757 *atan2*( ), *isnan*( ), *tan*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

4758 **CHANGE HISTORY**

4759 First released in Issue 1. Derived from Issue 1 of the SVID.

4760 **Issue 4**

4761 References to *matherr*( ) are removed.

4762 The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the
4763 ISO C standard and to rationalize error handling in the mathematics functions.

The return value specified for [EDOM] is marked as an extension.

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.

4770 **NAME**

4771 atan2 — arc tangent function

4772 **SYNOPSIS**

4773 ```
#include <math.h>
```

4774 ```
double atan2(double y, double x);
```
4775 ```
float atan2f(float y, float x);
```
4776 ```
long double atan2l(long double y, long double x);
```

4777 **DESCRIPTION**

4778 CX The functionality described on this reference page is aligned with the ISO C standard. Any
4779 conflict between the requirements described here and the ISO C standard is unintentional. This
4780 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

4781 The *atan2*( ), *atan2f*( ), and *atan2l*( ) functions shall compute the principal value of the arc tangent
4782 of *y* /*x*, using the signs of both arguments to determine the quadrant of the return value.

4783 An application wishing to check for error situations should set *errno* to 0 before calling *atan2*( ).
4784 If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

4785 **RETURN VALUE**

4786 Upon successful completion, the *atan2*( ), *atan2f*( ), and *atan2l*( ) functions shall return the arc
4787 tangent of *y* /*x* in the range [–π,π] radians. If both arguments are 0.0, an implementation-defined
4788 value is returned and *errno* may be set to [EDOM].

4789 XSI If *x* or *y* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

4790 If the result underflows, 0.0 shall be returned and *errno* may be set to [ERANGE].

4791 **ERRORS**

4792 The *atan2*( ), *atan2f*( ), and *atan2l*( ) functions may fail if:

4793 XSI [EDOM] Both arguments are 0.0 or one or more of the arguments is NaN.

4794 [ERANGE] The result underflows

4795 XSI No other errors shall occur.

4796 **EXAMPLES**

4797 None.

4798 **APPLICATION USAGE**

4799 None.

4800 **RATIONALE**

4801 None.

4802 **FUTURE DIRECTIONS**

4803 None.

4804 **SEE ALSO**

4805 *atan*( ), *isnan*( ), *tan*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

4806 **CHANGE HISTORY**

4807 First released in Issue 1. Derived from Issue 1 of the SVID.

4808 **Issue 4**

4809 References to *matherr*( ) are removed.

4810 The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the
4811 ISO C standard and to rationalize error handling in the mathematics functions.

The return value specified for [EDOM] is marked as an extension.

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.

4816 **NAME**

4817 atanh — hyperbolic arc tangent

4818 **SYNOPSIS**

4819 XSI
```
#include <math.h>
```

4820
```
double atanh(double x);
```

4821

4822 **DESCRIPTION**

4823 Refer to *acosh*( ).

NAME

atexit — register a function to run at process termination

SYNOPSIS

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

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 *atexit*( ) function registers the function pointed to by *func*, to be called without arguments at normal program termination. At normal program termination, all functions registered by the CX *atexit*( ) function shall be called, in the reverse order of their registration. Normal termination occurs either by a call to *exit*( ) or a return from *main*( ).

At least 32 functions can be registered with *atexit*( ).

CX After a successful call to any of the *exec* functions, any functions previously registered by *atexit*( ) shall no longer be registered.

RETURN VALUE

Upon successful completion, *atexit*( ) shall return 0; otherwise, it shall return a non-zero value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The functions registered by a call to *atexit*( ) must return to ensure that all registered functions are called.

The application should call *sysconf*( ) to obtain the value of {ATEXIT_MAX}, the number of functions that can be registered. There is no way for an application to tell how many functions have already been registered with *atexit*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exit( ), *sysconf*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

Issue 4, Version 2

The APPLICATION USAGE section is updated to indicate how an application can determine the setting of {ATEXIT_MAX}, which is a constant added for X/OPEN UNIX conformance.

Issue 6

Extensions beyond the ISO C standard are now marked.

4865 **NAME**

4866 atof — convert a string to double-precision number

4867 **SYNOPSIS**

4868 ```
#include <stdlib.h>
```

4869 ```
double atof(const char *str);
```

4870 **DESCRIPTION**

4871 CX The functionality described on this reference page is aligned with the ISO C standard. Any
4872 conflict between the requirements described here and the ISO C standard is unintentional. This
4873 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

4874 The call *atof*(*str*) shall be equivalent to:

4875 ```
strtod(str,(char **)NULL),
```

4876 except that the handling of errors may differ. If the value cannot be represented, the behavior is
4877 undefined.

4878 **RETURN VALUE**

4879 The *atof*( ) function shall return the converted value if the value can be represented.

4880 **ERRORS**

4881 No errors are defined.

4882 **EXAMPLES**

4883 None.

4884 **APPLICATION USAGE**

4885 The *atof*( ) function is subsumed by *strtod*( ) but is retained because it is used extensively in
4886 existing code. If the number is not known to be in range, *strtod*( ) should be used because *atof*( ) is
4887 not required to perform any error checking.

4888 **RATIONALE**

4889 None.

4890 **FUTURE DIRECTIONS**

4891 None.

4892 **SEE ALSO**

4893 *strtod*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>** |

4894 **CHANGE HISTORY**

4895 First released in Issue 1. Derived from Issue 1 of the SVID. |

4896 **Issue 4**

4897 Reference to how *str* is converted is removed from the DESCRIPTION.

4898 The APPLICATION USAGE section is added.

4899 The following change is incorporated for alignment with the ISO C standard:

4900 - The type of argument *str* is changed from **char*** to **const char***.
```

NAME

atoi — convert a string to an integer

SYNOPSIS

```
#include <stdlib.h>

int atoi(const char *str);
```

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 call *atoi*(*str*) shall be equivalent to:

```
(int) strtol(str, (char **)NULL, 10)
```

except that the handling of errors may differ. If the value cannot be represented, the behavior is undefined.

RETURN VALUE

The *atoi*( ) function shall return the converted value if the value can be represented.

ERRORS

No errors are defined.

EXAMPLES

Converting an Argument

The following example checks for proper usage of the program. If there is an argument and the decimal conversion of this argument (obtained using *atoi*( )) is greater than 0, then the program has a valid number of minutes to wait for an event.

```
#include <stdlib.h>
#include <stdio.h>
...
int minutes_to_event;
...
if (argc < 2 || ((minutes_to_event = atoi (argv[1]))) <= 0) {
   fprintf(stderr, "Usage: %s minutes\n", argv[0]); exit(1);
}
...
```

APPLICATION USAGE

The *atoi*( ) function is subsumed by *strtol*( ) but is retained because it is used extensively in existing code. If the number is not known to be in range, *strtol*( ) should be used because *atoi*( ) is not required to perform any error checking.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strtol( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>**

4942 **CHANGE HISTORY**

4943 First released in Issue 1. Derived from Issue 1 of the SVID.

4944 **Issue 4**

4945 Reference to how *str* is converted is removed from the DESCRIPTION.

4946 The APPLICATION USAGE section is added.

4947 The following change is incorporated for alignment with the ISO C standard:

4948 • The type of argument *str* is changed from **char*** to **const char***.

NAME

atol, atoll — convert a string to a long integer

SYNOPSIS

```
#include <stdlib.h>

long atol(const char *str);
long long atoll(const char *nptr);
```

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 call *atol*(*str*) shall be equivalent to:

```
strtol(str, (char **)NULL, 10)
```

The call *atoll*(*str*) shall be equivalent to:

```
strtoll(nptr, (char **)NULL, 10)
```

except that the handling of errors may differ. If the value cannot be represented, the behavior is undefined.

RETURN VALUE

These functions shall return the converted value if the value can be represented.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *atol*( ) function is subsumed by *strtol*( ) but is retained because it is used extensively in existing code. If the number is not known to be in range, *strtol*( ) should be used because *atol*( ) is not required to perform any error checking.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strtol( ), 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 4

Reference to how *str* is converted is removed from the DESCRIPTION.

The APPLICATION USAGE section is added.

The following changes are incorporated for alignment with the ISO C standard:

- The type of argument *str* is changed from **char*** to **const char***.
- The return type of the function is expanded to **long**.

Issue 6

The *atoll*( ) function is added for alignment with the ISO / IEC 9899: 1999 standard.

NAME

basename — return the last component of a path name

SYNOPSIS

XSI
```
#include <libgen.h>

char *basename(char *path);
```

DESCRIPTION

The *basename*( ) function shall take the path name pointed to by *path* and return a pointer to the final component of the path name, deleting any trailing `'/'` characters.

If the string consists entirely of the `'/'` character, *basename*( ) shall return a pointer to the string `"/"`. If the string is exactly `"//"`, it is implementation-defined whether `'/'` or `"//"` is returned.

If *path* is a null pointer or points to an empty string, *basename*( ) shall return a pointer to the string `"."`.

The *basename*( ) function may modify the string pointed to by *path*, and may return a pointer to static storage that may then be overwritten by a subsequent call to *basename*( ).

The *basename*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *basename*( ) function shall return a pointer to the final component of *path*.

ERRORS

No errors are defined.

EXAMPLES

Using basename( )

The following program fragment returns a pointer to the value *lib*, which is the base name of **/usr/lib**.

```
#include <libgen.h>
...
char *name = "/usr/lib";
char *base;

base = basename(name);
...
```

Sample Input and Output Strings for basename( )

In the following table, the input string is the value pointed to by *path*, and the output string is the return value of the *basename*( ) function.

Input String	Output String
`"/usr/lib"`	`"lib"`
`"/usr/"`	`"usr"`
`"/"`	`"/"`

5030 **APPLICATION USAGE**

5031 None.

5032 **RATIONALE**

5033 None.

5034 **FUTURE DIRECTIONS**

5035 None.

5036 **SEE ALSO**

5037 *dirname*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<libgen.h>**, the Shell and |
5038 Utilities volume of IEEE Std. 1003.1-200x, *basename* |

5039 **CHANGE HISTORY**

5040 First released in Issue 4, Version 2.

5041 **Issue 5**

5042 Moved from X/OPEN UNIX extension to BASE.

5043 Normative text previously in the APPLICATION USAGE section is moved to the
5044 DESCRIPTION.

5045 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

5046 **Issue 6**

5047 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

NAME

bcmp — memory operations (**LEGACY**)

SYNOPSIS

XSI
```
#include <strings.h>

int bcmp(const void *s1, const void *s2, size_t n);
```

DESCRIPTION

The *bcmp*( ) function shall compare the first *n* bytes of the area pointed to by *s1* with the area pointed to by *s2*.

RETURN VALUE

The *bcmp*( ) function shall return 0 if *s1* and *s2* are identical; otherwise, it shall return non-zero. Both areas are assumed to be *n* bytes long. If the value of *n* is 0, *bcmp*( ) shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

memcmp( ) is preferred over this function.

For maximum portability, it is recommended to replace the function call to *bcmp*( ) as follows:

```
#define bcmp(b1,b2,len) memcmp((b1), (b2), (size_t)(len))
```

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

memcmp( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<strings.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

This function is marked LEGACY.

NAME

bcopy — memory operations (**LEGACY**)

SYNOPSIS

XSI
```
#include <strings.h>

void bcopy(const void *s1, void *s2, size_t n);
```

DESCRIPTION

The *bcopy*( ) function shall copy *n* bytes from the area pointed to by *s1* to the area pointed to by *s2*.

The bytes are copied correctly even if the area pointed to by *s1* overlaps the area pointed to by *s2*.

RETURN VALUE

The *bcopy*( ) function shall return no value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

memmove( ) is preferred over this function.

The following are approximately equivalent (note the order of the arguments):

```
bcopy(s1,s2,n) ~= memmove(s2,s1,n)
```

For maximum portability, it is recommended to replace the function call to *bcopy*( ) as follows:

```
#define bcopy(b1,b2,len) (memmove((b2), (b1), (len)), (void) 0)
```

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

memmove( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<strings.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

This function is marked LEGACY.

NAME

bind — bind a name to a socket

SYNOPSIS

```
#include <sys/socket.h>

int bind(int socket, const struct sockaddr *address,
    socklen_t address_len);
```

DESCRIPTION

The *bind*( ) function shall assign a local socket address *address* to a socket identified by descriptor *socket* that has no local socket address assigned. Sockets created with the *socket*( ) function are initially unnamed; they are identified only by their address family.

The *bind*( ) function takes the following arguments:

socket Specifies the file descriptor of the socket to be bound.

address 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.

address_len Specifies the length of the **sockaddr** structure pointed to by the *address* argument.

The socket specified by *socket* may require the process to have appropriate privileges to use the *bind*( ) function.

RETURN VALUE

Upon successful completion, *bind*( ) shall return 0; otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *bind*( ) function shall fail if:

[EADDRINUSE]
The specified address is already in use.

[EADDRNOTAVAIL]
The specified address is not available from the local machine.

[EAFNOSUPPORT]
The specified address is not a valid address for the address family of the specified socket.

[EBADF] The *socket* argument is not a valid file descriptor.

[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.

[ENOTSOCK] The *socket* argument does not refer to a socket.

[EOPNOTSUPP] The socket type of the specified socket does not support binding to an address.

If the address family of the socket is AF_UNIX, then *bind*( ) shall fail if:

[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.

[EDESTADDRREQ] or [EISDIR]
The *address* argument is a null pointer.

[EIO] An I/O error occurred.

[ELOOP] A loop exists in symbolic links encountered during resolution of the path name in *address*.

[ENAMETOOLONG]
A component of a path name exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.

[ENOENT] A component of the path name does not name an existing file or the path name is an empty string.

[ENOTDIR] A component of the path prefix of the path name in *address* is not a directory.

[EROFS] The name would reside on a read-only file system.

The *bind*( ) function may fail if:

[EACCES] The specified address is protected and the current user does not have permission to bind to it.

[EINVAL] The *address_len* argument is not a valid length for the address family.

[EISCONN] The socket is already connected.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the path name in *address*.

[ENAMETOOLONG]
Path name resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

[ENOBUFS] Insufficient resources were available to complete the call.

EXAMPLES

None.

APPLICATION USAGE

An 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.

NAME

bsd_signal — simplified signal facilities

SYNOPSIS

OB

```
#include <signal.h>

void (*bsd_signal(int sig, void (*func)(int)))(int);
```

DESCRIPTION

The *bsd_signal*( ) function provides a partially compatible interface for programs written to historical system interfaces (see APPLICATION USAGE).

The function call *bsd_signal*(*sig*, *func*) has an effect as if implemented as:

```
void (*bsd_signal(int sig, void (*func)(int)))(int)
{
    struct sigaction act, oact;

    act.sa_handler = func;
    act.sa_flags = SA_RESTART;
    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask, sig);
    if (sigaction(sig, &act, &oact) == -1)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

The handler function should be declared:

```
void handler(int sig);
```

where *sig* is the signal number. The behavior is undefined if *func* is a function that takes more than one argument, or an argument of a different type.

RETURN VALUE

Upon successful completion, *bsd_signal*( ) shall return the previous action for *sig*. Otherwise, SIG_ERR shall be returned and *errno* shall be set to indicate the error.

ERRORS

Refer to *sigaction*( ).

EXAMPLES

None.

APPLICATION USAGE

This function is a direct replacement for the BSD *signal*( ) function for simple applications that are installing a single-argument signal handler function. If a BSD signal handler function is being installed that expects more than one argument, the application has to be modified to use *sigaction*( ). The *bsd_signal*( ) function differs from *signal*( ) in that the SA_RESTART flag is set and the SA_RESETHAND is clear when *bsd_signal*( ) is used. The state of these flags is not specified for *signal*( ).

It is recommended that new applications use the *sigaction*( ) function.

RATIONALE

None.

5234 **FUTURE DIRECTIONS**

5235 None.

5236 **SEE ALSO**

5237 *sigaction*( ), *sigaddset*( ), *sigemptyset*( ), *signal*( ), the Base Definitions volume of |
5238 IEEE Std. 1003.1-200x, **<signal.h>** |

5239 **CHANGE HISTORY**

5240 First released in Issue 4, Version 2.

5241 **Issue 5**

5242 Moved from X/OPEN UNIX extension to BASE. |

5243 **Issue 6** |

5244 This function is marked obsolescent. |

NAME

bsearch — binary search a sorted table

SYNOPSIS

```
#include <stdlib.h>

void *bsearch(const void *key, const void *base, size_t nel,
    size_t width, int (*compar)(const void *, const void *));
```

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 *bsearch*( ) function searches an array of *nel* objects, the initial element of which is pointed to by *base*, for an element that matches the object pointed to by *key*. The size of each element in the array is specified by *width*.

The comparison function pointed to by *compar* is called with two arguments that point to the *key* object and to an array element, in that order.

The application shall ensure that the function returns an integer less than, equal to, or greater than 0 if the *key* object is considered, respectively, to be less than, to match, or to be greater than the array element. The application shall ensure that the array consists of all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the *key* object, in that order.

RETURN VALUE

The *bsearch*( ) function shall return a pointer to a matching member of the array, or a null pointer if no match is found. If two or more members compare equal, which member is returned is unspecified.

ERRORS

No errors are defined.

EXAMPLES

The example below searches a table containing pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

The code fragment below reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message.

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

#define TABSIZE    1000

struct node {                   /* These are stored in the table. */
    char *string;
    int length;
};
struct node table[TABSIZE];     /* Table to be searched. */
    .
    .
    .
{
    struct node *node_ptr, node;
    /* routine to compare 2 nodes */
```

```
        int node_compare(const void *, const void *);
        char str_space[20];   /* Space to read string into. */
        .
        .
        .
        node.string = str_space;
        while (scanf("%s", node.string) != EOF) {
            node_ptr = (struct node *)bsearch((void *)(&node),
                   (void *)table, TABSIZE,
                   sizeof(struct node), node_compare);
            if (node_ptr != NULL) {
                (void)printf("string = %20s, length = %d\n",
                    node_ptr->string, node_ptr->length);
            } else {
                (void)printf("not found: %s\n", node.string);
            }
        }
    }
    /*
        This routine compares two nodes based on an
        alphabetical ordering of the string field.
    */
    int
    node_compare(const void *node1, const void *node2)
    {
        return strcoll(((const struct node *)node1)->string,
            ((const struct node *)node2)->string);
    }
```

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

None.

FUTURE DIRECTIONS

None.

SEE ALSO

hcreate( ), *lsearch*( ), *qsort*( ), *tsearch*( ), 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 4

Text indicating the need for various casts is removed from the APPLICATION USAGE section.

The code in the EXAMPLES section is changed to use *strcoll*( ) instead of *strcmp*( ) in *node_compare*( ).

The return value and the contents of the array are now requirements on the application.

The DESCRIPTION is changed to specify the order of arguments.

The following changes are incorporated for alignment with the ISO C standard:

- The type of arguments *key* and *base*, and the type of arguments to *compar*, are changed from **void*** to **const void***.
- The requirement that the table be sorted according to *compar* is removed from the DESCRIPTION.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

5347 **NAME**

5348 btowc — single-byte to wide-character conversion

5349 **SYNOPSIS**

5350 ```
#include <stdio.h>
```

5351 ```
#include <wchar.h>
```

5352 ```
wint_t btowc(int c);
```

5353 **DESCRIPTION**

5354 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5355 conflict between the requirements described here and the ISO C standard is unintentional. This
5356 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

5357 The *btowc*( ) function shall determine whether *c* constitutes a valid (one-byte) character in the
5358 initial shift state.

5359 The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

5360 **RETURN VALUE**

5361 The *btowc*( ) function shall return WEOF if *c* has the value EOF or if **(unsigned char)** *c* does not
5362 constitute a valid (one-byte) character in the initial shift state. Otherwise, it shall return the
5363 wide-character representation of that character.

5364 **ERRORS**

5365 No errors are defined.

5366 **EXAMPLES**

5367 None.

5368 **APPLICATION USAGE**

5369 None.

5370 **RATIONALE**

5371 None.

5372 **FUTURE DIRECTIONS**

5373 None.

5374 **SEE ALSO**

5375 *wctob*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>** |

5376 **CHANGE HISTORY**

5377 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
5378 (E).
```

NAME

bzero — memory operations (**LEGACY**)

SYNOPSIS

XSI
```
#include <strings.h>

void bzero(void *s, size_t n);
```

DESCRIPTION

The *bzero*( ) function shall place *n* zero-valued bytes in the area pointed to by *s*.

RETURN VALUE

The *bzero*( ) function shall return no value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

memset( ) is preferred over this function.

For maximum portability, it is recommended to replace the function call to *bzero*( ) as follows:

```
#define bzero(b,len) (memset((b), '\0', (len)), (void) 0)
```

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

memset( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<strings.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

This function is marked LEGACY.

NAME

cabs, cabsf, cabsl — return a complex absolute value

SYNOPSIS

```
#include <complex.h>

double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(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 absolute value (also called norm, modulus, or magnitude) of *z*.

RETURN VALUE

These functions shall return the complex absolute value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

cacos, cacosf, cacosl — complex arc cosine functions

SYNOPSIS

```
#include <complex.h>

double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(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 arc cosine of *z*, with branch cuts outside the interval [–1, +1] along the real axis.

RETURN VALUE

These functions shall return the complex arc cosine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [0, π] along the real axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ccos( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

cacosh, cacoshf, cacoshl — complex arc hyperbolic cosine functions

SYNOPSIS

```
#include <complex.h>

double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(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 arc hyperbolic cosine of *z*, with a branch cut at values less than 1 along the real axis.

RETURN VALUE

These functions shall return the complex arc hyperbolic cosine value, in the range of a half-strip of non-negative values along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ccosh( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

calloc — a memory allocator

SYNOPSIS

```
#include <stdlib.h>

void *calloc(size_t nelem, size_t elsize);
```

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 *calloc*( ) function allocates unused space for an array of *nelem* elements each of whose size in bytes is *elsize*. The space is initialized to all bits 0.

The order and contiguity of storage allocated by successive calls to *calloc*( ) is unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object or an array of such objects 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 is returned. If the size of the space requested is 0, the behavior is implementation-defined; the value returned shall be either a null pointer or a unique pointer.

RETURN VALUE

Upon successful completion with both *nelem* and *elsize* non-zero, *calloc*( ) shall return a pointer to the allocated space. If either *nelem* or *elsize* is 0, then either a null pointer or a unique pointer value that can be successfully passed to *free*( ) shall be returned. Otherwise, it shall return a null pointer CX and set *errno* to indicate the error.

ERRORS

The *calloc*( ) function shall fail if:

CX [ENOMEM] Insufficient memory is available.

EXAMPLES

None.

APPLICATION USAGE

There is now no requirement for the implementation to support the inclusion of **<malloc.h>**.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

free( ), *malloc*( ), *realloc*( ), 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 4

The setting of *errno* and the [ENOMEM] error are marked as extensions.

The APPLICATION USAGE section is changed to record that **<malloc.h>** need no longer be supported on XSI-conformant systems.

The following changes are incorporated in this issue for alignment with the ISO C standard:

- The DESCRIPTION is updated to indicate:
  - — The order and contiguity of storage allocated by successive calls to this function is unspecified.
  - — Each allocation yields a pointer to an object disjoint from any other object.
  - — The returned pointer points to the lowest byte address of the allocation.
  - — The behavior if space is requested of zero size.
- The RETURN VALUE section is updated to indicate what is returned if either *nelem* or *elsize* is 0.

Issue 6

Extensions beyond the ISO C standard are now marked.

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.

NAME

carg, cargf, cargl — complex argument functions

SYNOPSIS

```
#include <complex.h>

double carg(double complex z);
float cargf(float complex z);
long double cargl(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 argument (also called phase angle) of *z*, with a branch cut along the negative real axis.

RETURN VALUE

These functions shall return the value of the argument in the interval [–π, +π].

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cimag( ), *conj*( ), *cproj*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

casin, casinf, casinl — complex arc sine functions

SYNOPSIS

```
#include <complex.h>

double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(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 arc sine of *z*, with branch cuts outside the interval [–1, +1] along the real axis.

RETURN VALUE

These functions shall return the complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [$-\pi/2$, $+\pi/2$] along the real axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

csin( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

casinh, casinhf, casinhl — complex arc hyperbolic sine functions

SYNOPSIS

```
#include <complex.h>

double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(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 arc hyperbolic sine of *z*, with branch cuts outside the interval [–*i*, +*i*] along the imaginary axis.

RETURN VALUE

These functions shall return the complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

csinh( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

catan, catanf, catanl — complex arc tangent functions

SYNOPSIS

```
#include <complex.h>

double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(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 arc tangent of *z*, with branch cuts outside the interval [–*i*, +*i*] along the imaginary axis.

RETURN VALUE

These functions shall return the complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [$-\pi/2$, $+\pi/2$] along the real axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ctan( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

catanh, catanhf, catanhl — complex arc hyperbolic tangent functions

SYNOPSIS

```
#include <complex.h>

double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(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 arc hyperbolic tangent of z, with branch cuts outside the interval [–1, +1] along the real axis.

RETURN VALUE

These functions shall return the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ctanh( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

5708 **NAME**

5709 catclose — close a message catalog descriptor

5710 **SYNOPSIS**

5711 XSI
```
#include <nl_types.h>
```

5712
```
int catclose(nl_catd catd);
```

5713

5714 **DESCRIPTION**

5715 The *catclose*( ) function shall close the message catalog identified by *catd*. If a file descriptor is
5716 used to implement the type **nl_catd**, that file descriptor shall be closed.

5717 **RETURN VALUE**

5718 Upon successful completion, *catclose*( ) shall return 0; otherwise, –1 shall be returned, and *errno*
5719 set to indicate the error.

5720 **ERRORS**

5721 The *catclose*( ) function may fail if:

5722 [EBADF] The catalog descriptor is not valid. |

5723 [EINTR] The *catclose*( ) function was interrupted by a signal. |

5724 **EXAMPLES**

5725 None.

5726 **APPLICATION USAGE**

5727 None.

5728 **RATIONALE**

5729 None.

5730 **FUTURE DIRECTIONS**

5731 None.

5732 **SEE ALSO**

5733 *catgets*( ), *catopen*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<nl_types.h>** |

5734 **CHANGE HISTORY**

5735 First released in Issue 2.

5736 **Issue 4**

5737 The [EBADF] and [EINTR] errors are added to the ERRORS section.

NAME

catgets — read a program message

SYNOPSIS

XSI

```
#include <nl_types.h>

char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);
```

DESCRIPTION

The *catgets*( ) function attempts to read message *msg_id*, in set *set_id*, from the message catalog identified by *catd*. The *catd* argument is a message catalog descriptor returned from an earlier call to *catopen*( ). The *s* argument points to a default message string which shall be returned by *catgets*( ) if it cannot retrieve the identified message.

The *catgets*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

If the identified message is retrieved successfully, *catgets*( ) shall return a pointer to an internal buffer area containing the null-terminated message string. If the call is unsuccessful for any reason, *s* shall be returned and *errno* may be set to indicate the error.

ERRORS

The *catgets*( ) function may fail if:

[EBADF]	The *catd* argument is not a valid message catalog descriptor open for reading.
[EINTR]	The read operation was terminated due to the receipt of a signal, and no data was transferred.
[EINVAL]	The message catalog identified by *catd* is corrupted.
[ENOMSG]	The message identified by *set_id* and *msg_id* is not in the message catalog.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

catclose( ), *catopen*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<nl_types.h>**

CHANGE HISTORY

First released in Issue 2.

Issue 4

The type of argument *s* is changed from **char*** to **const char***.

The [EBADF] and [EINTR] errors are added to the ERRORS section.

Issue 4, Version 2

The following changes are incorporated for X/OPEN UNIX conformance:

- The RETURN VALUE section notes that *errno* may be set to indicate an error.
- In the ERRORS section, [EINVAL] and [ENOMSG] are added as optional errors.

Issue 5

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.

NAME

catopen — open a message catalog

SYNOPSIS

XSI
```
#include <nl_types.h>

nl_catd catopen(const char *name, int oflag);
```

DESCRIPTION

The *catopen*( ) function shall open a message catalog and return a message catalog descriptor. The *name* argument specifies the name of the message catalog to be opened. If *name* contains a `'/'`, then *name* specifies a complete name for the message catalog. Otherwise, the environment variable *NLSPATH* is used with *name* substituted for %*N* (see the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 8, Environment Variables). If *NLSPATH* does not exist in the environment, or if a message catalog cannot be found in any of the components specified by *NLSPATH*, then an implementation-defined default path is used. This default may be affected by the setting of *LC_MESSAGES* if the value of *oflag* is NL_CAT_LOCALE, or the *LANG* environment variable if *oflag* is 0.

A message catalog descriptor remains valid in a process until that process closes it, or a successful call to one of the *exec* functions. A change in the setting of the *LC_MESSAGES* category may invalidate existing open catalogs.

If a file descriptor is used to implement message catalog descriptors, the FD_CLOEXEC flag shall be set; see **<fcntl.h>**.

If the value of the *oflag* argument is 0, the *LANG* environment variable is used to locate the catalog without regard to the *LC_MESSAGES* category. If the *oflag* argument is NL_CAT_LOCALE, the *LC_MESSAGES* category is used to locate the message catalog (see the Base Definitions volume of IEEE Std. 1003.1-200x, Section 8.2, Internationalization Variables).

RETURN VALUE

Upon successful completion, *catopen*( ) shall return a message catalog descriptor for use on subsequent calls to *catgets*( ) and *catclose*( ). Otherwise, *catopen*( ) shall return (**nl_catd**) –1 and set *errno* to indicate the error.

ERRORS

The *catopen*( ) function may fail if:

[EACCES] Search permission is denied for the component of the path prefix of the message catalog or read permission is denied for the message catalog.

[EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

[ENAMETOOLONG]
The length of a path name of the message catalog exceeds {PATH_MAX} or a path name component is longer than {NAME_MAX}.

[ENAMETOOLONG]
Path name resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

[ENFILE] Too many files are currently open in the system.

[ENOENT] The message catalog does not exist or the *name* argument points to an empty string.

[ENOMEM] Insufficient storage space is available.

[ENOTDIR] A component of the path prefix of the message catalog is not a directory.

EXAMPLES

None.

APPLICATION USAGE

Some implementations of *catopen*( ) use *malloc*( ) to allocate space for internal buffer areas. The *catopen*( ) function may fail if there is insufficient storage space available to accommodate these buffers.

Portable applications must assume that message catalog descriptors are not valid after a call to one of the *exec* functions.

Application writers should be aware that guidelines for the location of message catalogs have not yet been developed. Therefore they should take care to avoid conflicting with catalogs used by other applications and the standard utilities.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

catclose( ), *catgets*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<fcntl.h>**, **<nl_types.h>**, the Shell and Utilities volume of IEEE Std. 1003.1-200x

CHANGE HISTORY

First released in Issue 2.

Issue 4

The type of argument *name* is changed from **char*** to **const char***.

The DESCRIPTION is updated:

- To indicate the longevity of message catalog descriptors.
- To specify values for the *oflag* argument and the effect of *LC_MESSAGES* and *NLSPATH*.

The [EACCES], [EMFILE], [ENAMETOOLONG], [ENFILE], [ENOENT], and [ENOTDIR] errors are added to the ERRORS section.

The APPLICATION USAGE section is updated to indicate:

- Portable applications should not assume the continued validity of message catalog descriptors after a call to one of the *exec* functions.
- Message catalogs must be located with care.

Issue 4, Version 2

The following change is incorporated for X/OPEN UNIX conformance:

- In the ERRORS section, an [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of path name resolution of a symbolic link.

NAME

cbrt, cbrtf, cbrtl — cube root functions

SYNOPSIS

```
#include <math.h>

double cbrt(double x);
float cbrtf(float x);
long double cbrtl(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 real cube root of *x*.

An application wishing to check for error situations should set *errno* to 0 before calling these functions. If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the cube root of *x*.

If *x* is ±Inf, these functions shall return *x*.

If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

ERRORS

These functions may fail if:

[EDOM] The value of *x* is NaN.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

For some applications, a true cube root function, which returns negative results for negative arguments, is more appropriate than *pow*(*x*, 1.0/3.0), which returns a NaN for *x* less than 0.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The *cbrt*( ) function is no longer marked XSI.

The *cbrtf*( ) and *cbrtl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

5904 **NAME**
5905 ccos, ccosf, ccosl — complex cosine functions

5906 **SYNOPSIS**
5907 `#include <complex.h>`

5908 `double complex ccos(double complex z);`
5909 `float complex ccosf(float complex z);`
5910 `long double complex ccosl(long double complex z);`

5911 **DESCRIPTION**
5912 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5913 conflict between the requirements described here and the ISO C standard is unintentional. This
5914 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

5915 These functions shall compute the complex cosine of *z*.

5916 **RETURN VALUE**
5917 These functions shall return the complex cosine value.

5918 **ERRORS**
5919 No errors are defined.

5920 **EXAMPLES**
5921 None.

5922 **APPLICATION USAGE**
5923 None.

5924 **RATIONALE**
5925 None.

5926 **FUTURE DIRECTIONS**
5927 None.

5928 **SEE ALSO**
5929 *cacos*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

5930 **CHANGE HISTORY**
5931 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

ccosh, ccoshf, ccoshl — complex hyperbolic cosine functions

SYNOPSIS

```
#include <complex.h>

double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(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 hyperbolic cosine of *z*.

RETURN VALUE

These functions shall return the complex hyperbolic cosine value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cacosh( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

5960 **NAME**

5961 ceil, ceilf, ceill — ceiling value function

5962 **SYNOPSIS**

5963
```
#include <math.h>
```

5964
```
double ceil(double x);
```
5965
```
float ceilf(float x);
```
5966
```
long double ceill(long double x);
```

5967 **DESCRIPTION**

5968 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5969 conflict between the requirements described here and the ISO C standard is unintentional. This
5970 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

5971 The *ceil*( ), *ceilf*( ), and *ceill*( ) functions shall compute the smallest integral value not less than *x*.

5972 An application wishing to check for error situations should set *errno* to 0 before calling *ceil*( ). If
5973 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

5974 **RETURN VALUE**

5975 Upon successful completion, the *ceil*( ), *ceilf*( ), and *ceill*( ) functions shall return the smallest
5976 integral value not less than *x*, expressed as a type **double**.

5977 XSI If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

5978 If the correct value would cause overflow, HUGE_VAL shall be returned and *errno* set to
5979 XSI [ERANGE]. If *x* is ±Inf or ±0, the value of *x* shall be returned.

5980 **ERRORS**

5981 The *ceil*( ), *ceilf*( ), and *ceill*( ) functions shall fail if:

5982 [ERANGE] The result overflows.

5983 The *ceil*( ), *ceilf*( ), and *ceill*( ) functions may fail if:

5984 XSI [EDOM] The value of *x* is NaN.

5985 XSI No other errors shall occur.

5986 **EXAMPLES**

5987 None.

5988 **APPLICATION USAGE**

5989 The integral value returned by *ceil*( ) as a **double** need not be expressible as an **int** or **long**. The
5990 return value should be tested before assigning it to an integer type to avoid the undefined results
5991 of an integer overflow.

5992 The *ceil*( ) function can only overflow when the floating point representation has
5993 DBL_MANT_DIG > DBL_MAX_EXP.

5994 **RATIONALE**

5995 None.

5996 **FUTURE DIRECTIONS**

5997 None.

5998 **SEE ALSO**

5999 *floor*( ), *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 4

References to *matherr*( ) are removed.

The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the ISO C standard and to rationalize error handling in the mathematics functions.

The return value specified for [EDOM] is marked as an extension.

Support for *x* being ±Inf or ±0 is added to the RETURN VALUE section and marked as an extension.

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 *ceilf*( ) and *ceill*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

NAME

cexp, cexpf, cexpl — complex exponential functions

SYNOPSIS

```
#include <complex.h>

double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(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 exponent of z, defined as e^z.

RETURN VALUE

These functions shall return the complex exponential value of z.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clog( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

cfgetispeed — get input baud rate

SYNOPSIS

```
#include <termios.h>

speed_t cfgetispeed(const struct termios *termios_p);
```

DESCRIPTION

The *cfgetispeed*( ) function shall extract the input baud rate from the **termios** structure to which the *termios_p* argument points.

This function shall return exactly the value in the **termios** data structure, without interpretation.

RETURN VALUE

Upon successful completion, *cfgetispeed*( ) shall return a value of type **speed_t** representing the input baud rate.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The term *baud* is used historically here, but is not technically correct. This is properly ''bits per second'', which may not be the same as baud. However, the term is used because of the historical usage and understanding.

The *cfgetospeed*( ), *cfgetispeed*( ), *cfsetospeed*( ), and *cfsetispeed*( ) functions do not take arguments as numbers, but rather as symbolic names. There are two reasons for this:

1. Historically, numbers were not used because of the way the rate was stored in the data structure. This is retained even though a function is now used.
2. More importantly, only a limited set of possible rates is at all portable, and this constrains the application to that set.

There is nothing to prevent an implementation to accept, as an extension, a number (such as 126) if it wished, and because the encoding of the Bxxx symbols is not specified, this can be done so no ambiguity is introduced.

Setting the input baud rate to zero was a mechanism to allow for split baud rates. Clarifications in this volume of IEEE Std. 1003.1-200x have made it possible to determine whether split rates are supported and to support them without having to treat zero as a special case. Since this functionality is also confusing, it has been declared obsolescent. The 0 argument referred to is the literal constant 0, not the symbolic constant B0. This volume of IEEE Std. 1003.1-200x does not preclude B0 from being defined as the value 0; in fact, implementations would likely benefit from the two being equivalent. This volume of IEEE Std. 1003.1-200x does not fully specify whether the previous *cfsetispeed*( ) value is retained after a *tcgetattr*( ) as the actual value or as zero. Therefore, portable applications should always set both the input speed and output speed when setting either.

In historical implementations, the baud rate information is traditionally kept in **c_cflag**. Applications should be written to presume that this might be the case (and thus not blindly copy **c_cflag**), but not to rely on it in case it is in some other field of the structure. Setting the **c_cflag** field absolutely after setting a baud rate is a non-portable action because of this. In general, the

unused parts of the flag fields might be used by the implementation and should not be blindly copied from the descriptions of one terminal device to another.

FUTURE DIRECTIONS

None.

SEE ALSO

cfgetospeed( ), *cfsetispeed*( ), *cfsetospeed*( ), *tcgetattr*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<termios.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The type of the argument *termios_p* is changed from **struct termios*** to **const struct termios***.
- The DESCRIPTION is changed to indicate that the function simply returns the value from *termios_p*, irrespective of how that structure was obtained. Issue 3 states that if *termios_p* was not obtained by a successful call to *tcgetattr*( ), the behavior is undefined.

NAME

cfgetospeed — get output baud rate

SYNOPSIS

```
#include <termios.h>

speed_t cfgetospeed(const struct termios *termios_p);
```

DESCRIPTION

The *cfgetospeed*( ) function shall extract the output baud rate from the **termios** structure to which the *termios_p* argument points.

This function shall return exactly the value in the **termios** data structure, without interpretation.

RETURN VALUE

Upon successful completion, *cfgetospeed*( ) shall return a value of type **speed_t** representing the output baud rate.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Refer to *cfgetispeed*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

cfgetispeed( ), *cfsetispeed*( ), *cfsetospeed*( ), *tcgetattr*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<termios.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The type of the argument *termios_p* is changed from **struct termios*** to **const struct termios***.
- The DESCRIPTION is changed to indicate that the function simply returns the value from *termios_p*, irrespective of how that structure was obtained. Issue 3 states that if *termios_p* was not obtained by a successful call to *tcgetattr*( ), the behavior is undefined.

6139 **NAME**

6140 cfsetispeed — set input baud rate

6141 **SYNOPSIS**

6142 ```
#include <termios.h>
```

6143 ```
int cfsetispeed(struct termios *termios_p, speed_t speed);
```

6144 **DESCRIPTION**

6145 The *cfsetispeed*( ) function shall set the input baud rate stored in the structure pointed to by
6146 *termios_p* to *speed*.

6147 There shall be no effect on the baud rates set in the hardware until a subsequent successful call
6148 to *tcsetattr*( ) with the same **termios** structure. Similarly, errors resulting from attempts to set
6149 baud rates not supported by the terminal device need not be detected until the *tcsetattr*( )
6150 function is called.

6151 **RETURN VALUE**

6152 Upon successful completion, *cfsetispeed*( ) shall return 0; otherwise, –1 shall be returned, and |
6153 *errno* may be set to indicate the error. |

6154 **ERRORS**

6155 The *cfsetispeed*( ) function may fail if:

6156 [EINVAL] The *speed* value is not a valid baud rate. |

6157 [EINVAL] The value of *speed* is outside the range of possible speed values as specified in |
6158 **<termios.h>**. |

6159 **EXAMPLES**

6160 None.

6161 **APPLICATION USAGE**

6162 None.

6163 **RATIONALE**

6164 Refer to *cfgetispeed*( ).

6165 **FUTURE DIRECTIONS**

6166 None.

6167 **SEE ALSO**

6168 *cfgetispeed*( ), *cfgetospeed*( ), *cfsetospeed*( ), *tcsetattr*( ), the Base Definitions volume of |
6169 IEEE Std. 1003.1-200x, **<termios.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, |
6170 Chapter 11, General Terminal Interface |

6171 **CHANGE HISTORY**

6172 First released in Issue 3.

6173 Entry included for alignment with the POSIX.1-1988 standard.

6174 **Issue 4**

6175 The first description of the [EINVAL] error is added and is marked as an extension. |

6176 **Issue 4, Version 2**

6177 The ERRORS section is changed to indicate that [EINVAL] may be returned if the specified
6178 speed is outside the range of possible speed values given in **<termios.h>**.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The optional setting of *errno* and the [EINVAL] error conditions are added.

6183 **NAME**

6184 cfsetospeed — set output baud rate

6185 **SYNOPSIS**

6186 `#include <termios.h>`

6187 `int cfsetospeed(struct termios *`*`termios_p`*`, speed_t `*`speed`*`);`

6188 **DESCRIPTION**

6189 The *cfsetospeed*( ) function shall set the output baud rate stored in the structure pointed to by
6190 *termios_p* to *speed*.

6191 There shall be no effect on the baud rates set in the hardware until a subsequent successful call
6192 to *tcsetattr*( ) with the same **termios** structure. Similarly, errors resulting from attempts to set
6193 baud rates not supported by the terminal device need not be detected until the *tcsetattr*( )
6194 function is called.

6195 **RETURN VALUE**

6196 Upon successful completion, *cfsetospeed*( ) shall return 0; otherwise, it shall return –1 and *errno* |
6197 may be set to indicate the error. |

6198 **ERRORS**

6199 The *cfsetospeed*( ) function may fail if:

6200 [EINVAL] The *speed* value is not a valid baud rate. |

6201 [EINVAL] The value of *speed* is outside the range of possible speed values as specified in |
6202 **<termios.h>**. |

6203 **EXAMPLES**

6204 None.

6205 **APPLICATION USAGE**

6206 None.

6207 **RATIONALE**

6208 Refer to *cfgetispeed*( ).

6209 **FUTURE DIRECTIONS**

6210 None.

6211 **SEE ALSO**

6212 *cfgetispeed*( ), *cfgetospeed*( ), *cfsetispeed*( ), *tcsetattr*( ), the Base Definitions volume of |
6213 IEEE Std. 1003.1-200x, **<termios.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, |
6214 Chapter 11, General Terminal Interface |

6215 **CHANGE HISTORY**

6216 First released in Issue 3.

6217 Entry included for alignment with the POSIX.1-1988 standard.

6218 **Issue 4**

6219 The first description of the [EINVAL] error is added and is marked as an extension. |

6220 **Issue 4, Version 2**

6221 The ERRORS section is changed to indicate that [EINVAL] may be returned if the specified
6222 speed is outside the range of possible speed values given in **<termios.h>**.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The optional setting of *errno* and the [EINVAL] error conditions are added.

6227 **NAME**

6228 chdir — change working directory

6229 **SYNOPSIS**

6230 ```
#include <unistd.h>
```

6231 ```
int chdir(const char *path);
```

6232 **DESCRIPTION**

6233 The *chdir*( ) function shall cause the directory named by the path name pointed to by the *path*
6234 argument to become the current working directory; that is, the starting point for path searches
6235 for path names not beginning with `'/'`.

6236 **RETURN VALUE**

6237 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned, the current
6238 working directory shall remain unchanged, and *errno* shall be set to indicate the error.

6239 **ERRORS**

6240 The *chdir*( ) function shall fail if:

6241 [EACCES] Search permission is denied for any component of the path name.

6242 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
6243 argument.

6244 [ENAMETOOLONG]
6245 The length of the *path* argument exceeds {PATH_MAX} or a path name
6246 component is longer than {NAME_MAX}.

6247 [ENOENT] A component of *path* does not name an existing directory or *path* is an empty
6248 string.

6249 [ENOTDIR] A component of the path name is not a directory.

6250 The *chdir*( ) function may fail if:

6251 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
6252 resolution of the *path* argument.

6253 [ENAMETOOLONG]
6254 As a result of encountering a symbolic link in resolution of the *path* argument,
6255 the length of the substituted path name string exceeded {PATH_MAX}.

6256 **EXAMPLES**

6257 **Changing the Current Working Directory**

6258 The following example makes the value pointed to by **directory**, **/tmp**, the current working
6259 directory.

6260 ```
#include <unistd.h>
```

6261 ```
...
```

6262 ```
char *directory = "/tmp";
```

6263 ```
int ret;
```

6264 ```
ret = chdir (directory);
```
```

APPLICATION USAGE

The *chdir*( ) function only affects the working directory of the current process. The result if a NULL argument is passed to *chdir*( ) is unspecified because some implementations dynamically allocate space in that case.

RATIONALE

The *chdir*( ) function only affects the working directory of the current process.

The result if a NULL argument is passed to *chdir*( ) is left implementation-defined because some implementations dynamically allocate space in that case.

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 4

The **<unistd.h>** header is added to the SYNOPSIS section.

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument *path* is changed from **char*** to **const char***.

The following change is incorporated for alignment with the FIPS requirements:

- In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path name component is larger that {NAME_MAX} is now defined as mandatory and marked as an extension.

Issue 4, Version 2

The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- It states that [ELOOP] is returned if too many symbolic links are encountered during path name resolution.
- A second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of path name resolution of a symbolic link.

Issue 6

The APPLICATION USAGE section is added.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

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.

6304 **NAME**

6305 chmod — change mode of a file

6306 **SYNOPSIS**

6307 `#include <sys/stat.h>`

6308 `int chmod(const char *`*`path`*`, mode_t `*`mode`*`);`

6309 **DESCRIPTION**

6310 XSI The *chmod*( ) function shall change S_ISUID, S_ISGID, S_ISVTX, and the file permission bits of
6311 the file named by the path name pointed to by the *path* argument to the corresponding bits in the
6312 *mode* argument. The application shall ensure that the effective user ID of the process matches the
6313 owner of the file or the process has appropriate privileges in order to do this.

6314 S_ISUID, S_ISGID, and the file permission bits are described in **<sys/stat.h>**.

6315 XSI If a directory is writable and the mode bit S_ISVTX is set on the directory, a process may remove
6316 or rename files within that directory only if one or more of the following is true:

6317 • The effective user ID of the process is the same as that of the owner ID of the file.

6318 • The effective user ID of the process is the same as that of the owner ID of the directory.

6319 • The process has appropriate privileges.

6320

6321 XSI If the S_ISVTX bit is set on a non-directory file, the behavior is unspecified.

6322 If the calling process does not have appropriate privileges, and if the group ID of the file does
6323 not match the effective group ID or one of the supplementary group IDs and if the file is a
6324 regular file, bit S_ISGID (set-group-ID on execution) in the file's mode shall be cleared upon
6325 successful return from *chmod*( ).

6326 Additional implementation-defined restrictions may cause the S_ISUID and S_ISGID bits in
6327 *mode* to be ignored.

6328 The effect on file descriptors for files open at the time of a call to *chmod*( ) is implementation-
6329 defined.

6330 Upon successful completion, *chmod*( ) shall mark for update the *st_ctime* field of the file.

6331 **RETURN VALUE**

6332 Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to
6333 indicate the error. If –1 is returned, no change to the file mode occurs.

6334 **ERRORS**

6335 The *chmod*( ) function shall fail if:

6336 [EACCES] Search permission is denied on a component of the path prefix.

6337 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
6338 argument.

6339 [ENAMETOOLONG]
6340 The length of the *path* argument exceeds {PATH_MAX} or a path name
6341 component is longer than {NAME_MAX}.

6342 [ENOTDIR] A component of the path prefix is not a directory.

6343 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

6344 [EPERM] The effective user ID does not match the owner of the file and the process
6345 does not have appropriate privileges.

[EROFS] The named file resides on a read-only file system.

The *chmod*( ) function may fail if:

[EINTR] A signal was caught during execution of the function.

[EINVAL] The value of the *mode* argument is invalid.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted path name strings exceeded {PATH_MAX}.

EXAMPLES

Setting Read Permissions for User, Group, and Others

The following example sets read permissions for the owner, group, and others.

```
#include <sys/stat.h>

const char *path;
...
chmod(path, S_IRUSR|S_IRGRP|S_IROTH);
```

Setting Read, Write, and Execute Permissions for the Owner Only

The following example sets read, write, and execute permissions for the owner, and no permissions for group and others.

```
#include <sys/stat.h>

const char *path;
...
chmod(path, S_IRWXU);
```

Setting Different Permissions for Owner, Group, and Other

The following example sets owner permissions for CHANGEFILE to read, write, and execute, group permissions to read and execute, and other permissions to read.

```
#include <sys/stat.h>

#define CHANGEFILE "/etc/myfile"
...
chmod(CHANGEFILE, S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH);
```

Setting and Checking File Permissions

The following example sets the file permission bits for a file named **/home/cnd/mod1**, then calls the *stat*( ) function to verify the permissions.

```
#include <sys/types.h>
#include <sys/stat.h>

int status;
struct stat buffer
...
```

```
chmod("home/cnd/mod1", S_IRWXU|S_IRWXG|S_IROTH|S_IWOTH);
status = stat("home/cnd/mod1", &buffer;);
```

APPLICATION USAGE

In order to ensure that the S_ISUID and S_ISGID bits are set, an application requiring this should use *stat*( ) after a successful *chmod*( ) to verify this.

Any file descriptors currently open by any process on the file could possibly become invalid if the mode of the file is changed to a value which would deny access to that process. One situation where this could occur is on a stateless file system. This behavior will not occur in a conforming environment.

RATIONALE

This volume of IEEE Std. 1003.1-200x specifies that the S_ISGID bit is cleared by *chmod*( ) on a regular file under certain conditions. This is specified on the assumption that regular files may be executed, and the system should prevent users from making executable *setgid*( ) files perform with privileges that the caller does not have. On implementations that support execution of other file types, the S_ISGID bit should be cleared for those file types under the same circumstances.

Implementations that use the S_ISUID bit to indicate some other function (for example, mandatory record locking) on non-executable files need not clear this bit on writing. They should clear the bit for executable files and any other cases where the bit grants special powers to processes that change the file contents. Similar comments apply to the S_ISGID bit.

FUTURE DIRECTIONS

None.

SEE ALSO

chown( ), *mkdir*( ), *mkfifo*( ), *open*( ), *stat*( ), *statvfs*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

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

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

The [EINVAL] error is marked as an extension.

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument *path* is changed from **char*** to **const char***.

The following change is incorporated for alignment with the FIPS requirements:

- In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path name component is larger that {NAME_MAX} is now defined as mandatory and marked as an extension.

Issue 4, Version 2

The following changes are incorporated for X/OPEN UNIX conformance:

- The DESCRIPTION is updated to describe X/OPEN UNIX functionality relating to permission checks applied when removing or renaming files in a directory having the S_ISVTX bit set.
- In the ERRORS section, the condition whereby [ELOOP] is returned if too many symbolic links are encountered during path name resolution is defined as mandatory, and [EINTR] is

added as an optional error.

- In the ERRORS section, a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of path name resolution of a symbolic link.

Issue 6

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

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 DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

chown — change owner and group of a file

SYNOPSIS

```
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
```

DESCRIPTION

The *path* argument points to a path name naming a file. The user ID and group ID of the named file are set to the numeric values contained in *owner* and *group*, respectively.

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*:

- Changing the user ID is restricted to processes with appropriate privileges.
- 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.

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.

If *owner* or *group* is specified as (**uid_t**)–1 or (**gid_t**)–1, respectively, the corresponding ID of the file is unchanged.

Upon successful completion, *chown*( ) shall mark for update the *st_ctime* field of the file.

RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to indicate the error. If –1 is returned, no changes are made in the user ID and group ID of the file.

ERRORS

The *chown*( ) function shall fail if:

[EACCES] Search permission is denied on a component of the path prefix. |

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* | argument.

[ENAMETOOLONG] |
The length of the *path* argument exceeds {PATH_MAX} or a path name component is longer than {NAME_MAX}. |

[ENOTDIR] A component of the path prefix is not a directory. |

[ENOENT] A component of *path* does not name an existing file or *path* is an empty string. |

[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.

[EROFS] The named file resides on a read-only file system.

The *chown*( ) function may fail if:

[EIO] An I/O error occurred while reading or writing to the file system.

[EINTR] The *chown*( ) function was interrupted by a signal which was caught.

[EINVAL] The owner or group ID supplied is not a value supported by the implementation.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted path name string exceeded {PATH_MAX}.

EXAMPLES

None.

APPLICATION USAGE

Although *chown*( ) can be used on some systems by the file owner to change the owner and group to any desired values, the only portable use of this function is to change the group of a file to the effective GID of the calling process or to a member of its group set.

RATIONALE

System III and System V allow a user to give away files; that is, the owner of a file may change its user ID to anything. This is a serious problem for implementations that are intended to meet government security regulations. Version 7 and 4.3 BSD permit only the superuser to change the user ID of a file. Some government agencies (usually not ones concerned directly with security) find this limitation too confining. This volume of IEEE Std. 1003.1-200x uses *may* to permit secure implementations while not disallowing System V.

System III and System V allow the owner of a file to change the group ID to anything. Version 7 permits only the superuser to change the group ID of a file. 4.3 BSD permits the owner to change the group ID of a file to its effective group ID or to any of the groups in the list of supplementary group IDs, but to no others.

The POSIX.1-1990 standard requires that the *chown*( ) function invoked by a non-appropriate privileged process clear the S_ISGID and the S_ISUID bits for regular files, and permits them to be cleared for other types of files. This is so that changes in accessibility do not accidentally cause files to become security holes. Unfortunately, requiring these bits to be cleared on non-executable data files also clears the mandatory file locking bit (shared withwith S_ISGID), which is an extension on many implementations (it first appeared in System V). These bits should only be required to be cleared on regular files that have one or more of their execute bits set.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *pathconf*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

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

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

The value for *owner* of (**uid_t**)–1 is added to the DESCRIPTION to allow the use of –1 by the owner of a file to change the group ID only.

The APPLICATION USAGE section is added.

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument *path* is changed from **char*** to **const char***.

The following changes are incorporated for alignment with the FIPS requirements:

- In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path name component is larger that {NAME_MAX} is now defined as mandatory and marked as an extension.
- In the ERRORS section, the condition whereby [EPERM] is returned when an attempt is made to change the user ID of a file and the caller does not have appropriate privileges is now defined as mandatory and marked as an extension.

Issue 4, Version 2

The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- It states that [ELOOP] is returned if too many symbolic links are encountered during path name resolution.
- The [EIO] and [EINTR] optional conditions are added.
- A second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of path name resolution of a symbolic link.

Issue 6

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The wording describing the optional dependency on _POSIX_CHOWN_RESTRICTED is restored.
- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.
- The [EPERM] error is restored as an error dependent on _POSIX_CHOWN_RESTRICTED. This is since its operand is a path name and applications should be aware that the error may not occur for that path name if the file system does not support _POSIX_CHOWN_RESTRICTED.

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 value for *owner* of (**uid_t**)–1 allows the use of –1 by the owner of a file to change the group ID only.
- The [ELOOP] mandatory error condition is added.
- The [EIO] 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:

- Clarification is added that the S_ISUID and S_ISGID bits do not need to be cleared when the process has appropriate privileges.
- The [ELOOP] optional error condition is added.

NAME

cimag, cimagf, cimagl — complex imaginary functions

SYNOPSIS

```
#include <complex.h>

double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(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 imaginary part of *z*.

RETURN VALUE

These functions shall return the imaginary part value (as a real).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

For a variable *z* of complex type:

```
z == creal(z) + cimag(z)*I
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

carg( ), *conj*( ), *cproj*( ), *creal*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

clearerr — clear indicators on a stream

SYNOPSIS

```
#include <stdio.h>

void clearerr(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 *clearerr*( ) function shall clear the end-of-file and error indicators for the stream to which *stream* points.

RETURN VALUE

The *clearerr*( ) function shall return no value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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.

6633 **NAME**

6634 clock — report CPU time used

6635 **SYNOPSIS**

6636 `#include <time.h>`

6637 `clock_t clock(void);`

6638 **DESCRIPTION**

6639 CX The functionality described on this reference page is aligned with the ISO C standard. Any
6640 conflict between the requirements described here and the ISO C standard is unintentional. This
6641 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

6642 The *clock*( ) function shall return the implementation's best approximation to the processor time
6643 used by the process since the beginning of an implementation-defined time related only to the
6644 process invocation.

6645 **RETURN VALUE**

6646 To determine the time in seconds, the value returned by *clock*( ) should be divided by the value
6647 XSI of the macro CLOCKS_PER_SEC. CLOCKS_PER_SEC is defined to be one million in **<time.h>**.
6648 If the processor time used is not available or its value cannot be represented, the function shall
6649 return the value (**clock_t**)–1.

6650 **ERRORS**

6651 No errors are defined.

6652 **EXAMPLES**

6653 None.

6654 **APPLICATION USAGE**

6655 In order to measure the time spent in a program, *clock*( ) should be called at the start of the
6656 program and its return value subtracted from the value returned by subsequent calls. The value
6657 returned by *clock*( ) is defined for compatibility across systems that have clocks with different
6658 resolutions. The resolution on any particular system need not be to microsecond accuracy.

6659 The value returned by *clock*( ) may wrap around on some systems. For example, on a machine
6660 with 32-bit values for **clock_t**, it wraps after 2 147 seconds or 36 minutes.

6661 **RATIONALE**

6662 None.

6663 **FUTURE DIRECTIONS**

6664 None.

6665 **SEE ALSO**

6666 *asctime*( ), *ctime*( ), *difftime*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ),
6667 the Base Definitions volume of IEEE Std. 1003.1-200x, **<time.h>**

6668 **CHANGE HISTORY**

6669 First released in Issue 1. Derived from Issue 1 of the SVID.

6670 **Issue 4**

6671 Reference to the resolution of CLOCKS_PER_SEC is marked as an extension.

6672 The ERRORS section is added.

6673 Advice on how to calculate the time spent in a program is added to the APPLICATION USAGE
6674 section.

6675 The following changes are incorporated for alignment with the ISO C standard:

- The **<time.h>** header is added to the SYNOPSIS section.
- The DESCRIPTION and RETURN VALUE sections, though functionally equivalent to Issue 3, are rewritten for clarity and consistency with the ISO C standard. This issue also defines under what circumstances (**clock_t**)–1 can be returned by the function.
- The function is no longer marked as an extension.

NAME

clock_getcpuclockid — access a process CPU-time clock (**REALTIME**)

SYNOPSIS

CPT
```
#include <time.h>

int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);
```

DESCRIPTION

The *clock_getcpuclockid*( ) function shall return the clock ID of the CPU-time clock of the process specified by *pid*. If the process described by *pid* exists and the calling process has permission, the clock ID of this clock shall be returned in *clock_id*.

If *pid* is zero, the *clock_getcpuclockid*( ) function shall return the clock ID of the CPU-time clock of the process making the call, in *clock_id*.

The conditions under which one process has permission to obtain the CPU-time clock ID of other processes are implementation-defined.

RETURN VALUE

Upon successful completion, *clock_getcpuclockid*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *clock_getcpuclockid*( ) function shall fail if:

[EPERM] The requesting process does not have permission to access the CPU-time clock for the process.

The *clock_getcpuclockid*( ) function may fail if:

[ESRCH] No process can be found corresponding to the process specified by *pid*.

EXAMPLES

None.

APPLICATION USAGE

The *clock_getcpuclockid*( ) function is part of the Process CPU-Time Clocks option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clock_getres( ), *timer_create*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<time.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.

NAME

clock_getres, clock_gettime, clock_settime — clock and timer functions (**REALTIME**)

SYNOPSIS

TMR
```
#include <time.h>

int clock_getres(clockid_t clock_id, struct timespec *res);
int clock_settime(clockid_t clock_id, const struct timespec *tp);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
```

DESCRIPTION

The resolution of any clock can be obtained by calling *clock_getres*( ). Clock resolutions are implementation-defined and cannot be set by a process. If the argument *res* is not NULL, the resolution of the specified clock shall be stored in the location pointed to by *res*. If *res* is NULL, the clock resolution is not returned. If the *time* argument of *clock_settime*( ) is not a multiple of *res*, then the value is truncated to a multiple of *res*.

The *clock_gettime*( ) function shall return the current value *tp* for the specified clock, *clock_id*.

The *clock_settime*( ) function shall set the specified clock, *clock_id*, to the value specified by *tp*. Time values that are between two consecutive non-negative integer multiples of the resolution of the specified clock are truncated down to the smaller multiple of the resolution.

A clock may be system-wide (that is, visible to all processes) or per-process (measuring time that is meaningful only within a process). All implementations shall support a *clock_id* of CLOCK_REALTIME defined in **<time.h>**. This clock represents the realtime clock for the system. For this clock, the values returned by *clock_gettime*( ) and specified by *clock_settime*( ) represent the amount of time (in seconds and nanoseconds) since the Epoch. An implementation may also support additional clocks. The interpretation of time values for these clocks is unspecified.

If the value of the CLOCK_REALTIME clock is set via *clock_settime*( ), the new value of the clock shall be used to determine the time of expiration for absolute time services based upon the CLOCK_REALTIME clock. This applies to the time at which armed absolute timers expire. If the absolute time requested at the invocation of such a time service is before the new value of the clock, the time service shall expire immediately as if the clock had reached the requested time normally.

Setting the value of the CLOCK_REALTIME clock via *clock_settime*( ) shall have no effect on threads that are blocked waiting for a relative time service based upon this clock, including the *nanosleep*( ) function; nor on the expiration of relative timers based upon this clock. Consequently, these time services shall expire when the requested relative interval elapses, independently of the new or old value of the clock.

MON If the Monotonic Clock option is supported, all implementations shall support a *clock_id* of CLOCK_MONOTONIC defined in **<time.h>**. This clock represents the monotonic clock for the system. For this clock, the value returned by *clock_gettime*( ) represents the amount of time (in seconds and nanoseconds) since an unspecified point in the past (for example, system start-up time, or the Epoch). This point does not change after system start-up time. The value of the CLOCK_MONOTONIC clock cannot be set via *clock_settime*( ). This function shall fail if it is invoked with a *clock_id* argument of CLOCK_MONOTONIC.

The effect of setting a clock via *clock_settime*( ) on armed per-process timers associated with a clock other than CLOCK_REALTIME is implementation-defined.

CS If the value of the CLOCK_REALTIME clock is set via *clock_settime*( ), the new value of the clock shall be used to determine the time at which the system shall awaken a thread blocked on an

absolute *clock_nanosleep*( ) call based upon the CLOCK_REALTIME clock. If the absolute time requested at the invocation of such a time service is before the new value of the clock, the call shall return immediately as if the clock had reached the requested time normally.

Setting the value of the CLOCK_REALTIME clock via *clock_settime*( ) shall have no effect on any thread that is blocked on a relative *clock_nanosleep*( ) call. Consequently, the call shall return when the requested relative interval elapses, independently of the new or old value of the clock.

The appropriate privilege to set a particular clock is implementation-defined.

CPT If _POSIX_CPUTIME is defined, implementations shall support clock ID values obtained by invoking *clock_getcpuclockid*( ), which represent the CPU-time clock of a given process. Implementations shall also support the special **clockid_t** value CLOCK_PROCESS_CPUTIME_ID, which represents the CPU-time clock of the calling process when invoking one of the *clock_**( ) or *timer_**( ) functions. For these clock IDs, the values returned by *clock_gettime*( ) and specified by *clock_settime*( ) represent the amount of execution time of the process associated with the clock. Changing the value of a CPU-time clock via *clock_settime*( ) shall have no effect on the behavior of the sporadic server scheduling policy (see **Scheduling Policies** (on page 546)).

TCT If _POSIX_THREAD_CPUTIME is defined, implementations shall support clock ID values obtained by invoking *pthread_getcpuclockid*( ), which represent the CPU-time clock of a given thread. Implementations shall also support the special **clockid_t** value CLOCK_THREAD_CPUTIME_ID, which represents the CPU-time clock of the calling thread when invoking one of the *clock_**( ) or *timer_**( ) functions. For these clock IDs, the values returned by *clock_gettime*( ) and specified by *clock_settime*( ) represent the amount of execution time of the thread associated with the clock. Changing the value of a CPU-time clock via *clock_settime*( ) shall have no effect on the behavior of the sporadic server scheduling policy (see **Scheduling Policies** (on page 546)).

RETURN VALUE

A return value of 0 shall indicate that the call succeeded. A return value of –1 shall indicate that an error occurred, and *errno* shall be set to indicate the error.

ERRORS

The *clock_getres*( ), *clock_gettime*( ), and *clock_settime*( ) functions shall fail if:

[EINVAL] The *clock_id* argument does not specify a known clock.

The *clock_settime*( ) function shall fail if:

[EINVAL] The *tp* argument to *clock_settime*( ) is outside the range for the given clock ID.

[EINVAL] The *tp* argument specified a nanosecond value less than zero or greater than or equal to 1 000 million.

MON [EINVAL] The value of the *clock_id* argument is CLOCK_MONOTONIC.

The *clock_settime*( ) function may fail if:

[EPERM] The requesting process does not have the appropriate privilege to set the specified clock.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Timers option and need not be available on all implementations.

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.
  - *clock_settime*( ) failing for CLOCK_MONOTONIC is specified.
  - The effects of *clock_settime*( ) on the *clock_nanosleep*( ) function with respect to CLOCK_REALTIME is 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.

6844 **NAME**

6845 clock_nanosleep — high resolution sleep with specifiable clock

6846 **SYNOPSIS**

6847 CS
```
#include <time.h>

int clock_nanosleep(clockid_t clock_id, int flags,
    const struct timespec *rqtp, struct timespec *rmtp);
```
6848
6849
6850

6851 **DESCRIPTION**

6852 If the flag TIMER_ABSTIME is not set in the *flags* argument, the *clock_nanosleep*( ) function shall
6853 cause the current thread to be suspended from execution until either the time interval specified
6854 by the *rqtp* argument has elapsed, or a signal is delivered to the calling thread and its action is to
6855 invoke a signal-catching function, or the process is terminated. The clock used to measure the
6856 time shall be the clock specified by *clock_id*.

6857 If the flag TIMER_ABSTIME is set in the *flags* argument, the *clock_nanosleep*( ) function shall
6858 cause the current thread to be suspended from execution until either the time value of the clock
6859 specified by *clock_id* reaches the absolute time specified by the *rqtp* argument, or a signal is
6860 delivered to the calling thread and its action is to invoke a signal-catching function, or the
6861 process is terminated. If, at the time of the call, the time value specified by *rqtp* is less than or
6862 equal to the time value of the specified clock, then *clock_nanosleep*( ) shall return immediately
6863 and the calling process shall not be suspended.

6864 The suspension time caused by this function may be longer than requested because the
6865 argument value is rounded up to an integer multiple of the sleep resolution, or because of the
6866 scheduling of other activity by the system. But, except for the case of being interrupted by a
6867 signal, the suspension time for the relative *clock_nanosleep*( ) function (that is, with the
6868 TIMER_ABSTIME flag not set) shall not be less than the time interval specified by *rqtp*, as
6869 measured by the corresponding clock. The suspension for the absolute *clock_nanosleep*( ) function
6870 (that is, with the TIMER_ABSTIME flag set) shall be in effect at least until the value of the
6871 corresponding clock reaches the absolute time specified by *rqtp*, except for the case of being
6872 interrupted by a signal.

6873 The use of the *clock_nanosleep*( ) function shall have no effect on the action or blockage of any
6874 signal.

6875 The *clock_nanosleep*( ) function shall fail if the *clock_id* argument refers to the CPU-time clock of
6876 the calling thread. It is unspecified if *clock_id* values of other CPU-time clocks are allowed.

6877 **RETURN VALUE**

6878 If the *clock_nanosleep*( ) function returns because the requested time has elapsed, its return value
6879 shall be zero.

6880 If the *clock_nanosleep*( ) function returns because it has been interrupted by a signal, it shall return
6881 the corresponding error value. For the relative *clock_nanosleep*( ) function, if the *rmtp* argument is
6882 non-NULL, the **timespec** structure referenced by it shall be updated to contain the amount of
6883 time remaining in the interval (the requested time minus the time actually slept). If the *rmtp*
6884 argument is NULL, the remaining time is not returned. The absolute *clock_nanosleep*( ) function
6885 has no effect on the structure referenced by *rmtp*.

6886 If *clock_nanosleep*( ) fails, it shall return the corresponding error value.

6887 **ERRORS**

6888 The *clock_nanosleep*( ) function shall fail if:

6889 [EINTR] The *clock_nanosleep*( ) function was interrupted by a signal.

6890 [EINVAL] The *rqtp* argument specified a nanosecond value less than zero or greater than
6891 or equal to 1 000 million; or the TIMER_ABSTIME flag was specified in flags
6892 and the *rqtp* argument is outside the range for the clock specified by *clock_id*;
6893 or the *clock_id* argument does not specify a known clock, or specifies the
6894 CPU-time clock of the calling thread.

6895 [ENOTSUP] The *clock_id* argument specifies a clock for which *clock_nanosleep*( ) is not
6896 supported, such as a CPU-time clock.

6897 **EXAMPLES**

6898 None.

6899 **APPLICATION USAGE**

6900 Calling *clock_nanosleep*( ) with the value TIMER_ABSTIME not set in the *flags* argument and with
6901 a *clock_id* of CLOCK_REALTIME is equivalent to calling *nanosleep*( ) with the same *rqtp* and *rmtp*
6902 arguments.

6903 **RATIONALE**

6904 The *nanosleep*( ) function specifies that the system-wide clock CLOCK_REALTIME is used to
6905 measure the elapsed time for this time service. However, with the introduction of the monotonic
6906 clock CLOCK_MONOTONIC a new relative sleep function is needed to allow an application to
6907 take advantage of the special characteristics of this clock.

6908 There are many applications in which a process needs to be suspended and then activated
6909 multiple times in a periodic way; for example, to poll the status of a non-interrupting device or
6910 to refresh a display device. For these cases, it is known that precise periodic activation cannot be
6911 achieved with a relative *sleep*( ) or *nanosleep*( ) function call. Suppose, for example, a periodic
6912 process that is activated at time $T0$, executes for a while, and then wants to suspend itself until
6913 time $T0+T$, the period being T. If this process wants to use the *nanosleep*( ) function, it must first
6914 call *clock_gettime*( ) to get the current time, then calculate the difference between the current time
6915 and $T0+T$ and, finally, call *nanosleep*( ) using the computed interval. However, the process could
6916 be preempted by a different process between the two function calls, and in this case the interval
6917 computed would be wrong; the process would wake up later than desired. This problem would
6918 not occur with the absolute *clock_nanosleep*( ) function, since only one function call would be
6919 necessary to suspend the process until the desired time. In other cases, however, a relative sleep
6920 is needed, and that is why both functionalities are required.

6921 Although it is possible to implement periodic processes using the timers interface, this
6922 implementation would require the use of signals, and the reservation of some signal numbers. In
6923 this regard, the reasons for including an absolute version of the *clock_nanosleep*( ) function in
6924 IEEE Std. 1003.1-200x are the same as for the inclusion of the relative *nanosleep*( ).

6925 It is also possible to implement precise periodic processes using *pthread_cond_timedwait*( ), in
6926 which an absolute timeout is specified that takes effect if the condition variable involved is
6927 never signaled. However, the use of this interface is unnatural, and involves performing other
6928 operations on mutexes and condition variables that imply an unnecessary overhead.
6929 Furthermore, *pthread_cond_timedwait*( ) is not available in implementations that do not support
6930 threads.

6931 Although the interface of the relative and absolute versions of the new high resolution sleep
6932 service is the same *clock_nanosleep*( ) function, the *rmtp* argument is only used in the relative
6933 sleep. This argument is needed in the relative *clock_nanosleep*( ) function to reissue the function

call if it is interrupted by a signal, but it is not needed in the absolute *clock_nanosleep*( ) function call; if the call is interrupted by a signal, the absolute *clock_nanosleep*( ) function can be invoked again with the same *rqtp* argument used in the interrupted call.

FUTURE DIRECTIONS

None.

SEE ALSO

clock_getres( ), *nanosleep*( ), *pthread_cond_timedwait*( ), *sleep*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std. 1003.1j-2000.

NAME

clog, clogf, clogl — complex natural logarithm functions

SYNOPSIS

```
#include <complex.h>

double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(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 natural (base *e*) logarithm of *z*, with a branch cut along the negative real axis.

RETURN VALUE

These functions shall return the complex natural logarithm value, in the range of a strip mathematically unbounded along the real axis and in the interval [$-i\pi$, $+i\pi$] along the imaginary axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cexp( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

close — close a file descriptor

SYNOPSIS

```
#include <unistd.h>

int close(int fildes);
```

DESCRIPTION

The *close*( ) function shall deallocate the file descriptor indicated by *fildes*. To deallocate means to make the file descriptor available for return by subsequent calls to *open*( ) or other functions that allocate file descriptors. All outstanding record locks owned by the process on the file associated with the file descriptor shall be removed (that is, unlocked).

If *close*( ) is interrupted by a signal that is to be caught, it shall return –1 with *errno* set to [EINTR] and the state of *fildes* is unspecified. If an I/O error occurred while reading from or writing to the file system during *close*( ), it may return –1 with *errno* set to [EIO]; if this error is returned, the state of *fildes* is unspecified.

When all file descriptors associated with a pipe or FIFO special file are closed, any data remaining in the pipe or FIFO shall be discarded.

When all file descriptors associated with an open file description have been closed the open file description shall be freed.

If the link count of the file is 0, when all file descriptors associated with the file are closed, the space occupied by the file shall be freed and the file shall no longer be accessible.

XSR If a STREAMS-based *fildes* is closed and the calling process was previously registered to receive a SIGPOLL signal for events associated with that STREAM, the calling process shall be unregistered for events associated with the STREAM. The last *close*( ) for a STREAM causes the STREAM associated with *fildes* to be dismantled. If O_NONBLOCK is not set and there have been no signals posted for the STREAM, and if there is data on the module's write queue, *close*( ) waits for an unspecified time (for each module and driver) for any output to drain before dismantling the STREAM. The time delay can be changed via an I_SETCLTIME *ioctl*( ) request. If the O_NONBLOCK flag is set, or if there are any pending signals, *close*( ) does not wait for output to drain, and dismantles the STREAM immediately.

If the implementation supports STREAMS-based pipes, and *fildes* is associated with one end of a pipe, the last *close*( ) causes a hangup to occur on the other end of the pipe. In addition, if the other end of the pipe has been named by *fattach*( ), then the last *close*( ) forces the named end to be detached by *fdetach*( ). If the named end has no open file descriptors associated with it and gets detached, the STREAM associated with that end is also dismantled.

If *fildes* refers to the master side of a pseudo-terminal, and this is the last close, a SIGHUP signal is sent to the process group, if any, for which the slave side of the pseudo-terminal is the controlling terminal. It is unspecified whether closing the master side of the pseudo-terminal flushes all queued input and output.

If *fildes* refers to the slave side of a STREAMS-based pseudo-terminal, a zero-length message may be sent to the master.

AIO When there is an outstanding cancelable asynchronous I/O operation against *fildes* when *close*( ) is called, that I/O operation may be canceled. An I/O operation that is not canceled completes as if the *close*( ) operation had not yet occurred. All operations that are not canceled shall complete as if the *close*( ) blocked until the operations completed. The *close*( ) operation itself need not block awaiting such I/O completion. Whether any I/O operation is canceled, and which I/O operation may be canceled upon *close*( ), is implementation-defined.

MF|SHM If a shared memory object or a memory mapped file remains referenced at the last close (that is, a process has it mapped), then the entire contents of the memory object shall persist until the memory object becomes unreferenced. If this is the last close of a shared memory object or a memory mapped file and the close results in the memory object becoming unreferenced, and the memory object has been unlinked, then the memory object shall be removed.

If *fildes* refers to a socket, *close*( ) shall cause the socket to be destroyed. If the socket is in connection-mode, and the SO_LINGER option is set for the socket with non-zero linger time, and the socket has untransmitted data, then *close*( ) shall block for up to the current linger interval until all data is transmitted.

RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *close*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINTR] The *close*( ) function was interrupted by a signal.

The *close*( ) function may fail if:

[EIO] An I/O error occurred while reading from or writing to the file system.

EXAMPLES

Reassigning a File Descriptor

The following example closes the file descriptor associated with standard output for the current process, re-assigns standard output to a new file descriptor, and closes the original file descriptor to clean up. This example assumes that the file descriptor 0 (which is the descriptor for standard input) is not closed.

```
#include <unistd.h>
...
int pfd;
...
close(1);
dup(pfd);
close(pfd);
...
```

Incidentally, this is exactly what could be achieved using:

```
dup2(pfd, 1);
close(pfd);
```

Closing a File Descriptor

In the following example, *close*( ) is used to close a file descriptor after an unsuccessful attempt is made to associate that file descriptor with a stream.

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

```
#define LOCKFILE "/etc/ptmp"
...
int pfd;
FILE *fpfd;
...
if ((fpfd = fdopen (pfd, "w")) == NULL) {
    close(pfd);
    unlink(LOCKFILE);
    exit(1);
}
...
```

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

fattach( ), *fclose*( ), *fdetach*( ), *fopen*( ), *ioctl*( ), *open*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>**, Section 2.6 (on page 539)

CHANGE HISTORY

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

Issue 4

The **<unistd.h>** header is added to the SYNOPSIS section.

Issue 4, Version 2

The following changes are incorporated for X/OPEN UNIX conformance:

- The DESCRIPTION is updated to describe the actions of closing a file descriptor referring to a STREAMS-based file or either side of a pseudo-terminal.
- The ERRORS section describes a condition under which the [EIO] error may be returned.

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.

7109 **NAME**

7110 closedir — close a directory stream

7111 **SYNOPSIS**

7112 `#include <dirent.h>`

7113 `int closedir(DIR *dirp);`

7114 **DESCRIPTION**

7115 The *closedir*( ) function shall close the directory stream referred to by the argument *dirp*. Upon
7116 return, the value of *dirp* may no longer point to an accessible object of the type **DIR**. If a file
7117 descriptor is used to implement type **DIR**, that file descriptor shall be closed.

7118 **RETURN VALUE**

7119 Upon successful completion, *closedir*( ) shall return 0; otherwise, –1 shall be returned and *errno*
7120 set to indicate the error.

7121 **ERRORS**

7122 The *closedir*( ) function may fail if:

7123 [EBADF] The *dirp* argument does not refer to an open directory stream. |

7124 [EINTR] The *closedir*( ) function was interrupted by a signal. |

7125 **EXAMPLES**

7126 **Closing a Directory Stream**

7127 The following program fragment demonstrates how the *closedir*( ) function is used.

```
7128 ...
7129     DIR *dir;
7130     struct dirent *dp;
7131 ...
7132     if ((dir = opendir (".")) == NULL) {
7133 ...
7134     }
7135     while ((dp = readdir (dir)) != NULL) {
7136 ...
7137     }
7138     closedir(dir);
7139 ...
```

7140 **APPLICATION USAGE**

7141 None.

7142 **RATIONALE**

7143 None.

7144 **FUTURE DIRECTIONS**

7145 None.

7146 **SEE ALSO**

7147 *opendir*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<dirent.h>** |

CHANGE HISTORY

First released in Issue 2.

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

The [EINTR] error is marked as an extension.

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.
- The [EINTR] error condition is added as an optional error condition.

NAME

closelog, openlog, setlogmask, syslog — control system log

SYNOPSIS

XSI

```
#include <syslog.h>

void closelog(void);
void openlog(const char *ident, int logopt, int facility);
int setlogmask(int maskpri);
void syslog(int priority, const char *message, ... /* arguments */);
```

DESCRIPTION

The *syslog*( ) function shall send a message to an implementation-defined logging facility, which may log it in an implementation-defined system log, write it to the system console, forward it to a list of users, or forward it to the logging facility on another host over the network. The logged message shall include a message header and a message body. The message header contains at least a timestamp and a tag string.

The message body is generated from the *message* and following arguments in the same manner as if these were arguments to *printf*( ), except that occurrences of %m in the format string pointed to by the *message* argument are replaced by the error message string associated with the current value of *errno*. A trailing <newline> character is added if needed.

Values of the *priority* argument are formed by OR'ing together a severity level value and an optional facility value. If no facility value is specified, the current default facility value is used.

Possible values of severity level include:

LOG_EMERG — A panic condition.

LOG_ALERT — A condition that should be corrected immediately, such as a corrupted system database.

LOG_CRIT — Critical conditions, such as hard device errors.

LOG_ERR — Errors.

LOG_WARNING — Warning messages.

LOG_NOTICE — Conditions that are not error conditions, but that may require special handling.

LOG_INFO — Informational messages.

LOG_DEBUG — Messages that contain information normally of use only when debugging a program.

The facility indicates the application or system component generating the message. Possible facility values include:

LOG_USER — Messages generated by arbitrary processes. This is the default facility identifier if none is specified.

LOG_LOCAL0 — Reserved for local use.

LOG_LOCAL1 — Reserved for local use.

LOG_LOCAL2 — Reserved for local use.

7203 LOG_LOCAL3 Reserved for local use.

7204 LOG_LOCAL4 Reserved for local use.

7205 LOG_LOCAL5 Reserved for local use.

7206 LOG_LOCAL6 Reserved for local use.

7207 LOG_LOCAL7 Reserved for local use.

7208 The *openlog*( ) function shall set process attributes that affect subsequent calls to *syslog*( ). The
7209 *ident* argument is a string that is prepended to every message. The *logopt* argument indicates
7210 logging options. Values for *logopt* are constructed by a bitwise-inclusive OR of zero or more of
7211 the following:

7212 LOG_PID Log the process ID with each message. This is useful for identifying specific
7213 processes.

7214 LOG_CONS Write messages to the system console if they cannot be sent to the logging
7215 facility. The *syslog*( ) function ensures that the process does not acquire the
7216 console as a controlling terminal in the process of writing the message.

7217 LOG_NDELAY Open the connection to the logging facility immediately. Normally the open is
7218 delayed until the first message is logged. This is useful for programs that need
7219 to manage the order in which file descriptors are allocated.

7220 LOG_ODELAY Delay open until *syslog*( ) is called.

7221 LOG_NOWAIT Do not wait for child processes that may have been created during the course
7222 of logging the message. This option should be used by processes that enable
7223 notification of child termination using SIGCHLD, since *syslog*( ) may
7224 otherwise block waiting for a child whose exit status has already been
7225 collected.

7226 The *facility* argument encodes a default facility to be assigned to all messages that do not have
7227 an explicit facility already encoded. The initial default facility is LOG_USER.

7228 The *openlog*( ) and *syslog*( ) functions may allocate a file descriptor. It is not necessary to call
7229 *openlog*( ) prior to calling *syslog*( ).

7230 The *closelog*( ) function shall close any open file descriptors allocated by previous calls to
7231 *openlog*( ) or *syslog*( ).

7232 The *setlogmask*( ) function shall set the log priority mask for the current process to *maskpri* and
7233 return the previous mask. If the *maskpri* argument is 0, the current log mask is not modified.
7234 Calls by the current process to *syslog*( ) with a priority not set in *maskpri* shall be rejected. The
7235 default log mask allows all priorities to be logged. A call to *openlog*( ) is not required prior to
7236 calling *setlogmask*( ).

7237 Symbolic constants for use as values of the *logopt*, *facility*, *priority*, and *maskpri* arguments are
7238 defined in the **<syslog.h>** header.

7239 **RETURN VALUE**

7240 The *setlogmask*( ) function shall return the previous log priority mask. The *closelog*( ), *openlog*( ),
7241 and *syslog*( ) functions shall return no value.

7242 **ERRORS**

7243 No errors are defined.

EXAMPLES

Using openlog()

The following example causes subsequent calls to *syslog*( ) to log the process ID with each message, and to write messages to the system console if they cannot be sent to the logging facility.

```
#include <syslog.h>

char *ident = "Process demo";
int logopt = LOG_PID | LOG_CONS;
int facility = LOG_USER;
...
openlog(ident, logopt, facility);
```

Using setlogmask()

The following example causes subsequent calls to *syslog*( ) to accept error messages or messages generated by arbitrary processes, and to reject all other messages.

```
#include <syslog.h>

int result;
int mask = LOG_MASK (LOG_ERR | LOG_USER);
...
result = setlogmask(mask);
```

Using syslog

The following example sends the message "This is a message" to the default logging facility, marking the message as an error message generated by random processes.

```
#include <syslog.h>

char *message = "This is a message";
int priority = LOG_ERR | LOG_USER;
...
syslog(priority, message);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

printf( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<syslog.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

7283 **NAME**

7284 confstr — get configurable variables

7285 **SYNOPSIS**

7286 ```
#include <unistd.h>
```

7287 ```
size_t confstr(int name, char *buf, size_t len);
```

7288 **DESCRIPTION**

7289 The *confstr*( ) function provides a method for applications to get configuration-defined string
7290 values. Its use and purpose are similar to *sysconf*( ), but it is used where string values rather than
7291 numeric values are returned.

7292 The *name* argument represents the system variable to be queried. The implementation shall
7293 support the following name values, defined in **<unistd.h>**. It may support others:

7294 _CS_PATH
7295 _CS_POSIX_V6_ILP32_OFF32_CFLAGS
7296 _CS_POSIX_V6_ILP32_OFF32_LDFLAGS
7297 _CS_POSIX_V6_ILP32_OFF32_LIBS
7298 _CS_POSIX_V6_ILP32_OFF32_LINTFLAGS
7299 _CS_POSIX_V6_ILP32_OFFBIG_CFLAGS
7300 _CS_POSIX_V6_ILP32_OFFBIG_LDFLAGS
7301 _CS_POSIX_V6_ILP32_OFFBIG_LIBS
7302 _CS_POSIX_V6_ILP32_OFFBIG_LINTFLAGS
7303 _CS_POSIX_V6_LP64_OFF64_CFLAGS
7304 _CS_POSIX_V6_LP64_OFF64_LDFLAGS
7305 _CS_POSIX_V6_LP64_OFF64_LIBS
7306 _CS_POSIX_V6_LP64_OFF64_LINTFLAGS
7307 _CS_POSIX_V6_LPBIG_OFFBIG_CFLAGS
7308 _CS_POSIX_V6_LPBIG_OFFBIG_LDFLAGS
7309 _CS_POSIX_V6_LPBIG_OFFBIG_LIBS
7310 _CS_POSIX_V6_LPBIG_OFFBIG_LINTFLAGS
7311 XSI _CS_XBS5_ILP32_OFF32_CFLAGS (**LEGACY**)
7312 _CS_XBS5_ILP32_OFF32_LDFLAGS (**LEGACY**)
7313 _CS_XBS5_ILP32_OFF32_LIBS (**LEGACY**)
7314 _CS_XBS5_ILP32_OFF32_LINTFLAGS (**LEGACY**)
7315 _CS_XBS5_ILP32_OFFBIG_CFLAGS (**LEGACY**)
7316 _CS_XBS5_ILP32_OFFBIG_LDFLAGS (**LEGACY**)
7317 _CS_XBS5_ILP32_OFFBIG_LIBS (**LEGACY**)
7318 _CS_XBS5_ILP32_OFFBIG_LINTFLAGS (**LEGACY**)
7319 _CS_XBS5_LP64_OFF64_CFLAGS (**LEGACY**)
7320 _CS_XBS5_LP64_OFF64_LDFLAGS (**LEGACY**)
7321 _CS_XBS5_LP64_OFF64_LIBS (**LEGACY**)
7322 _CS_XBS5_LP64_OFF64_LINTFLAGS (**LEGACY**)
7323 _CS_XBS5_LPBIG_OFFBIG_CFLAGS (**LEGACY**)
7324 _CS_XBS5_LPBIG_OFFBIG_LDFLAGS (**LEGACY**)
7325 _CS_XBS5_LPBIG_OFFBIG_LIBS (**LEGACY**)
7326 _CS_XBS5_LPBIG_OFFBIG_LINTFLAGS (**LEGACY**)
7327

7328 If *len* is not 0, and if *name* has a configuration-defined value, *confstr*( ) shall copy that value into
7329 the *len*-byte buffer pointed to by *buf*. If the string to be returned is longer than *len* bytes,
7330 including the terminating null, then *confstr*( ) shall truncate the string to *len*–1 bytes and null-
7331 terminate the result. The application can detect that the string was truncated by comparing the

value returned by *confstr*( ) with *len*.

If *len* is 0 and *buf* is a null pointer, then *confstr*( ) shall still return the integer value as defined below, but shall not return a string. If *len* is 0 but *buf* is not a null pointer, the result is unspecified.

If the implementation supports the Shell option, the string stored in *buf* after a call to:

```
confstr(_CS_PATH, buf, sizeof(buf))
```

can be used as a value of the *PATH* environment variable that accesses all of the standard utilities of IEEE Std. 1003.1-200x, if the return value is less than or equal to *sizeof*(*buf*).

RETURN VALUE

If *name* has a configuration-defined value, *confstr*( ) shall return the size of buffer that would be needed to hold the entire configuration-defined value including the terminating null. If this return value is greater than *len*, the string returned in *buf* is truncated.

If *name* is invalid, *confstr*( ) shall return 0 and set *errno* to indicate the error.

If *name* does not have a configuration-defined value, *confstr*( ) shall return 0 and leave *errno* unchanged.

ERRORS

The *confstr*( ) function shall fail if:

[EINVAL] The value of the *name* argument is invalid.

EXAMPLES

None.

APPLICATION USAGE

An application can distinguish between an invalid *name* parameter value and one that corresponds to a configurable variable that has no configuration-defined value by checking if *errno* is modified. This mirrors the behavior of *sysconf*( ).

The original need for this function was to provide a way of finding the configuration-defined default value for the environment variable *PATH*. Since *PATH* can be modified by the user to include directories that could contain utilities replacing the standard utilities in the Shell and Utilities volume of IEEE Std. 1003.1-200x, applications need a way to determine the system-supplied *PATH* environment variable value that contains the correct search path for the standard utilities.

An application could use:

```
confstr(name, (char *)NULL, (size_t)0)
```

to find out how big a buffer is needed for the string value; use *malloc*( ) to allocate a buffer to hold the string; and call *confstr*( ) again to get the string. Alternately, it could allocate a fixed, static buffer that is big enough to hold most answers (perhaps 512 or 1 024 bytes), but then use *malloc*( ) to allocate a larger buffer if it finds that this is too small.

RATIONALE

Application developers can normally determine any configuration variable by means of reading from the stream opened by a call to:

```
popen("command -p getconf variable", "r");
```

The *confstr*( ) function with a *name* argument of _CS_PATH returns a string that can be used as a *PATH* environment variable setting that will reference the standard shell and utilities as described in the Shell and Utilities volume of IEEE Std. 1003.1-200x.

The *confstr*( ) function copies the returned string into a buffer supplied by the application instead of returning a pointer to a string. This allows a cleaner function in some implementations (such as those with lightweight threads) and resolves questions about when the application must copy the string returned.

FUTURE DIRECTIONS

None.

SEE ALSO

pathconf( ), *sysconf*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>**, the Shell and Utilities volume of IEEE Std. 1003.1-200x, *c99*

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 5

A table indicating the permissible values of *name* are added to the DESCRIPTION. All those marked EX are new in this issue.

Issue 6

The Open Group corrigenda item U033/7 has been applied. The return value for the case returning the size of the buffer now explicitly states that this includes the terminating null.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated with new arguments which can be used to determine configuration strings for C compiler flags, linker/loader flags, and libraries for each different supported programming environment. This is a change to support data size neutrality.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The DESCRIPTION is updated to include text describing how _CS_PATH can be used to obtain a *PATH* to access the standard utilities.

The macros associated with the *c89* programming models are marked LEGACY and new equivalent macros associated with *c99* are introduced.

NAME

conj, conjf, conjl — complex conjugate functions

SYNOPSIS

```
#include <complex.h>

double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(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 conjugate of *z*, by reversing the sign of its imaginary part.

RETURN VALUE

These functions return the complex conjugate value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

carg( ), *cimag*( ), *cproj*( ), *creal*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

7432 **NAME**

7433 connect — connect a socket

7434 **SYNOPSIS**

7435 ```
#include <sys/socket.h>
```

7436 ```
int connect(int socket, const struct sockaddr *address,
```

7437 ```
    socklen_t address_len);
```

7438 **DESCRIPTION**

7439 The *connect*( ) function requests a connection to be made on a socket. The function takes the
7440 following arguments:

7441 *socket* Specifies the file descriptor associated with the socket.

7442 *address* Points to a **sockaddr** structure containing the peer address. The length and
7443 format of the address depend on the address family of the socket.

7444 *address_len* Specifies the length of the **sockaddr** structure pointed to by the *address*
7445 argument.

7446 If the socket has not already been bound to a local address, *connect*( ) shall bind it to an address
7447 which, unless the socket's address family is AF_UNIX, is an unused local address.

7448 If the initiating socket is not connection-mode, then *connect*( ) shall set the socket's peer address,
7449 and no connection is made. For SOCK_DGRAM sockets, the peer address identifies where all
7450 datagrams are sent on subsequent *send*( ) functions, and limits the remote sender for subsequent
7451 *recv*( ) functions. If *address* is a null address for the protocol, the socket's peer address shall be
7452 reset.

7453 If the initiating socket is connection-mode, then *connect*( ) attempts to establish a connection to
7454 the address specified by the *address* argument.

7455 If the connection cannot be established immediately and O_NONBLOCK is not set for the file
7456 descriptor for the socket, *connect*( ) shall block for up to an unspecified timeout interval until the
7457 connection is established. If the timeout interval expires before the connection is established,
7458 *connect*( ) shall fail and the connection attempt shall be aborted. If *connect*( ) is interrupted by a
7459 signal that is caught while blocked waiting to establish a connection, *connect*( ) shall fail and set
7460 *errno* to [EINTR], but the connection request shall not be aborted, and the connection shall be
7461 established asynchronously.

7462 If the connection cannot be established immediately and O_NONBLOCK is set for the file
7463 descriptor for the socket, *connect*( ) shall fail and set *errno* to [EINPROGRESS], but the connection
7464 request shall not be aborted, and the connection shall be established asynchronously.
7465 Subsequent calls to *connect*( ) for the same socket, before the connection is established, shall fail
7466 and set *errno* to [EALREADY].

7467 When the connection has been established asynchronously, *select*( ) and *poll*( ) shall indicate that
7468 the file descriptor for the socket is ready for writing.

7469 The socket in use may require the process to have appropriate privileges to use the *connect*( )
7470 function.

7471 **RETURN VALUE**

7472 Upon successful completion, *connect*( ) shall return 0; otherwise, –1 shall be returned and *errno*
7473 set to indicate the error.
```

ERRORS

The *connect*( ) function shall fail if:

[EADDRNOTAVAIL]
The specified address is not available from the local machine.

[EAFNOSUPPORT]
The specified address is not a valid address for the address family of the specified socket.

[EALREADY] A connection request is already in progress for the specified socket.

[EBADF] The *socket* argument is not a valid file descriptor.

[ECONNREFUSED]
The target address was not listening for connections or refused the connection request.

[EINPROGRESS] O_NONBLOCK is set for the file descriptor for the socket and the connection cannot be immediately established; the connection shall be established asynchronously.

[EINTR] The attempt to establish a connection was interrupted by delivery of a signal that was caught; the connection shall be established asynchronously.

[EISCONN] The specified socket is connection-mode and is already connected.

[ENETUNREACH]
No route to the network is present.

[ENOTSOCK] The *socket* argument does not refer to a socket.

[EPROTOTYPE] The specified address has a different type than the socket bound to the specified peer address.

[ETIMEDOUT] The attempt to connect timed out before a connection was made.

If the address family of the socket is AF_UNIX, then *connect*( ) shall fail if:

[EIO] An I/O error occurred while reading from or writing to the file system.

[ELOOP] A loop exists in symbolic links encountered during resolution of the path name in *address*.

[ENAMETOOLONG]
A component of a path name exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.

[ENOENT] A component of the path name does not name an existing file or the path name is an empty string.

[ENOTDIR] A component of the path prefix of the path name in *address* is not a directory.

The *connect*( ) function may fail if:

[EACCES] Search permission is denied for a component of the path prefix; or write access to the named socket is denied.

[EADDRINUSE] Attempt to establish a connection that uses addresses that are already in use.

[ECONNRESET] Remote host reset the connection request.

[EHOSTUNREACH]
The destination host cannot be reached (probably because the host is down or

a remote router cannot reach it).

[EINVAL] The *address_len* argument is not a valid length for the address family; or invalid address family in the **sockaddr** structure.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the path name in *address*.

[ENAMETOOLONG] Path name resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

[ENETDOWN] The local network interface used to reach the destination is down.

[ENOBUFS] No buffer space is available.

[EOPNOTSUPP] The socket is listening and cannot be connected.

EXAMPLES

None.

APPLICATION USAGE

If *connect*( ) fails, the state of the socket is unspecified. Portable applications should close the file descriptor and create a new socket before attempting to reconnect.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

accept( ), *bind*( ), *close*( ), *getsockname*( ), *poll*( ), *select*( ), *send*( ), *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.

NAME

copysign, copysignf, copysignl — number manipulation function

SYNOPSIS

```
#include <math.h>

double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double 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.

These functions shall produce a value with the magnitude of *x* and the sign of *y*. They produce a NaN (with the sign of *y*) if *x* is a NaN. On implementations that represent a signed zero but do not treat negative zero consistently in arithmetic operations, these functions regard the sign of zero as positive.

An application wishing to check for error situations should set *errno* to 0 before calling these functions. If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return a value with the magnitude of *x* and the sign of *y*.

If *x* is ±Inf, these functions shall return *x*.

If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

ERRORS

These functions may fail if:

[EDOM] The value of *x* is NaN.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

copysign( ) and *signbit*( ) need not be consistent with each other if the arithmetic is not consistent in its treatment of zeros. For example, the IBM S/370 has instructions to flip the sign bit making it possible to create a negative zero, but ±0.0 x ±1.0 is always +0.0. In this case, *copysign*( ) will treat 0.0 as positive, while *signbit*( ) will treat it as negative.

FUTURE DIRECTIONS

None.

SEE ALSO

signbit( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

cos, cosf, cosl — cosine function

SYNOPSIS

```
#include <math.h>

double cos(double x);
float cosf(float x);
long double cosl(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 cosine of *x*, measured in radians.

An application wishing to check for error situations should set *errno* to 0 before calling *cos*(). If *errno* is non-zero on return, or the returned value is NaN, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the cosine of *x*.

XSI If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

XSI If *x* is ±Inf, either 0 shall be returned and *errno* set to [EDOM], or NaN shall be returned and *errno* may be set to [EDOM].

If the result underflows, 0 shall be returned and *errno* may be set to [ERANGE].

ERRORS

These functions may fail if:

XSI [EDOM] The value of *x* is NaN or *x* is ±Inf.

[ERANGE] The result underflows

XSI No other errors shall occur.

EXAMPLES

Taking the Cosine of a 45-Degree Angle

```
#include <math.h>
...
double radians = 45 * M_PI / 180;
double result;
...
result = cos(radians);
```

APPLICATION USAGE

The *cos*() function may lose accuracy when its argument is far from 0.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

acos( ), *isnan*( ), *sin*( ), *tan*( ), 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 4

References to *matherr*( ) are removed.

The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the ISO C standard and to rationalize error handling in the mathematics functions.

The return value specified for [EDOM] is marked as an extension.

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.

7635 **NAME**

7636 cosh, coshf, coshl — hyperbolic cosine function

7637 **SYNOPSIS**

7638
```
#include <math.h>
```

7639
```
double cosh(double x);
```
7640
```
float coshf(float x);
```
7641
```
long double coshl(long double x);
```

7642 **DESCRIPTION**

7643 CX The functionality described on this reference page is aligned with the ISO C standard. Any
7644 conflict between the requirements described here and the ISO C standard is unintentional. This
7645 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

7646 These functions shall compute the hyperbolic cosine of *x*.

7647 An application wishing to check for error situations should set *errno* to 0 before calling *cosh*( ). If
7648 *errno* is non-zero on return, or the returned value is NaN, an error has occurred.

7649 **RETURN VALUE**

7650 Upon successful completion, these functions shall return the hyperbolic cosine of *x*.

7651 If the result would cause an overflow, HUGE_VAL shall be returned and *errno* set to [ERANGE].

7652 XSI If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

7653 **ERRORS**

7654 These functions shall fail if:

7655 [ERANGE] The result would cause an overflow.

7656 These functions may fail if:

7657 XSI [EDOM] The value of *x* is NaN.

7658 XSI No other errors shall occur.

7659 **EXAMPLES**

7660 None.

7661 **APPLICATION USAGE**

7662 None.

7663 **RATIONALE**

7664 None.

7665 **FUTURE DIRECTIONS**

7666 None.

7667 **SEE ALSO**

7668 *acosh*( ), *isnan*( ), *sinh*( ), *tanh*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

7669 **CHANGE HISTORY**

7670 First released in Issue 1. Derived from Issue 1 of the SVID.

7671 **Issue 4**

7672 References to *matherr*( ) are removed.

7673 The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the
7674 ISO C standard and to rationalize error handling in the mathematics functions.

The return value specified for [EDOM] is marked as an extension.

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.

7681 **NAME**

7682 cpow, cpowf, cpowl — complex power functions

7683 **SYNOPSIS**

7684
```
#include <complex.h>
```

7685
```
double complex cpow(double complex x, double complex y);
```
7686
```
float complex cpowf(float complex x, float complex y);
```
7687
```
long double complex cpowl(long double complex x,
```
7688
```
    long double complex y);
```

7689 **DESCRIPTION**

7690 CX The functionality described on this reference page is aligned with the ISO C standard. Any
7691 conflict between the requirements described here and the ISO C standard is unintentional. This
7692 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

7693 These functions shall compute the complex power function x^y, with a branch cut for the first
7694 parameter along the negative real axis.

7695 **RETURN VALUE**

7696 These functions shall return the complex power function value.

7697 **ERRORS**

7698 No errors are defined.

7699 **EXAMPLES**

7700 None.

7701 **APPLICATION USAGE**

7702 None.

7703 **RATIONALE**

7704 None.

7705 **FUTURE DIRECTIONS**

7706 None.

7707 **SEE ALSO**

7708 *cabs*( ), *csqrt*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

7709 **CHANGE HISTORY**

7710 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

cproj, cprojf, cprojl — complex projection functions

SYNOPSIS

```
#include <complex.h>

double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(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 a projection of *z* onto the Riemann sphere: *z* projects to *z*, except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If *z* has an infinite part, then *cproj*(*z*) is equivalent to:

```
INFINITY + I * copysign(0.0, cimag(z))
```

RETURN VALUE

These functions shall return the value of the projection onto the Riemann sphere.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Two topologies are commonly used in complex mathematics: the complex plane with its continuum of infinities, and the Riemann sphere with its single infinity. The complex plane is better suited for transcendental functions, the Riemann sphere for algebraic functions. The complex types with their multiplicity of infinities provide a useful (though imperfect) model for the complex plane. The *cproj*( ) function helps model the Riemann sphere by mapping all infinities to one, and should be used just before any operation, especially comparisons, that might give spurious results for any of the other infinities. Note that a complex value with one infinite part and one NaN part is regarded as an infinity, not a NaN, because if one part is infinite, the complex value is infinite independent of the value of the other part. For the same reason, *cabs*( ) returns an infinity if its argument has an infinite part and a NaN part.

FUTURE DIRECTIONS

None.

SEE ALSO

carg( ), *cimag*( ), *conj*( ), *creal*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

creal, crealf, creall — complex real functions

SYNOPSIS

```
#include <complex.h>

double creal(double complex z);
float crealf(float complex z);
long double creall(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 real part of *z*.

RETURN VALUE

These functions shall return the real part value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

For a variable *z* of complex type:

```
z == creal(z) + cimag(z)*I
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

carg( ), *cimag*( ), *conj*( ), *cproj*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

creat — create a new file or rewrite an existing one

SYNOPSIS

OH
```
#include <sys/stat.h>
#include <fcntl.h>

int creat(const char *path, mode_t mode);
```

DESCRIPTION

The function call:

```
creat(path, mode)
```

is equivalent to:

```
open(path, O_WRONLY|O_CREAT|O_TRUNC, mode)
```

RETURN VALUE

Refer to *open*( ).

ERRORS

Refer to *open*( ).

EXAMPLES

Creating a File

The following example creates the file **/tmp/file** with read and write permissions for the file owner and read permission for group and others. The resulting file descriptor is assigned to the *fd* variable.

```
#include <fcntl.h>
...
int fd;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
char *filename = "/tmp/file";
...
fd = creat(filename, mode);
...
```

APPLICATION USAGE

None.

RATIONALE

The *creat*( ) function is redundant. Its services are also provided by the *open*( ) function. It has been included primarily for historical purposes since many existing applications depend on it. It is best considered a part of the C binding rather than a function that should be provided in other languages.

FUTURE DIRECTIONS

None.

SEE ALSO

open( ), 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 4

The **<sys/types.h>** and **<sys/stat.h>** headers are now marked as optional (OH); these headers need not be included on XSI-conformant systems.

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument *path* is changed from **char*** to **const char***.

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.

NAME

crypt — string encoding function (**CRYPT**)

SYNOPSIS

XSI

```
#include <unistd.h>

char *crypt(const char *key, const char *salt);
```

DESCRIPTION

The *crypt*( ) function is a string encoding function. The algorithm is implementation-defined.

The *key* argument points to a string to be encoded. The *salt* argument is a string chosen from the set:

```
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
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
0 1 2 3 4 5 6 7 8 9 . /
```

The first two characters of this string may be used to perturb the encoding algorithm.

The return value of *crypt*( ) points to static data that is overwritten by each call.

The *crypt*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *crypt*( ) shall return a pointer to the encoded string. The first two characters of the returned value are those of the *salt* argument. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

ERRORS

The *crypt*( ) function shall fail if:

[ENOSYS] The functionality is not supported on this implementation.

EXAMPLES

Encoding Passwords

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 is used to generate 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.

The *putpwent*( ) function, used in the following example, is not part of IEEE Std. 1003.1-200x.

```
#include <unistd.h>
#include <pwd.h>
#include <string.h>
#include <stdio.h>
...
int valid_change;
int pfd;  /* Integer for file descriptor returned by open(). */
FILE *fpfd;  /* File pointer for use in putpwent(). */
struct passwd *p;
char user[100];
char oldpasswd[100];
char newpasswd[100];
```

```
        char savepasswd[100];
        ...
        valid_change = 0;
        while ((p = getpwent()) != NULL) {
            /* Change entry if found. */
            if (strcmp(p->pw_name, user) == 0) {
                if (strcmp(p->pw_passwd, crypt(oldpasswd, p->pw_passwd)) == 0) {
                    strcpy(savepasswd, crypt(newpasswd, user));
                    p->pw_passwd = savepasswd;
                    valid_change = 1;
                }
                else {
                    fprintf(stderr, "Old password is not valid\n");
                }
            }
            /* Put passwd entry into ptmp. */
            putpwent(p, fpfd);
        }
```

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 4

The **<unistd.h>** header is added to the SYNOPSIS section.

The type of arguments *key* and *salt* are changed from **char*** to **const char***.

The DESCRIPTION now explicitly defines the characters that can appear in the *salt* argument.

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.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

casin( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

7943 **NAME**

7944 csinh, csinhf, csinhl — complex hyperbolic sine functions

7945 **SYNOPSIS**

7946
```
#include <complex.h>
```

7947
```
double complex csinh(double complex z);
```
7948
```
float complex csinhf(float complex z);
```
7949
```
long double complex csinhl(long double complex z);
```

7950 **DESCRIPTION**

7951 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.

7954 These functions shall compute the complex hyperbolic sine of *z*.

7955 **RETURN VALUE**

7956 These functions shall return the complex hyperbolic sine value.

7957 **ERRORS**

7958 No errors are defined.

7959 **EXAMPLES**

7960 None.

7961 **APPLICATION USAGE**

7962 None.

7963 **RATIONALE**

7964 None.

7965 **FUTURE DIRECTIONS**

7966 None.

7967 **SEE ALSO**

7968 *casinh*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

7969 **CHANGE HISTORY**

7970 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

csqrt, csqrtf, csqrtl — complex square root functions

SYNOPSIS

```
#include <complex.h>

double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(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 square root of *z*, with a branch cut along the negative real axis.

RETURN VALUE

These functions shall return the complex square root value, in the range of the right half-plane (including the imaginary axis).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cabs( ), *cpow*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

8001 **NAME**

8002 ctan, ctanf, ctanl — complex tangent functions

8003 **SYNOPSIS**

8004
```
#include <complex.h>
```

8005
```
double complex ctan(double complex z);
```
8006
```
float complex ctanf(float complex z);
```
8007
```
long double complex ctanl(long double complex z);
```

8008 **DESCRIPTION**

8009 CX The functionality described on this reference page is aligned with the ISO C standard. Any
8010 conflict between the requirements described here and the ISO C standard is unintentional. This
8011 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

8012 These functions shall compute the complex tangent of *z*.

8013 **RETURN VALUE**

8014 These functions shall return the complex tangent value.

8015 **ERRORS**

8016 No errors are defined.

8017 **EXAMPLES**

8018 None.

8019 **APPLICATION USAGE**

8020 None.

8021 **RATIONALE**

8022 None.

8023 **FUTURE DIRECTIONS**

8024 None.

8025 **SEE ALSO**

8026 *catan*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

8027 **CHANGE HISTORY**

8028 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

ctanh, ctanhf, ctanhl — complex hyperbolic tangent functions

SYNOPSIS

```
#include <complex.h>

double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(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 hyperbolic tangent of *z*.

RETURN VALUE

These functions shall return the complex hyperbolic tangent value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

catanh( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

8057 **NAME**

8058 ctermid — generate a path name for controlling terminal

8059 **SYNOPSIS**

8060
```
#include <stdio.h>
```

8061
```
char *ctermid(char *s);
```

8062 **DESCRIPTION**

8063 The *ctermid*( ) function shall generate a string that, when used as a path name, refers to the
8064 current controlling terminal for the current process. If *ctermid*( ) returns a path name, access to
8065 the file is not guaranteed.

8066 If the application uses any of the _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS
8067 functions, it shall ensure that the *ctermid*( ) function is called with a non-NULL parameter.

8068 **RETURN VALUE**

8069 If *s* is a null pointer, the string is generated in an area that may be static (and therefore may be
8070 overwritten by each call), the address of which shall be returned. Otherwise, *s* is assumed to |
8071 point to a character array of at least {L_ctermid} bytes; the string is placed in this array and the |
8072 value of *s* shall be returned. The symbolic constant {L_ctermid} is defined in **<stdio.h>**, and shall |
8073 have a value greater than 0.

8074 The *ctermid*( ) function shall return an empty string if the path name that would refer to the
8075 controlling terminal cannot be determined, or if the function is unsuccessful.

8076 **ERRORS**

8077 No errors are defined.

8078 **EXAMPLES**

8079 **Determining the Controlling Terminal for the Current Process**

8080 The following example returns a pointer to a string that identifies the controlling terminal for the
8081 current process. The path name for the terminal is stored in the array pointed to by the *ptr* |
8082 argument, which has a size of {L_ctermid} bytes, as indicated by the *term* argument. |

8083
```
#include <stdio.h>
```
8084
```
...
```
8085
```
char term[L_ctermid];
```
8086
```
char *ptr;
```

8087
```
ptr = ctermid(term);
```

8088 **APPLICATION USAGE**

8089 The difference between *ctermid*( ) and *ttyname*( ) is that *ttyname*( ) must be handed a file
8090 descriptor and return a path of the terminal associated with that file descriptor, while *ctermid*( )
8091 returns a string (such as `"/dev/tty"`) that refers to the current controlling terminal if used as a
8092 path name.

8093 **RATIONALE**

8094 {L_ctermid} must be defined appropriately for a given implementation and must be greater than |
8095 zero so that array declarations using it are accepted by the compiler. The value includes the
8096 terminating null byte.

8097 Portable applications that use threads cannot call *ctermid*( ) with NULL as the parameter if either
8098 _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS is defined. If *s* is not NULL, the
8099 *ctermid*( ) function generates a string that, when used as a path name, refers to the current
8100 controlling terminal for the current process. If *s* is NULL, the return value of *ctermid*( ) is

undefined.

If the *ctermid*( ) function returns a path name, access to the file is not guaranteed.

There is no additional burden on the programmer—changing to use a hypothetical thread-safe version of *ctermid*( ) along with allocating a buffer is more of a burden than merely allocating a buffer. Application code should not assume that the returned string is short, as some implementations have more than two path name components before reaching a logical device name.

FUTURE DIRECTIONS

None.

SEE ALSO

ttyname( ), 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 4

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The DESCRIPTION and RETURN VALUE sections, though functionally identical to Issue 3, are rewritten.

Issue 5

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

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

ctime, ctime_r — convert a time value to date and time string

SYNOPSIS

```
#include <time.h>

char *ctime(const time_t *clock);
TSF char *ctime_r(const time_t *clock, char *buf);
```

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 *ctime*( ) function shall convert the time pointed to by *clock*, representing time in seconds since the Epoch, to local time in the form of a string. It is equivalent to:

```
asctime(localtime(clock))
```

CX The *asctime*( ), *ctime*( ), *gmtime*( ), and *localtime*( ) functions return values in one of two static objects: a broken-down time structure and an array of **char**. Execution of any of the functions may overwrite the information returned in either of these objects by any of the other functions.

The *ctime*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *ctime_r*( ) function shall convert the calendar time pointed to by *clock* to local time in exactly the same form as *ctime*( ) and puts the string into the array pointed to by *buf* (which contains at least 26 bytes) and return *buf*.

Unlike *ctime*( ), the thread-safe version *ctime_r*( ) is not required to set *tzname*.

RETURN VALUE

The *ctime*( ) function shall return the pointer returned by *asctime*( ) with that broken-down time as an argument.

TSF Upon successful completion, *ctime_r*( ) shall return a pointer to the string pointed to by *buf*. When an error is encountered, a null pointer shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Values for the broken-down time structure can be obtained by calling *gmtime*( ) or *localtime*( ). The *ctime*( ) function is included for compatibility with older implementations, and does not support localized date and time formats. Applications should use the *strftime*( ) function to achieve maximum portability.

The *ctime_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

asctime( ), *clock*( ), *difftime*( ), *gmtime*( ), *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 4

The APPLICATION USAGE section is expanded to describe the time-handling functions generally and to refer users to *strftime*( ), which is a locale-dependent time-handling function.

The following change is incorporated for alignment with the ISO C standard:

- The type of argument *clock* is changed from **time_t*** to **const time_t***.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

The *ctime_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *ctime*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

Extensions beyond the ISO C standard are now marked.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

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.

8185 **NAME**

8186 daylight — daylight savings time flag

8187 **SYNOPSIS**

8188 XSI
```
#include <time.h>
```

8189
```
extern int daylight;
```

8190

8191 **DESCRIPTION**

8192 Refer to *tzset*( ).

8193 **NAME**

8194 dbm_clearerr, dbm_close, dbm_delete, dbm_error, dbm_fetch, dbm_firstkey, dbm_nextkey,
8195 dbm_open, dbm_store — database functions

8196 **SYNOPSIS**

8197 XSI
```
#include <ndbm.h>

int dbm_clearerr(DBM *db);
void dbm_close(DBM *db);
int dbm_delete(DBM *db, datum key);
int dbm_error(DBM *db);
datum dbm_fetch(DBM *db, datum key);
datum dbm_firstkey(DBM *db);
datum dbm_nextkey(DBM *db);
DBM *dbm_open(const char *file, int open_flags, mode_t file_mode);
int dbm_store(DBM *db, datum key, datum content, int store_mode);
```
8207

8208 **DESCRIPTION**

8209 These functions create, access, and modify a database.

8210 A **datum** consists of at least two members, *dptr* and *dsize*. The *dptr* member points to an object
8211 that is *dsize* bytes in length. Arbitrary binary data, as well as character strings, may be stored in
8212 the object pointed to by *dptr*.

8213 The database is stored in two files. One file is a directory containing a bit map of keys and has
8214 **.dir** as its suffix. The second file contains all data and has **.pag** as its suffix.

8215 The *dbm_open*( ) function shall open a database. The *file* argument to the function is the path
8216 name of the database. The function opens two files named *file***.dir** and *file***.pag**. The *open_flags*
8217 argument has the same meaning as the *flags* argument of *open*( ) except that a database opened
8218 for write-only access opens the files for read and write access and the behavior of the
8219 O_APPEND flag is unspecified. The *file_mode* argument has the same meaning as the third
8220 argument of *open*( ).

8221 The *dbm_close*( ) function shall close a database. The application shall ensure that argument *db* is
8222 a pointer to a **dbm** structure that has been returned from a call to *dbm_open*( ).

8223 The *dbm_fetch*( ) function shall read a record from a database. The argument *db* is a pointer to a
8224 database structure that has been returned from a call to *dbm_open*( ). The argument *key* is a
8225 **datum** that has been initialized by the application to the value of the key that matches the key of
8226 the record the program is fetching.

8227 The *dbm_store*( ) function shall write a record to a database. The argument *db* is a pointer to a
8228 database structure that has been returned from a call to *dbm_open*( ). The argument *key* is a
8229 **datum** that has been initialized by the application to the value of the key that identifies (for
8230 subsequent reading, writing, or deleting) the record the application is writing. The argument
8231 *content* is a **datum** that has been initialized by the application to the value of the record the
8232 program is writing. The argument *store_mode* controls whether *dbm_store*( ) replaces any pre-
8233 existing record that has the same key that is specified by the *key* argument. The application shall
8234 set *store_mode* to either DBM_INSERT or DBM_REPLACE. If the database contains a record that
8235 matches the *key* argument and *store_mode* is DBM_REPLACE, the existing record is replaced with
8236 the new record. If the database contains a record that matches the *key* argument and *store_mode*
8237 is DBM_INSERT, the existing record is left unchanged and the new record ignored. If the
8238 database does not contain a record that matches the *key* argument and *store_mode* is either
8239 DBM_INSERT or DBM_REPLACE, the new record is inserted in the database.

The application shall ensure that the sum of the sizes of a key/content pair does not exceed the internal block size. Moreover, the application shall ensure that all key/content pairs that hash together fit on a single block. The *dbm_store*( ) function shall return an error in the event that a disk block fills with inseparable data.

The *dbm_delete*( ) function shall delete a record and its key from the database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ). The argument *key* is a **datum** that has been initialized by the application to the value of the key that identifies the record the program is deleting.

The *dbm_firstkey*( ) function shall return the first key in the database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ).

The *dbm_nextkey*( ) function shall return the next key in the database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ). The application shall ensure that the *dbm_firstkey*( ) function is called before calling *dbm_nextkey*( ). Subsequent calls to *dbm_nextkey*( ) return the next key until all of the keys in the database have been returned.

The *dbm_error*( ) function shall return the error condition of the database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ).

The *dbm_clearerr*( ) function shall clear the error condition of the database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ).

These database functions shall support an internal block size large enough to support key/content pairs of at least 1 023 bytes.

The *dptr* pointers returned by these functions may point into static storage that may be changed by subsequent calls.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *dbm_store*( ) and *dbm_delete*( ) functions shall return 0 when they succeed and a negative value when they fail.

The *dbm_store*( ) function shall return 1 if it is called with a *flags* value of DBM_INSERT and the function finds an existing record with the same key.

The *dbm_error*( ) function shall return 0 if the error condition is not set and return a non-zero value if the error condition is set.

The return value of *dbm_clearerr*( ) is unspecified.

The *dbm_firstkey*( ) and *dbm_nextkey*( ) functions shall return a key **datum**. When the end of the database is reached, the *dptr* member of the key is a null pointer. If an error is detected, the *dptr* member of the key shall be a null pointer and the error condition of the database shall be set.

The *dbm_fetch*( ) function shall return a content **datum**. If no record in the database matches the key or if an error condition has been detected in the database, the *dptr* member of the content shall be a null pointer.

The *dbm_open*( ) function shall return a pointer to a database structure. If an error is detected during the operation, *dbm_open*( ) shall return a (**DBM***)0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The following code can be used to traverse the database:

```
for(key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))
```

The *dbm_* functions provided in this library should not be confused in any way with those of a general-purpose database management system. These functions do not provide for multiple search keys per entry, they do not protect against multi-user access (in other words they do not lock records or files), and they do not provide the many other useful database functions that are found in more robust database management systems. Creating and updating databases by use of these functions is relatively slow because of data copies that occur upon hash collisions. These functions are useful for applications requiring fast lookup of relatively static information that is to be indexed by a single key.

The *dbm_delete*( ) function need not physically reclaim file space, although it does make it available for reuse by the database.

After calling *dbm_store*( ) or *dbm_delete*( ) during a pass through the keys by *dbm_firstkey*( ) and *dbm_nextkey*( ), the application should reset the database by calling *dbm_firstkey*( ) before again calling *dbm_nextkey*( ). The contents of these files are unspecified and may not be portable.

RATIONALE

None.

FUTURE DIRECTIONS

None.

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 DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

8316 **NAME**

8317 difftime — compute the difference between two calendar time values

8318 **SYNOPSIS**

8319 `#include <time.h>`

8320 `double difftime(time_t time1, time_t time0);`

8321 **DESCRIPTION**

8322 CX The functionality described on this reference page is aligned with the ISO C standard. Any
8323 conflict between the requirements described here and the ISO C standard is unintentional. This
8324 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

8325 The *difftime*( ) function shall compute the difference between two calendar times (as returned by
8326 *time*( )): *time1*– *time0*.

8327 **RETURN VALUE**

8328 The *difftime*( ) function shall return the difference expressed in seconds as a type **double**.

8329 **ERRORS**

8330 No errors are defined.

8331 **EXAMPLES**

8332 None.

8333 **APPLICATION USAGE**

8334 None.

8335 **RATIONALE**

8336 None.

8337 **FUTURE DIRECTIONS**

8338 None.

8339 **SEE ALSO**

8340 *asctime*( ), *clock*( ), *ctime*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ), |
8341 the Base Definitions volume of IEEE Std. 1003.1-200x, **<time.h>** |

8342 **CHANGE HISTORY**

8343 First released in Issue 4. Derived from the ISO C standard. |

NAME

dirname — report the parent directory name of a file path name

SYNOPSIS

XSI
```
#include <libgen.h>

char *dirname(char *path);
```

DESCRIPTION

The *dirname*( ) function shall take a pointer to a character string that contains a path name, and return a pointer to a string that is a path name of the parent directory of that file. Trailing '/' characters in the path are not counted as part of the path.

If *path* does not contain a '/', then *dirname*( ) shall return a pointer to the string ".". If *path* is a null pointer or points to an empty string, *dirname*( ) shall return a pointer to the string ".".

The *dirname*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *dirname*( ) function shall return a pointer to a string that is the parent directory of *path*. If *path* is a null pointer or points to an empty string, a pointer to a string "." is returned.

The *dirname*( ) function may modify the string pointed to by *path*, and may return a pointer to static storage that may then be overwritten by subsequent calls to *dirname*( ).

ERRORS

No errors are defined.

EXAMPLES

The following code fragment reads a path name, changes the current working directory to the parent directory, and opens the file.

```
char path[MAXPATHLEN], *pathcopy;
int fd;
fgets(path, MAXPATHLEN, stdin);
pathcopy = strdup(path);
chdir(dirname(pathcopy));
fd = open(basename(path), O_RDONLY);
```

Sample Input and Output Strings for dirname( )

In the following table, the input string is the value pointed to by *path*, and the output string is the return value of the *dirname*( ) function.

Input String	Output String
"/usr/lib"	"/usr"
"/usr/"	"/"
"usr"	"."
"/"	"/"
"."	"."
".."	"."

Changing the Current Directory to the Parent Directory

The following program fragment reads a path name, changes the current working directory to the parent directory, and opens the file.

```
#include <unistd.h>
#include <limits.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <libgen.h>
...
char path[PATH_MAX], *pathcopy;
int fd;
...
fgets(path, PATH_MAX, stdin);
pathcopy = strdup(path);
chdir(dirname(pathcopy));
fd = open(basename(path), O_RDONLY);
```

APPLICATION USAGE

The *dirname*( ) and *basename*( ) functions together yield a complete path name. The expression *dirname*(*path*) obtains the path name of the directory where *basename*(*path*) is found.

Since the meaning of the leading `"//"` is implementation-defined, *dirname*("**//foo**) may return either `"//"` or `'//'` (but nothing else).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

basename( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<libgen.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 this function need not be reentrant is added to the DESCRIPTION.

8419 **NAME**

8420 div — compute the quotient and remainder of an integer division

8421 **SYNOPSIS**

8422 ```
#include <stdlib.h>
```

8423 ```
div_t div(int numer, int denom);
```

8424 **DESCRIPTION**

8425 CX The functionality described on this reference page is aligned with the ISO C standard. Any
8426 conflict between the requirements described here and the ISO C standard is unintentional. This
8427 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

8428 The *div*( ) function shall compute the quotient and remainder of the division of the numerator
8429 *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the integer
8430 of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be
8431 represented, the behavior is undefined; otherwise, *quot***denom*+*rem* shall equal *numer*.

8432 **RETURN VALUE**

8433 The *div*( ) function shall return a structure of type **div_t**, comprising both the quotient and the
8434 remainder. The structure includes the following members, in any order:

8435 ```
int quot; /* quotient */
```
8436 ```
int  rem;   /* remainder */
```

8437 **ERRORS**

8438 No errors are defined.

8439 **EXAMPLES**

8440 None.

8441 **APPLICATION USAGE**

8442 None.

8443 **RATIONALE**

8444 None.

8445 **FUTURE DIRECTIONS**

8446 None.

8447 **SEE ALSO**

8448 *ldiv*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>** |

8449 **CHANGE HISTORY**

8450 First released in Issue 4. Derived from the ISO C standard. |

8451 **NAME**

8452 dlclose — close a *dlopen*( ) object

8453 **SYNOPSIS**

8454 XSI
```
#include <dlfcn.h>
```

8455
```
int dlclose(void *handle);
```

8456

8457 **DESCRIPTION**

8458 The *dlclose*( ) function is used to inform the system that the object referenced by a *handle* returned
8459 from a previous *dlopen*( ) invocation is no longer needed by the application.

8460 The use of *dlclose*( ) reflects a statement of intent on the part of the process, but does not create
8461 any requirement upon the implementation, such as removal of the code or symbols referenced
8462 by *handle*. Once an object has been closed using *dlclose*( ) an application should assume that its
8463 symbols are no longer available to *dlsym*( ). All objects loaded automatically as a result of
8464 invoking *dlopen*( ) on the referenced object are also closed if this is the last reference to it.

8465 Although a *dlclose*( ) operation is not required to remove structures from an address space,
8466 neither is an implementation prohibited from doing so. The only restriction on such a removal is
8467 that no object shall be removed to which references have been relocated, until or unless all such
8468 references are removed. For instance, an object that had been loaded with a *dlopen*( ) operation
8469 specifying the RTLD_GLOBAL flag might provide a target for dynamic relocations performed in
8470 the processing of other objects—in such environments, an application may assume that no
8471 relocation, once made, shall be undone or remade unless the object requiring the relocation has
8472 itself been removed.

8473 **RETURN VALUE**

8474 If the referenced object was successfully closed, *dlclose*( ) shall return 0. If the object could not be
8475 closed, or if *handle* does not refer to an open object, *dlclose*( ) shall return a non-zero value. More
8476 detailed diagnostic information shall be available through *dlerror*( ).

8477 **ERRORS**

8478 No errors are defined.

8479 **EXAMPLES**

8480 The following example illustrates use of *dlopen*( ) and *dlclose*( ):

```
8481 ...
8482 /* Open a dynamic library and then close it ... */

8483 #include <dlfcn.h>
8484 void *mylib;
8485 int eret;

8486 mylib = dlopen("mylib.so.1", RTLD_LAZY);
8487 ...
8488 eret = dlclose(mylib);
8489 ...
```

8490 **APPLICATION USAGE**

8491 A portable application should employ a *handle* returned from a *dlopen*( ) invocation only within a
8492 given scope bracketed by the *dlopen*( ) and *dlclose*( ) operations. Implementations are free to use
8493 reference counting or other techniques such that multiple calls to *dlopen*( ) referencing the same
8494 object may return the same object for *handle*. Implementations are also free to reuse a *handle*.
8495 For these reasons, the value of a *handle* must be treated as an opaque object by the application,
8496 used only in calls to *dlsym*( ) and *dlclose*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

dlerror( ), *dlopen*( ), *dlsym*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<dlfcn.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

The DESCRIPTION is updated to say that the referenced object is closed ‘‘if this is the last reference to it’’.

8508 **NAME**

8509 dlerror — get diagnostic information

8510 **SYNOPSIS**

8511 XSI
```
#include <dlfcn.h>
```
8512
```
char *dlerror(void);
```
8513

8514 **DESCRIPTION**

8515 The *dlerror*( ) function shall return a null-terminated character string (with no trailing <newline>)
8516 that describes the last error that occurred during dynamic linking processing. If no dynamic
8517 linking errors have occurred since the last invocation of *dlerror*( ), *dlerror*( ) shall return NULL.
8518 Thus, invoking *dlerror*( ) a second time, immediately following a prior invocation, shall result in
8519 NULL being returned.

8520 The *dlerror*( ) function need not be reentrant. A function that is not required to be reentrant is not
8521 required to be thread-safe.

8522 **RETURN VALUE**

8523 If successful, *dlerror*( ) shall return a null-terminated character string; otherwise, NULL shall be
8524 returned.

8525 **ERRORS**

8526 No errors are defined.

8527 **EXAMPLES**

8528 The following example prints out the last dynamic linking error:

8529
```
...
```
8530
```
#include <dlfcn.h>
```
8531
```
char *errstr;
```
8532
```
errstr = dlerror();
if (errstr != NULL)
printf ("A dynamic linking error occurred: (%s)\n", errstr);
```
8535
```
...
```

8536 **APPLICATION USAGE**

8537 The messages returned by *dlerror*( ) may reside in a static buffer that is overwritten on each call
8538 to *dlerror*( ). Application code should not write to this buffer. Programs wishing to preserve an
8539 error message should make their own copies of that message. Depending on the application
8540 environment with respect to asynchronous execution events, such as signals or other
8541 asynchronous computation sharing the address space, portable applications should use a critical
8542 section to retrieve the error pointer and buffer.

8543 **RATIONALE**

8544 None.

8545 **FUTURE DIRECTIONS**

8546 None.

8547 **SEE ALSO**

8548 *dlclose*( ), *dlopen*( ), *dlsym*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<dlfcn.h>**

8549 **CHANGE HISTORY**

8550 First released in Issue 5.

8551 **Issue 6**

8552 In the DESCRIPTION the note about reentrancy and thread-safety is added.

8553 **NAME**

8554 dlopen — gain access to an executable object file

8555 **SYNOPSIS**

8556 XSI
```
#include <dlfcn.h>
```

8557
```
void *dlopen(const char *file, int mode);
```

8558

8559 **DESCRIPTION**

8560 The *dlopen*( ) function shall make an executable object file specified by *file* available to the calling
8561 program. The class of files eligible for this operation and the manner of their construction are
8562 specified by the implementation, though typically such files are executable objects such as
8563 shared libraries, relocatable files, or programs. Note that some implementations permit the
8564 construction of dependencies between such objects that are embedded within files. In such
8565 cases, a *dlopen*( ) operation shall load such dependencies in addition to the object referenced by
8566 *file*. Implementations may also impose specific constraints on the construction of programs that
8567 can employ *dlopen*( ) and its related services.

8568 A successful *dlopen*( ) shall return a *handle* which the caller may use on subsequent calls to
8569 *dlsym*( ) and *dlclose*( ). The value of this *handle* should not be interpreted in any way by the caller.

8570 *file* is used to construct a path name to the object file. If *file* contains a slash character, the *file*
8571 argument is used as the path name for the file. Otherwise, *file* is used in an implementation- |
8572 defined manner to yield a path name. |

8573 If the value of *file* is 0, *dlopen*( ) shall provide a *handle* on a global symbol object. This object
8574 provides access to the symbols from an ordered set of objects consisting of the original program
8575 image file, together with any objects loaded at program start-up as specified by that process
8576 image file (for example, shared libraries), and the set of objects loaded using a *dlopen*( ) operation
8577 together with the RTLD_GLOBAL flag. As the latter set of objects can change during execution,
8578 the set identified by *handle* can also change dynamically.

8579 Only a single copy of an object file is brought into the address space, even if *dlopen*( ) is invoked
8580 multiple times in reference to the file, and even if different path names are used to reference the
8581 file.

8582 The *mode* parameter describes how *dlopen*( ) shall operate upon *file* with respect to the processing
8583 of relocations and the scope of visibility of the symbols provided within *file*. When an object is
8584 brought into the address space of a process, it may contain references to symbols whose
8585 addresses are not known until the object is loaded. These references shall be relocated before the
8586 symbols can be accessed. The *mode* parameter governs when these relocations take place and
8587 may have the following values:

8588 RTLD_LAZY — Relocations shall be performed at an implementation-defined time, |
8589 ranging from the time of the *dlopen*( ) call until the first reference to a
8590 given symbol occurs. Specifying RTLD_LAZY should improve
8591 performance on implementations supporting dynamic symbol binding as
8592 a process may not reference all of the functions in any given object. And,
8593 for systems supporting dynamic symbol resolution for normal process
8594 execution, this behavior mimics the normal handling of process
8595 execution.

8596 RTLD_NOW — All necessary relocations shall be performed when the object is first
8597 loaded. This may waste some processing if relocations are performed for
8598 functions that are never referenced. This behavior may be useful for
8599 applications that need to know as soon as an object is loaded that all |

symbols referenced during execution are available.

Any object loaded by *dlopen*( ) that requires relocations against global symbols can reference the symbols in the original process image file, any objects loaded at program start-up, from the object itself as well as any other object included in the same *dlopen*( ) invocation, and any objects that were loaded in any *dlopen*( ) invocation and which specified the RTLD_GLOBAL flag. To determine the scope of visibility for the symbols loaded with a *dlopen*( ) invocation, the *mode* parameter should be a bitwise-inclusive OR with one of the following values:

RTLD_GLOBAL — The object's symbols shall be made available for the relocation processing of any other object. In addition, symbol lookup using *dlopen*(*0*, *mode*) and an associated *dlsym*( ) allows objects loaded with this *mode* to be searched.

RTLD_LOCAL — The object's symbols shall not be made available for the relocation processing of any other object.

If neither RTLD_GLOBAL nor RTLD_LOCAL are specified, then an implementation-defined default behavior shall be applied.

If a *file* is specified in multiple *dlopen*( ) invocations, *mode* is interpreted at each invocation. Note, however, that once RTLD_NOW has been specified all relocations shall have been completed rendering further RTLD_NOW operations redundant and any further RTLD_LAZY operations irrelevant. Similarly, note that once RTLD_GLOBAL has been specified the object shall maintain the RTLD_GLOBAL status regardless of any previous or future specification of RTLD_LOCAL, as long as the object remains in the address space (see *dlclose*( )).

Symbols introduced into a program through calls to *dlopen*( ) may be used in relocation activities. Symbols so introduced may duplicate symbols already defined by the program or previous *dlopen*( ) operations. To resolve the ambiguities such a situation might present, the resolution of a symbol reference to symbol definition is based on a symbol resolution order. Two such resolution orders are defined: *load* or *dependency* ordering. Load order establishes an ordering among symbol definitions, such that the definition first loaded (including definitions from the image file and any dependent objects loaded with it) has priority over objects added later (via *dlopen*( )). Load ordering is used in relocation processing. Dependency ordering uses a breadth-first order starting with a given object, then all of its dependencies, then any dependents of those, iterating until all dependencies are satisfied. With the exception of the global symbol object obtained via a *dlopen*( ) operation on a *file* of 0, dependency ordering is used by the *dlsym*( ) function. Load ordering is used in *dlsym*( ) operations upon the global symbol object.

When an object is first made accessible via *dlopen*( ) it and its dependent objects are added in dependency order. Once all the objects are added, relocations are performed using load order. Note that if an object or its dependencies had been previously loaded, the load and dependency orders may yield different resolutions.

The symbols introduced by *dlopen*( ) operations, and available through *dlsym*( ) are at a minimum those which are exported as symbols of global scope by the object. Typically such symbols shall be those that were specified in (for example) C source code as having *extern* linkage. The precise manner in which an implementation constructs the set of exported symbols for a *dlopen*( ) object is specified by that implementation.

RETURN VALUE

If *file* cannot be found, cannot be opened for reading, is not of an appropriate object format for processing by *dlopen*( ), or if an error occurs during the process of loading *file* or relocating its symbolic references, *dlopen*( ) shall return NULL. More detailed diagnostic information shall be available through *dlerror*( ).

8646 **ERRORS**

8647 No errors are defined.

8648 **EXAMPLES**

8649 None.

8650 **APPLICATION USAGE**

8651 None.

8652 **RATIONALE**

8653 None.

8654 **FUTURE DIRECTIONS**

8655 None.

8656 **SEE ALSO**

8657 *dlclose*( ), *dlerror*( ), *dlsym*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<dlfcn.h>** |

8658 **CHANGE HISTORY**

8659 First released in Issue 5.

NAME

dlsym — obtain the address of a symbol from a *dlopen*( ) object

SYNOPSIS

XSI
```
#include <dlfcn.h>

void *dlsym(void *restrict handle, const char *restrict name);
```

DESCRIPTION

The *dlsym*( ) function allows a process to obtain the address of a symbol defined within an object made accessible through a *dlopen*( ) call. *handle* is the value returned from a call to *dlopen*( ) (and which has not since been released via a call to *dlclose*( )), and *name* is the symbol's name as a character string.

The *dlsym*( ) function shall search for the named symbol in all objects loaded automatically as a result of loading the object referenced by *handle* (see *dlopen*( )). Load ordering is used in *dlsym*( ) operations upon the global symbol object. The symbol resolution algorithm used shall be dependency order as described in *dlopen*( ).

The RTLD_NEXT flag is reserved for future use.

RETURN VALUE

If *handle* does not refer to a valid object opened by *dlopen*( ), or if the named symbol cannot be found within any of the objects associated with *handle*, *dlsym*( ) shall return NULL. More detailed diagnostic information shall be available through *dlerror*( ).

ERRORS

No errors are defined.

EXAMPLES

The following example shows how *dlopen*( ) and *dlsym*( ) can be used to access either function or data objects. For simplicity, error checking has been omitted.

```
void    *handle;
int     *iptr, (*fptr)(int);

/* open the needed object */
handle = dlopen("/usr/home/me/libfoo.so.1", RTLD_LAZY);

/* find the address of function and data objects */
fptr = (int (*)(int))dlsym(handle, "my_function");
iptr = (int *)dlsym(handle, "my_object");

/* invoke function, passing value of integer as a parameter */
(*fptr)(*iptr);
```

APPLICATION USAGE

Special purpose values for *handle* are reserved for future use. These values and their meanings are:

RTLD_NEXT — Specifies the next object after this one that defines *name*. *This one* refers to the object containing the invocation of *dlsym*( ). The *next* object is the one found upon the application of a load order symbol resolution algorithm (see *dlopen*( )). The next object is either one of global scope (because it was 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

None.

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.

NAME

drand48, erand48, jrand48, lcong48, lrand48, mrand48, nrand48, seed48, srand48 — generate uniformly distributed pseudo-random numbers

SYNOPSIS

XSI

```
#include <stdlib.h>

double drand48(void);
double erand48(unsigned short xsubi[3]);
long jrand48(unsigned short xsubi[3]);
void lcong48(unsigned short param[7]);
long lrand48(void);
long mrand48(void);
long nrand48(unsigned short xsubi[3]);
unsigned short *seed48(unsigned short seed16v[3]);
void srand48(long seedval);
```

DESCRIPTION

This family of functions generates pseudo-random numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The *drand48*( ) and *erand48*( ) functions shall return non-negative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0).

The *lrand48*( ) and *nrand48*( ) functions shall return non-negative, long integers, uniformly distributed over the interval $[0,2^{31})$.

The *mrand48*( ) and *jrand48*( ) functions shall return signed long integers uniformly distributed over the interval $[-2^{31},2^{31})$.

The *srand48*( ), *seed48*( ), and *lcong48*( ) are initialization entry points, one of which should be invoked before either *drand48*( ), *lrand48*( ), or *mrand48*( ) is called. (Although it is not recommended practice, constant default initializer values shall be supplied automatically if *drand48*( ), *lrand48*( ), or *mrand48*( ) is called without a prior call to an initialization entry point.) The *erand48*( ), *nrand48*( ), and *jrand48*( ) functions do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula:

$$X_{n+1} = (aX_n + c)_{\mathrm{mod}\ m} \qquad n \geq 0$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless *lcong48*( ) is invoked, the multiplier value a and the addend value c are given by:

$$a = 5\mathrm{DEECE66D}_{16} = 273673163155_8$$

$$c = \mathrm{B}_{16} = 13_8$$

The value returned by any of the *drand48*( ), *erand48*( ), *jrand48*( ), *lrand48*( ), *mrand48*( ), or *nrand48*( ) functions is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of X_i and transformed into the returned value.

The *drand48*( ), *lrand48*( ), and *mrand48*( ) functions store the last 48-bit X_i generated in an internal buffer; that is why the application shall ensure that these are initialized prior to being invoked. The *erand48*( ), *nrand48*( ), and *jrand48*( ) functions require the calling program to provide storage for the successive X_i values in the array specified as an argument when the

functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, *erand48*(), *nrand48*(), and *jrand48*() allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers; that is, the sequence of numbers in each stream shall *not* depend upon how many times the routines are called to generate numbers for the other streams.

The initializer function *srand48*() sets the high-order 32 bits of X_i to the low-order 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

The initializer function *seed48*() sets the value of X_i to the 48-bit value specified in the argument array. The low-order 16 bits of X_i are set to the low-order 16 bits of *seed16v*[*0*]. The mid-order 16 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 low-order 16 bits of *seed16v*[*2*]. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by *seed48*(), and a pointer to this buffer is the value returned by *seed48*(). This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time—use the pointer to get at and store the last X_i value, and then use this value to re-initialize via *seed48*() when the program is restarted.

The initializer function *lcong48*() allows the user to specify the initial X_i, the multiplier value *a*, and the addend value *c*. Argument array elements *param*[*0-2*] specify X_i, *param*[*3-5*] specify the multiplier *a*, and *param*[*6*] specifies the 16-bit addend *c*. After *lcong48*() is called, a subsequent call to either *srand48*() or *seed48*() shall restore the standard multiplier and addend values, *a* and *c*, specified above.

The *drand48*(), *lrand48*(), and *mrand48*() functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

As described in the DESCRIPTION above.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

rand(), 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 4

The type **long** is replaced by **long** and the type **unsigned short** is replaced by **unsigned short** in the SYNOPSIS section.

In the DESCRIPTION, the description of *srand48*() is amended to fix a limitation in Issue 3, which indicates that the high-order 32 bits of X_i are set to the {LONG_BIT} bits in the argument. Though unintentional, the implication of this statement is that {LONG_BIT} would be 32 on all

systems compliant with Issue 3, when in fact Issue 3 imposes no such restriction.

The **<stdlib.h>** header is added to the SYNOPSIS section.

The argument list for the *lrand48*( ) and *mrand48*( ) functions now contains **void**.

Issue 5

A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

8818 **NAME**

8819 dup, dup2 — duplicate an open file descriptor

8820 **SYNOPSIS**

8821 `#include <unistd.h>`

8822 `int dup(int fildes);`
8823 `int dup2(int fildes, int fildes2);`

8824 **DESCRIPTION**

8825 The *dup*( ) and *dup2*( ) functions provide an alternative interface to the service provided by
8826 *fcntl*( ) using the F_DUPFD command. The call:

8827 `fid = dup(fildes);`

8828 is equivalent to:

8829 `fid = fcntl(fildes, F_DUPFD, 0);`

8830 The call:

8831 `fid = dup2(fildes, fildes2);`

8832 is equivalent to:

8833 `close(fildes2);`
8834 `fid = fcntl(fildes, F_DUPFD, fildes2);`

8835 except for the following:

8836 • If *fildes2* is less than 0 or greater than or equal to {OPEN_MAX}, *dup2*( ) shall return –1 with
8837 *errno* set to [EBADF]. |

8838 • If *fildes* is a valid file descriptor and is equal to *fildes2*, *dup2*( ) shall return *fildes2* without
8839 closing it.

8840 • If *fildes* is not a valid file descriptor, *dup2*( ) shall return –1 and shall not close *fildes2*.

8841 • The value returned shall be equal to the value of *fildes2* upon successful completion, or –1
8842 upon failure.

8843 **RETURN VALUE**

8844 Upon successful completion a non-negative integer, namely the file descriptor, shall be returned;
8845 otherwise, –1 shall be returned and *errno* set to indicate the error.

8846 **ERRORS**

8847 The *dup*( ) function shall fail if:

8848 [EBADF] The *fildes* argument is not a valid open file descriptor. |

8849 [EMFILE] The number of file descriptors in use by this process would exceed |
8850 {OPEN_MAX}.

8851 The *dup2*( ) function shall fail if:

8852 [EBADF] The *fildes* argument is not a valid open file descriptor or the argument *fildes2* is |
8853 negative or greater than or equal to {OPEN_MAX}.

8854 [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 *pfd*, 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 *fildes* being out of range is covered by the given case of *fildes* not being valid. The descriptions for *fildes* and *fildes2* are different because the only kind of invalidity that is relevant for *fildes2* is whether it is out of range; that is, it does not matter whether *fildes2* 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>**

8895 **CHANGE HISTORY**

8896 First released in Issue 1. Derived from Issue 1 of the SVID.

8897 **Issue 4**

8898 The **<unistd.h>** header is added to the SYNOPSIS section.

8899 [EINTR] is no longer required for *dup*( ) because *fcntl*( ) does not return [EINTR] for F_DUPFD.

8900 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

8901 • In the DESCRIPTION, the fourth bullet item describing differences between *dup*( ) and
8902 *dup2*( ) is added.

8903 • In the ERRORS section, error values returned by *dup*( ) and *dup2*( ) are now described
8904 separately.

8905 **Issue 6**

8906 The RATIONALE section is added.

8907 **NAME**

8908 ecvt, fcvt, gcvt — convert a floating-point number to a string (**LEGACY**)

8909 **SYNOPSIS**

8910 XSI
```
#include <stdlib.h>

char *ecvt(double value, int ndigit, int *restrict decpt,
    int *restrict sign);
char *fcvt(double value, int ndigit, int *restrict decpt,
    int *restrict sign);
char *gcvt(double value, int ndigit, char *buf);
```
8916

8917 **DESCRIPTION**

8918 The *ecvt*( ), *fcvt*( ), and *gcvt*( ) functions shall convert floating-point numbers to null-terminated
8919 strings.

8920 The *ecvt*( ) function shall convert *value* to a null-terminated string of *ndigit* digits (where *ndigit* is
8921 reduced to an unspecified limit determined by the precision of a **double**) and return a pointer to
8922 the string. The high-order digit is non-zero, unless the value is 0. The low-order digit is rounded.
8923 The position of the radix character relative to the beginning of the string is stored in the integer
8924 pointed to by *decpt* (negative means to the left of the returned digits). If *value* is zero, it is
8925 unspecified whether the integer pointed to by *decpt* would be 0 or 1. The radix character is not
8926 included in the returned string. If the sign of the result is negative, the integer pointed to by *sign*
8927 is non-zero; otherwise, it is 0.

8928 If the converted value is out of range or is not representable, the contents of the returned string
8929 are unspecified.

8930 The *fcvt*( ) function is identical to *ecvt*( ) except that *ndigit* specifies the number of digits desired
8931 after the radix character. The total number of digits in the result string is restricted to an
8932 unspecified limit as determined by the precision of a **double**.

8933 The *gcvt*( ) function shall convert *value* to a null-terminated string (similar to that of the %*g*
8934 format of *printf*( )) in the array pointed to by *buf* and return *buf*. It produces *ndigit* significant
8935 digits (limited to an unspecified value determined by the precision of a **double**) in %*f* if possible,
8936 or %*e* (scientific notation) otherwise. A minus sign is included in the returned string if *value* is
8937 less than 0. A radix character is included in the returned string if *value* is not a whole number.
8938 Trailing zeros are suppressed where *value* is not a whole number. The radix character is
8939 determined by the current locale. If *setlocale*( ) has not been called successfully, the default locale,
8940 POSIX, is used. The default locale specifies a period (`'.'`) as the radix character. The
8941 *LC_NUMERIC* category determines the value of the radix character within the current locale.

8942 These functions need not be reentrant. A function that is not required to be reentrant is not
8943 required to be thread-safe.

8944 **RETURN VALUE**

8945 The *ecvt*( ) and *fcvt*( ) functions shall return a pointer to a null-terminated string of digits.

8946 The *gcvt*( ) function shall return *buf*.

8947 The return values from *ecvt*( ) and *fcvt*( ) may point to static data which may be overwritten by
8948 subsequent calls to these functions.

8949 **ERRORS**

8950 No errors are defined.

8951 **EXAMPLES**

8952 None.

8953 **APPLICATION USAGE**

8954 *sprintf*( ) is preferred over this function.

8955 **RATIONALE**

8956 None.

8957 **FUTURE DIRECTIONS**

8958 These functions may be withdrawn in a future version.

8959 **SEE ALSO**

8960 *printf*( ), *setlocale*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>** |

8961 **CHANGE HISTORY**

8962 First released in Issue 4, Version 2.

8963 **Issue 5**

8964 Moved from X/OPEN UNIX extension to BASE.

8965 Normative text previously in the APPLICATION USAGE section is moved to the
8966 DESCRIPTION.

8967 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

8968 **Issue 6**

8969 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

8970 This function is marked LEGACY. |

8971 The **restrict** keyword is added to the *ecvt*( ) and *fcvt*( ) prototypes for alignment with the |
8972 ISO/IEC 9899:1999 standard. |

NAME

encrypt — encoding function (**CRYPT**)

SYNOPSIS

XSI
```
#include <unistd.h>

void encrypt(char block[64], int edflag);
```

DESCRIPTION

The *encrypt*( ) function provides (rather primitive) 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*( ) is an array of length 64 bytes containing only the bytes with numerical value 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 reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *encrypt*( ) function shall return no value.

ERRORS

The *encrypt*( ) function shall fail if:

[ENOSYS] The functionality is not supported on this implementation.

EXAMPLES

None.

APPLICATION USAGE

In some environments, decoding might not be implemented. This is related to U.S. Government restrictions on encryption and decryption routines: the DES decryption algorithm cannot be exported outside the U.S. 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.

Issue 4

The **<unistd.h>** header is added to the SYNOPSIS section.

The DESCRIPTION is amended:

- To specify the encoding algorithm as implementation-defined
- To change entry to function
- To make decoding optional

The APPLICATION USAGE section is expanded to explain the restrictions on the availability of the DES decryption algorithm.

Issue 5

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.

NAME

endgrent, getgrent, setgrent — group database entry functions

SYNOPSIS

XSI
```
#include <grp.h>

void endgrent(void);
struct group *getgrent(void);
void setgrent(void);
```

DESCRIPTION

The *endgrent*( ) function may be called to close the group database when processing is complete.

An implementation that provides extended security controls may impose further implementation-defined restrictions on accessing the group database. In particular, the system may deny the existence of some or all of the group database entries associated with groups other than those groups associated with the caller and may omit users other than the caller from the list of members of groups in database entries that are returned.

The *getgrent*( ) function shall return a pointer to a structure containing the broken-out fields of an entry in the group database. When first called, *getgrent*( ) shall return a pointer to a **group** structure containing the first entry in the group database. Thereafter, it shall return a pointer to a **group** structure containing the next group structure in the group database, so successive calls may be used to search the entire database.

The *setgrent*( ) function effectively rewinds the group database to allow repeated searches.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

When first called, *getgrent*( ) shall return a pointer to the first group structure in the group database. Upon subsequent calls it shall return the next group structure in the group database. The *getgrent*( ) function shall return a null pointer on end-of-file or an error and *errno* may be set to indicate the error.

The return value may point to a static area which is overwritten by a subsequent call to *getgrgid*( ), *getgrnam*( ), or *getgrent*( ).

ERRORS

The *getgrent*( ) function may fail if:

[EINTR]	A signal was caught during the operation.
[EIO]	An I/O error has occurred.
[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
[ENFILE]	The maximum allowable number of files is currently open in the system.

9062 **EXAMPLES**

9063 None.

9064 **APPLICATION USAGE**

9065 These functions are provided due to their historical usage. Applications should avoid
9066 dependencies on fields in the group database, whether the database is a single file, or where in
9067 the file system name space the database resides. Applications should use *getgrnam*( ) and
9068 *getgrgid*( ) whenever possible because it avoids these dependencies.

9069 **RATIONALE**

9070 None.

9071 **FUTURE DIRECTIONS**

9072 None.

9073 **SEE ALSO**

9074 *getgrgid*( ), *getgrnam*( ), *getlogin*( ), *getpwent*( ), the Base Definitions volume of |
9075 IEEE Std. 1003.1-200x, **<grp.h>** |

9076 **CHANGE HISTORY**

9077 First released in Issue 4, Version 2.

9078 **Issue 5**

9079 Moved from X/OPEN UNIX extension to BASE.

9080 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
9081 VALUE section.

9082 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

9083 **Issue 6**

9084 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

NAME

endhostent, freehostent, gethostent, sethostent — network host database functions

SYNOPSIS

```
#include <netdb.h>

void endhostent(void);
void freehostent(struct hostent *ptr);
struct hostent *gethostent(void);
void sethostent(int stayopen);
```

DESCRIPTION

These functions enable applications to retrieve information about hosts. This information is considered to be stored in a database that can be accessed sequentially or randomly. Implementation of this database is unspecified.

Note: In many cases it is implemented by the Domain Name System, as documented in RFC 1034, RFC 1035, and RFC 1886.

Entries are returned in **hostent** structures. Refer to *gethostbyaddr*( ) for a definition of the **hostent** structure.

The *endhostent*( ) function shall close the connection to the database, releasing any open file descriptor.

The *freehostent*( ) function shall free the memory occupied by the **hostent** structure pointed to by *ptr* and any structures pointed to from that structure, provided that **hostent** was obtained by a call to *getipnodebyaddr*( ) or *getipnodebyname*( ). Applications shall not call *freehostent*( ) except to pass it a pointer that was obtained from *getipnodebyaddr*( ) or *getipnodebyname*( ).

The *gethostent*( ) function shall read the next entry in the database, opening and closing a connection to the database as necessary.

The *sethostent*( ) function shall open a connection to the database and set the next entry for retrieval to the first entry in the database. If the *stayopen* argument is non-zero, the connection shall not be closed by a call to *gethostent*( ), *getipnodebyname*( ), *gethostbyname*( ), *getipnodebyaddr*( ), or *gethostbyaddr*( ), and the implementation may maintain an open file descriptor.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, the *gethostent*( ) function shall return a pointer to a **hostent** structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found.

ERRORS

No errors are defined for *endhostent*( ), *gethostent*( ), and *sethostent*( ).

EXAMPLES

None.

APPLICATION USAGE

The **hostent** structure returned by *getipnodebyaddr*( ) and *getipnodebyname*( ), and any structures pointed to from those structures, are dynamically allocated. Applications should call *freehostent*( ) to free the memory used by these structures.

The *gethostent*( ) function may return pointers to static data, which may be overwritten by subsequent calls to any of these functions. Applications shall not call *freehostent*( ) for this area.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endservent( ), *gethostbyaddr*( ), *gethostbyname*( ), *getipnodebyaddr*( ), *getipnodebyname*( ), 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.

NAME

endnetent, getnetbyaddr, getnetbyname, getnetent, setnetent — network database functions

SYNOPSIS

```
#include <netdb.h>

void endnetent(void);
struct netent *getnetbyaddr(uint32_t net, int type);
struct netent *getnetbyname(const char *name);
struct netent *getnetent(void);
void setnetent(int stayopen);
```

DESCRIPTION

These functions enable applications to retrieve information about networks. This information is considered to be stored in a database that can be accessed sequentially or randomly. Implementation of this database is unspecified.

The *endnetent*( ) function shall close the database, releasing any open file descriptor.

The *getnetbyaddr*( ) function shall search the database from the beginning, and find the first entry for which the address family specified by *type* matches the *n_addrtype* member and the network number *net* matches the *n_net* member, opening a connection to the database if necessary. The *net* argument shall be the network number in host byte order.

The *getnetbyname*( ) function shall search the database from the beginning and find the first entry for which the network name specified by *name* matches the *n_name* member, opening and closing a connection to the database as necessary.

The *getnetent*( ) function shall read the next entry of the database, opening a connection to the database if necessary.

The *setnetent*( ) function shall open and rewind the database. If the *stayopen* argument is non-zero, the connection to the *net* database shall not be closed after each call to *getnetent*( ) (either directly, or indirectly through one of the other *getnet**( ) functions), and the implementation may maintain an open file descriptor to the database.

The *getnetbyaddr*( ), *getnetbyname*( ), and *getnetent*( ), functions shall each return a pointer to a **netent** structure, the members of which shall contain the fields of an entry in the network database.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *getnetbyaddr*( ), *getnetbyname*( ), and *getnetent*( ), shall return a pointer to a **netent** structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found. Otherwise, a null pointer shall be returned.

ERRORS

No errors are defined.

9177 **EXAMPLES**

9178 None.

9179 **APPLICATION USAGE**

9180 The *getnetbyaddr*( ), *getnetbyname*( ), and *getnetent*( ), functions may return pointers to static data,
9181 which may be overwritten by subsequent calls to any of these functions.

9182 **RATIONALE**

9183 None.

9184 **FUTURE DIRECTIONS**

9185 None.

9186 **SEE ALSO**

9187 The Base Definitions volume of IEEE Std. 1003.1-200x, **<netdb.h>**

9188 **CHANGE HISTORY**

9189 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

endprotoent, getprotobyname, getprotobynumber, getprotoent, setprotoent — network protocol database functions

SYNOPSIS

```
#include <netdb.h>

void endprotoent(void);
struct protoent *getprotobyname(const char *name);
struct protoent *getprotobynumber(int proto);
struct protoent *getprotoent(void);
void setprotoent(int stayopen);
```

DESCRIPTION

These functions enable applications to retrieve information about protocols. This information is considered to be stored in a database that can be accessed sequentially or randomly. Implementation of this database is unspecified.

The *endprotoent*( ) function shall close the connection to the database, releasing any open file descriptor.

The *getprotobyname*( ) function shall search the database from the beginning and find the first entry for which the protocol name specified by *name* matches the *p_name* member, opening a connection to the database if necessary.

The *getprotobynumber*( ) function shall search the database from the beginning and find the first entry for which the protocol number specified by *proto* matches the *p_proto* member, opening a connection to the database if necessary.

The *getprotoent*( ) function shall read the next entry of the database, opening and closing a connection to the database as necessary.

The *setprotoent*( ) function shall open a connection to the database, and set the next entry to the first entry. If the *stayopen* argument is non-zero, the connection to the network protocol database shall not be closed after each call to *getprotoent*( ) (either directly, or indirectly through one of the other *getproto**( ) functions), and the implementation may maintain an open file descriptor for the database.

The *getprotobyname*( ), *getprotobynumber*( ), and *getprotoent*( ), functions shall each return a pointer to a **protoent** structure, the members of which shall contain the fields of an entry in the network protocol database.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *getprotobyname*( ), *getprotobynumber*( ), and *getprotoent*( ) return a pointer to a **protoent** structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found. Otherwise, a null pointer is returned.

ERRORS

No errors are defined.

9231 **EXAMPLES**

9232 None.

9233 **APPLICATION USAGE**

9234 The *getprotobyname*(), *getprotobynumber*(), and *getprotoent*() functions may return pointers to
9235 static data, which may be overwritten by subsequent calls to any of these functions.

9236 **RATIONALE**

9237 None.

9238 **FUTURE DIRECTIONS**

9239 None.

9240 **SEE ALSO**

9241 The Base Definitions volume of IEEE Std. 1003.1-200x, **<netdb.h>**

9242 **CHANGE HISTORY**

9243 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

endpwent, getpwent, setpwent — user database functions

SYNOPSIS

XSI
```
#include <pwd.h>

void endpwent(void);
struct passwd *getpwent(void);
void setpwent(void);
```

DESCRIPTION

The *endpwent*( ) function may be called to close the user database when processing is complete.

An implementation that provides extended security controls may impose further implementation-defined restrictions on accessing the user database. In particular, the system may deny the existence of some or all of the user database entries associated with users other than the caller.

The *getpwent*( ) function shall return a pointer to a structure containing the broken-out fields of an entry in the user database. Each entry in the user database contains a **passwd** structure. When first called, *getpwent*( ) shall return a pointer to a **passwd** structure containing the first entry in the user database. Thereafter, it shall return a pointer to a **passwd** structure containing the next entry in the user database. Successive calls can be used to search the entire user database.

If an end-of-file or an error is encountered on reading, *getpwent*( ) shall return a null pointer.

The *setpwent*( ) function effectively rewinds the user database to allow repeated searches.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *getpwent*( ) function shall return a null pointer on end-of-file or error.

ERRORS

The *getpwent*( ), *setpwent*( ), and *endpwent*( ) functions may fail if:

[EIO] An I/O error has occurred.

In addition, *getpwent*( ) and *setpwent*( ) may fail if:

[EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

[ENFILE] The maximum allowable number of files is currently open in the system.

The return value may point to a static area which is overwritten by a subsequent call to *getpwuid*( ), *getpwnam*( ), or *getpwent*( ).

EXAMPLES

Searching the User Database

The following example uses the *getpwent*( ) function to get successive entries in the user database, returning a pointer to a **passwd** structure that contains information about each user. The call to *endpwent*( ) closes the user database and cleans up.

```
#include <pwd.h>
...
struct passwd *p;
...
```

```
        while ((p = getpwent ()) != NULL) {
        ...
        }
        endpwent();
        ...
```

APPLICATION USAGE

These functions are provided due to their historical usage. Applications should avoid dependencies on fields in the password database, whether the database is a single file, or where in the file system name space the database resides. Applications should use *getpwuid*() whenever possible because it avoids these dependencies.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endgrent(), *getlogin*(), *getpwnam*(), *getpwuid*(), the Base Definitions volume of IEEE Std. 1003.1-200x, **<pwd.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.

NAME

endservent, getservbyname, getservbyport, getservent, setservent — network services database functions

SYNOPSIS

```
#include <netdb.h>

void endservent(void);
struct servent *getservbyname(const char *name, const char *proto);
struct servent *getservbyport(int port, const char *proto);
struct servent *getservent(void);
void setservent(int stayopen);
```

DESCRIPTION

These functions enable applications to retrieve information about network services. This information is considered to be stored in a database that can be accessed sequentially or randomly. Implementation of this database is unspecified.

The *endservent*( ) function shall close the database, releasing any open file descriptor.

The *getservbyname*( ) function shall search the database from the beginning and find the first entry for which the service name specified by *name* matches the *s_name* member and the protocol name specified by *proto* matches the *s_proto* member, opening a connection to the database if necessary. If *proto* is a null pointer, any value of the *s_proto* member shall be matched.

The *getservbyport*( ) function shall search the database from the beginning and find the first entry for which the port specified by *port* matches the *s_port* member and the protocol name specified by *proto* matches the *s_proto* member, opening a connection to the database if necessary. If *proto* is a null pointer, any value of the *s_proto* member shall be matched. The *port* argument shall be in network byte order.

The *getservent*( ) function shall read the next entry of the database, opening and closing a connection to the database as necessary.

The *setservent*( ) function shall open a connection to the database, and set the next entry to the first entry. If the *stayopen* argument is non-zero, the *net* database shall not be closed after each call to the *getservent*( ) function (either directly, or indirectly through one of the other *getserv**( ) functions), and the implementation may maintain an open file descriptor for the database.

The *getservbyname*( ), *getservbyport*( ), and *getservent*( ) functions shall each return a pointer to a **servent** structure, the members of which shall contain the fields of an entry in the network services database.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *getservbyname*( ), *getservbyport*( ), and *getservent*( ) return a pointer to a **servent** structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found. Otherwise, a null pointer is returned.

ERRORS

No errors are defined.

9353 **EXAMPLES**

9354 None.

9355 **APPLICATION USAGE**

9356 The *port* argument of *getservbyport*( ) need not be compatible with the port values of all address
9357 families.

9358 The *getservbyname*( ), *getservbyport*( ), and *getservent*( ) functions may return pointers to static
9359 data, which may be overwritten by subsequent calls to any of these functions.

9360 **RATIONALE**

9361 None.

9362 **FUTURE DIRECTIONS**

9363 None.

9364 **SEE ALSO**

9365 *endhostent*( ), *endprotoent*( ), *htonl*( ), *inet_addr*( ), the Base Definitions volume of |
9366 IEEE Std. 1003.1-200x, **<netdb.h>** |

9367 **CHANGE HISTORY**

9368 First released in Issue 6. Derived from the XNS, Issue 5.2 specification. |

NAME

endutxent, getutxent, getutxid, getutxline, pututxline, setutxent — user accounting database functions

SYNOPSIS

XSI
```
#include <utmpx.h>

void endutxent(void);
struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *id);
struct utmpx *getutxline(const struct utmpx *line);
struct utmpx *pututxline(const struct utmpx *utmpx);
void setutxent(void);
```

DESCRIPTION

These functions provide access to the user accounting database.

The *endutxent*( ) function closes the user accounting database.

An implementation that provides extended security controls may impose further 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.

The *getutxent*( ) function reads in the next entry from the user accounting database. If the database is not already open, it opens it. If it reaches the end of the database, it fails.

The *getutxid*( ) function searches forward from the current point in the database. If the *ut_type* value of the **utmpx** structure pointed to by *id* is BOOT_TIME, OLD_TIME, or NEW_TIME, then it stops when it finds an entry with a matching *ut_type* value. If the *ut_type* value is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS, or DEAD_PROCESS, then it stops when it finds an entry whose type is one of these four and whose *ut_id* member matches the *ut_id* member of the **utmpx** structure pointed to by *id*. If the end of the database is reached without a match, *getutxid*( ) fails.

The *getutxid*( ) or *getutxline*( ) function may cache data. For this reason, to use *getutxline*( ) to search for multiple occurrences, it is necessary to zero out the static data after each success, or *getutxline*( ) could just return a pointer to the same **utmpx** structure over and over again.

There is one exception to the rule about removing the structure before further reads are done. The implicit read done by *pututxline*( ) (if it finds that it is not already at the correct place in the user accounting database) shall not modify the static structure returned by *getutxent*( ), *getutxid*( ), or *getutxline*( ), if the application has just modified this structure and passed the pointer back to *pututxline*( ).

For all entries that match a request, the *ut_type* member indicates the type of the entry. Other members of the entry shall contain meaningful data based on the value of the *ut_type* member as follows:

ut_type Member	Other Members with Meaningful Data
EMPTY	No others
BOOT_TIME	*ut_tv*
OLD_TIME	*ut_tv*
NEW_TIME	*ut_tv*
USER_PROCESS	*ut_id*, *ut_user* (login name of the user), *ut_line*, *ut_pid*, *ut_tv*
INIT_PROCESS	*ut_id*, *ut_pid*, *ut_tv*
LOGIN_PROCESS	*ut_id*, *ut_user* (implementation-defined name of the login process), *ut_pid*, *ut_tv*
DEAD_PROCESS	*ut_id*, *ut_pid*, *ut_tv*

The *getutxline*( ) function searches forward from the current point in the database until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS which also has a *ut_line* value matching that in the **utmpx** structure pointed to by *line*. If the end of the database is reached without a match, *getutxline*( ) fails.

If the process has appropriate privileges, the *pututxline*( ) function writes out the structure into the user accounting database. It uses *getutxid*( ) to search for a record that satisfies the request. If this search succeeds, then the entry is replaced. Otherwise, a new entry is made at the end of the user accounting database.

The *setutxent*( ) function resets 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 reentrant. A function that is not required to be reentrant is not required to be thread-safe.

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 return no 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.

9446 **EXAMPLES**

9447 None.

9448 **APPLICATION USAGE**

9449 The sizes of the arrays in the structure can be found using the *sizeof* operator.

9450 **RATIONALE**

9451 None.

9452 **FUTURE DIRECTIONS**

9453 None.

9454 **SEE ALSO**

9455 The Base Definitions volume of IEEE Std. 1003.1-200x, **<utmpx.h>**

9456 **CHANGE HISTORY**

9457 First released in Issue 4, Version 2.

9458 **Issue 5**

9459 Moved from X/OPEN UNIX extension to BASE.

9460 Normative text previously in the APPLICATION USAGE section is moved to the
9461 DESCRIPTION.

9462 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

9463 **Issue 6**

9464 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

9465 **NAME**

9466 environ — array of character pointers to the environment strings

9467 **SYNOPSIS**

9468
```
extern char **environ;
```

9469 **DESCRIPTION**

9470 Refer to the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 8, Environment Variables |
9471 and *exec*. |

NAME

erand48 — generate uniformly distributed pseudo-random numbers

SYNOPSIS

XSI

```
#include <stdlib.h>

double erand48(unsigned short xsubi[3]);
```

DESCRIPTION

Refer to *drand48*( ).

9480 **NAME**

9481 erf, erfc, erfcf, erfcl, erff, erfl — error and complementary error functions

9482 **SYNOPSIS**

9483 XSI
```
#include <math.h>

double erf(double x);
double erfc(double x);
float erfcf(float x);
long double erfcl(long double x);
float erff(float x);
long double erfl(long double x);
```

9490

9491 **DESCRIPTION**

9492 The *erf*( ), *erff,*( ) and *erfl*( ) functions shall compute the error function of *x*, defined as:

9493 $$\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$$

9494 The *erfc*( ), *erfcf*( ), and *erfcl*( ) functions shall compute 1.0 – *erf*(*x*).

9495 An application wishing to check for error situations should set *errno* to 0 before calling *erf*( ). If
9496 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

9497 **RETURN VALUE**

9498 Upon successful completion, these functions shall return the value of the error function and
9499 complementary error function, respectively.

9500 If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

9501 If the correct value would cause underflow, 0 shall be returned and *errno* may be set to
9502 [ERANGE].

9503 **ERRORS**

9504 The *erfc*( ), *erfcf*( ), and *erfcl*( ) functions shall fail if:

9505 [ERANGE] The value of *x* is too large.

9506 The *erf*( ) and *erfc*( ) functions may fail if:

9507 [EDOM] The value of *x* is NaN.

9508 [ERANGE] The result underflows

9509 No other errors shall occur.

9510 **EXAMPLES**

9511 None.

9512 **APPLICATION USAGE**

9513 The *erfc*( ) function is provided because of the extreme loss of relative accuracy if *erf*(*x*) is called
9514 for large *x* and the result subtracted from 1.0.

9515 **RATIONALE**

9516 None.

9517 **FUTURE DIRECTIONS**

9518 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 4

References to *matherr*( ) are removed.

The RETURN VALUE and ERRORS sections are substantially rewritten to rationalize error handling in the mathematics functions.

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 *erfcf*( ), *erfcl*( ), *erff*( ), and *erfl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

9533 **NAME**

9534 errno — error return value

9535 **SYNOPSIS**

9536 `#include <errno.h>`

9537 **DESCRIPTION**

9538 *errno* is used by many functions to return error values.

9539 Many functions provide an error number in *errno* which has type **int** and is defined in **<errno.h>**.
9540 The value of *errno* shall be defined only after a call to a function for which it is explicitly stated to
9541 be set and until it is changed by the next function call. The value of *errno* should only be
9542 examined when it is indicated to be valid by a function's return value. Programs should obtain
9543 the definition of *errno* by the inclusion of **<errno.h>**. No function in this volume of
9544 IEEE Std. 1003.1-200x shall set *errno* to 0.

9545 It is unspecified whether *errno* is a macro or an identifier declared with external linkage. If a
9546 macro definition is suppressed in order to access an actual object, or a program defines an
9547 identifier with the name *errno*, the behavior is undefined.

9548 The symbolic values stored in *errno* are documented in the ERRORS sections on all relevant
9549 pages.

9550 **RETURN VALUE**

9551 None.

9552 **ERRORS**

9553 None.

9554 **EXAMPLES**

9555 None.

9556 **APPLICATION USAGE**

9557 Previously both POSIX and X/Open documents were more restrictive than the ISO C standard
9558 in that they required *errno* to be defined as an external variable, whereas the ISO C standard
9559 required only that *errno* be defined as a modifiable **lvalue** with type **int**.

9560 A program that uses *errno* for error checking should set it to 0 before a function call, then inspect
9561 it before a subsequent function call.

9562 **RATIONALE**

9563 None.

9564 **FUTURE DIRECTIONS**

9565 None.

9566 **SEE ALSO**

9567 Section 2.3, the Base Definitions volume of IEEE Std. 1003.1-200x, **<errno.h>**

9568 **CHANGE HISTORY**

9569 First released in Issue 1. Derived from Issue 1 of the SVID.

9570 **Issue 4**

9571 The FUTURE DIRECTIONS section is deleted.

9572 The following changes are incorporated for alignment with the ISO C standard:

9573 • The DESCRIPTION now guarantees that *errno* is set to 0 at program start-up, and that it is
9574 never reset to 0 by any XSI function.

- The APPLICATION USAGE section is added. This issue is aligned with the ISO C standard, which permits *errno* to be a macro.

Issue 5

The following sentence is deleted from the DESCRIPTION: ‘‘The value of *errno* is 0 at program start-up, but is never set to 0 by any XSI function’’. The DESCRIPTION also no longer states that conforming implementations may support the declaration:

```
extern int errno;
```

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.

9588 **NAME**

9589 environ, execl, execv, execle, execve, execlp, execvp — execute a file

9590 **SYNOPSIS**

9591
```
#include <unistd.h>
```

9592
```
extern char **environ;
```
9593
```
int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
```
9594
```
int execv(const char *path, char *const argv[]);
```
9595
```
int execle(const char *path, const char *arg0, ... /*,
```
9596
```
    (char *)0, char *const envp[]*/);
```
9597
```
int execve(const char *path, char *const argv[], char *const envp[]);
```
9598
```
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
```
9599
```
int execvp(const char *file, char *const argv[]);
```

9600 **DESCRIPTION**

9601 The *exec* functions shall replace the current process image with a new process image. The new
9602 image is constructed from a regular, executable file called the *new process image file*. There shall
9603 be no return from a successful *exec*, because the calling process image is overlaid by the new
9604 process image.

9605 When a C-language program is executed as a result of this call, it shall be entered as a C-
9606 language function call as follows:

9607
```
int main (int argc, char *argv[]);
```

9608 where *argc* is the argument count and *argv* is an array of character pointers to the arguments
9609 themselves. In addition, the following variable:

9610
```
extern char **environ;
```

9611 is initialized as a pointer to an array of character pointers to the environment strings. The *argv*
9612 and *environ* arrays are each terminated by a null pointer. The null pointer terminating the *argv*
9613 array is not counted in *argc*.

9614 THR Conforming multi-threaded applications shall not use the *environ* variable to access or modify
9615 any environment variable while any other thread is concurrently modifying any environment
9616 variable. A call to any function dependent on any environment variable shall be considered a use
9617 of the *environ* variable to access that environment variable.

9618 The arguments specified by a program with one of the *exec* functions shall be passed on to the
9619 new process image in the corresponding *main*( ) arguments.

9620 The argument *path* points to a path name that identifies the new process image file.

9621 The argument *file* is used to construct a path name that identifies the new process image file. If
9622 the *file* argument contains a slash character, the *file* argument shall be used as the path name for
9623 this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed
9624 as the environment variable *PATH* (see the Base Definitions volume of IEEE Std. 1003.1-200x, |
9625 Chapter 8, Environment Variables). If this environment variable is not present, the results of the |
9626 search are implementation-defined. |

9627 If the process image file is not a valid executable object, *execlp*( ) and *execvp*( ) shall use the |
9628 contents of that file as standard input to a command interpreter conforming to *system*( ). In this |
9629 case, the command interpreter becomes the new process image.

9630 The arguments represented by *arg0*,... are pointers to null-terminated character strings. These
9631 strings constitute the argument list available to the new process image. The list is terminated by
9632 a null pointer. The argument *arg0* should point to a file name that is associated with the process

being started by one of the *exec* functions.

The argument *argv* is an array of character pointers to null-terminated strings. The application shall ensure that the last member of this array is a null pointer. These strings constitute the argument list available to the new process image. The value in *argv*[0] should point to a file name that is associated with the process being started by one of the *exec* functions.

The argument *envp* is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The *envp* array is terminated by a null pointer.

For those forms not containing an *envp* pointer (*execl*( ), *execv*( ), *execlp*( ), and *execvp*( )), the environment for the new process image is taken from the external variable *environ* in the calling process.

The number of bytes available for the new process' combined argument and environment lists is {ARG_MAX}. It is implementation-defined whether null terminators, pointers, and/or any alignment bytes are included in this total.

File descriptors open in the calling process image remain open in the new process image, except for those whose close-on-exec flag FD_CLOEXEC is set. For those file descriptors that remain open, all attributes of the open file description remain unchanged. For any file descriptor that is closed for this reason, file locks are removed as a result of the close as described in *close*( ). Locks that are not removed by closing of file descriptors remain unchanged.

Directory streams open in the calling process image shall be closed in the new process image.

XSI The state of conversion descriptors and message catalog descriptors in the new process image is undefined. For the new process, the equivalent of:

```
setlocale(LC_ALL, "C")
```

is executed at start-up.

Signals set to the default action (SIG_DFL) in the calling process image shall be set to the default action in the new process image. Signals set to be ignored (SIG_IGN) by the calling process image shall be set to be ignored by the new process image. Signals set to be caught by the calling
XSI process image shall be set to the default action in the new process image (see **<signal.h>**). After a successful call to any of the *exec* functions, alternate signal stacks are not preserved and the SA_ONSTACK flag shall be cleared for all signals.

After a successful call to any of the *exec* functions, any functions previously registered by *atexit*( ) are no longer registered.

XSI If the ST_NOSUID bit is set for the file system containing the new process image file, then the effective user ID, effective group ID, saved set-user-ID, and saved set-group-ID are unchanged in the new process image. Otherwise, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the new process image is set to the user ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image shall be saved (as the saved set-user-ID and the saved set-group-ID) for use by *setuid*( ).

XSI Any shared memory segments attached to the calling process image shall not be attached to the new process image.

SEM Any named semaphores open in the calling process shall be closed as if by appropriate calls to *sem_close*( ).

TYM Any blocks of typed memory that were mapped in the calling process are unmapped, as if *munmap*( ) was implicitly called to unmap them.

ML Memory locks established by the calling process via calls to *mlockall*( ) or *mlock*( ) shall be removed. If locked pages in the address space of the calling process are also mapped into the address spaces of other processes and are locked by those processes, the locks established by the other processes shall be unaffected by the call by this process to the *exec* function. If the *exec* function fails, the effect on memory locks is unspecified.

MF|SHM Memory mappings created in the process are unmapped before the address space is rebuilt for the new process image.

PS For the SCHED_FIFO and SCHED_RR scheduling policies, the policy and priority settings shall not be changed by a call to an *exec* function. For other scheduling policies, the policy and priority settings on *exec* are implementation-defined.

TMR Per-process timers created by the calling process shall be deleted before replacing the current process image with the new process image.

MSG All open message queue descriptors in the calling process shall be closed, as described in *mq_close*( ).

AIO Any outstanding asynchronous I/O operations may be canceled. Those asynchronous I/O operations that are not canceled shall complete as if the *exec* function had not yet occurred, but any associated signal notifications shall be suppressed. It is unspecified whether the *exec* function itself blocks awaiting such I/O completion. In no event, however, shall the new process image created by the *exec* function be affected by the presence of outstanding asynchronous I/O operations at the time the *exec* function is called. Whether any I/O is canceled, and which I/O may be canceled upon *exec*, is implementation-defined.

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 *exec*ed shall not be reinitialized or altered as a result of the *exec* function other than to reflect the time spent by the process executing the *exec* function itself.

TCT The initial value of the CPU-time clock of the initial thread of the new process image shall be set to zero.

TRC If the calling process is being traced, the new process image continues 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 *posix_trace_eventid_open*( ) or the *posix_trace_trid_eventid_open*( ) functions in the calling process image.

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 *posix_trace_shutdown*( ) function.

The new process also inherits at least the following attributes from the calling process image:

XSI • Nice value (see *nice*( ))

XSI • *semadj* values (see *semop*( ))

• Process ID

• Parent process ID

- Process group ID
- Session membership
- Real user ID
- Real group ID
- Supplementary group IDs
- Time left until an alarm clock signal (see *alarm*( ))
- Current working directory
- Root directory
- File mode creation mask (see *umask*( ))

XSI
- File size limit (see *ulimit*( ))

- Process signal mask (see *sigprocmask*( ))
- Pending signal (see *sigpending*( ))
- *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* (see *times*( ))

XSI
- Resource limits

XSI
- Controlling terminal

XSI
- Interval timers

All other process attributes defined in this volume of IEEE Std. 1003.1-200x shall be the same in the new and old process images. The inheritance of process attributes not defined by this volume of IEEE Std. 1003.1-200x is implementation-defined.

A call to any *exec* function from a process with more than one thread results in all threads being terminated and the new executable image being loaded and executed. No destructor functions shall be called.

Upon successful completion, the *exec* functions shall mark for update the *st_atime* field of the file. If an *exec* function failed but was able to locate the *process image file*, whether the *st_atime* field is marked for update is unspecified. Should the *exec* function succeed, the process image file shall be considered to have been opened with *open*( ). The corresponding *close*( ) shall be considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the *exec* functions. The *argv*[ ] and *envp*[ ] arrays of pointers and the strings to which those arrays point shall not be modified by a call to one of the *exec* functions, except as a consequence of replacing the process image.

XSI The saved resource limits in the new process image are set to be a copy of the process' corresponding hard and soft limits.

RETURN VALUE

If one of the *exec* functions returns to the calling process image, an error has occurred; the return value shall be –1, and *errno* shall be set to indicate the error.

ERRORS

The *exec* functions shall fail if:

[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.

[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.

[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.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* or *file* argument.

[ENAMETOOLONG] The length of the *path* or *file* arguments exceeds {PATH_MAX} or a path name component is longer than {NAME_MAX}.

[ENOENT] A component of *path* or *file* does not name an existing file or *path* or *file* is an empty string.

[ENOTDIR] A component of the new process image file's path prefix is not a directory.

The *exec* functions, except for *execlp*( ) and *execvp*( ), shall fail if:

[ENOEXEC] The new process image file has the appropriate access permission but has an unrecognized format.

The *exec* functions may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* or *file* argument.

[ENAMETOOLONG] As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted path name string exceeded {PATH_MAX}.

[ENOMEM] The new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.

[ETXTBSY] The new process image file is a pure procedure (shared text) file that is currently open for writing by some process.

EXAMPLES

Using execl( )

The following example executes the *ls* command, specifying the path name of the executable (**/bin/ls**) and using arguments supplied directly to the command to produce single-column output.

```
#include <unistd.h>

int ret;
...
ret = execl ("/bin/ls", "ls", "-1", NULL);
```

Using execle()

The following example is similar to **Using execl()** (on page 794). In addition, it specifies the environment for the new process image using the *env* argument.

```
#include <unistd.h>

int ret;
char *env[] = { "HOME=/usr/home", "LOGNAME=home", NULL };
...
ret = execle ("/bin/ls", "ls", "-l", NULL, env);
```

Using execlp()

The following example searches for the location of the *ls* command among the directories specified by the *PATH* environment variable.

```
#include <unistd.h>

int ret;
...
ret = execlp ("ls", "ls", "-l", NULL);
```

Using execv()

The following example passes arguments to the *ls* command in the *cmd* array.

```
#include <unistd.h>

int ret;
char *cmd[] = { "ls", "-l", NULL };
...
ret = execv ("/bin/ls", cmd);
```

Using execve()

The following example passes arguments to the *ls* command in the *cmd* array, and specifies the environment for the new process image using the *env* argument.

```
#include <unistd.h>

int ret;
char *cmd[] = { "ls", "-l", NULL };
char *env[] = { "HOME=/usr/home", "LOGNAME=home", NULL };
...
ret = execve ("/bin/ls", cmd, env);
```

Using execvp()

The following example searches for the location of the *ls* command among the directories specified by the *PATH* environment variable, and passes arguments to the *ls* command in the *cmd* array.

```
#include <unistd.h>

int ret;
char *cmd[] = { "ls", "-l", NULL };
...
```

```
ret = execvp ("ls", cmd);
```

APPLICATION USAGE

As the state of conversion descriptors and message catalog descriptors in the new process image is undefined, portable applications should not rely on their use and should close them prior to calling one of the *exec* functions.

Applications that require other than the default POSIX locale should call *setlocale*( ) with the appropriate parameters to establish the locale of the new process.

The *environ* array should not be accessed directly by the application.

RATIONALE

Early proposals required that the value of *argc* passed to *main*( ) be ''one or greater''. This was driven by the same requirement in drafts of the ISO C standard. In fact, historical implementations have passed a value of zero when no arguments are supplied to the caller of the *exec* functions. This requirement was removed from the ISO C standard and subsequently removed from this volume of IEEE Std. 1003.1-200x as well. The wording, in particular the use of the word *should*, requires a Strictly Conforming POSIX Application to pass at least one argument to the *exec* function, thus guaranteeing that *argc* be one or greater when invoked by such an application. In fact, this is good practice, since many existing applications reference *argv*[0] without first checking the value of *argc*.

The requirement on a Strictly Conforming POSIX Application also states that the value passed as the first argument be a file name associated with the process being started. Although some existing applications pass a path name rather than a file name in some circumstances, a file name is more generally useful, since the common usage of *argv*[0] is in printing diagnostics. In some cases the file name passed is not the actual file name of the file; for example, many implementations of the *login* utility use a convention of prefixing a hyphen ('−') to the actual file name, which indicates to the command interpreter being invoked that it is a ''login shell''.

Some systems can *exec* shell scripts.

One common historical implementation is that the *execl*( ), *execv*( ), *execle*( ), and *execve*( ) functions return an [ENOEXEC] error for any file not recognizable as executable, including a shell script. When the *execlp*( ) and *execvp*( ) functions encounter such a file, they assume the file to be a shell script and invoke a known command interpreter to interpret such files. These implementations of *execvp*( ) and *execlp*( ) only give the [ENOEXEC] error in the rare case of a problem with the command interpreter's executable file. Because of these implementations, the [ENOEXEC] error is not mentioned for *execlp*( ) or *execvp*( ), although implementations can still give it.

Another way that some historical implementations handle shell scripts is by recognizing the first two bytes of the file as the character string "#!" and using the remainder of the first line of the file as the name of the command interpreter to execute.

Some implementations provide a third argument to *main*( ) called *envp*. This is defined as a pointer to the environment. The ISO C standard specifies invoking *main*( ) with two arguments, so implementations must support applications written this way. Since this volume of IEEE Std. 1003.1-200x defines the global variable *environ*, which is also provided by historical implementations and can be used anywhere that *envp* could be used, there is no functional need for the *envp* argument. Applications should use the *getenv*( ) function rather than accessing the environment directly via either *envp* or *environ*. Implementations are required to support the two-argument calling sequence, but this does not prohibit an implementation from supporting *envp* as an optional third argument.

This volume of IEEE Std. 1003.1-200x specifies that signals set to SIG_IGN remain set to SIG_IGN, and that the process signal mask be unchanged across an *exec*. This is consistent with historical implementations, and it permits some useful functionality, such as the *nohup* command. However, it should be noted that many existing applications wrongly assume that they start with certain signals set to the default action and/or unblocked. In particular, applications written with a simpler signal model that does not include blocking of signals, such as the one in the ISO C standard, may not behave properly if invoked with some signals blocked. Therefore, it is best not to block or ignore signals across *exec*s without explicit reason to do so, and especially not to block signals across *exec*s of arbitrary (not closely co-operating) programs.

The *exec* functions always save the value of the effective user ID and effective group ID of the process at the completion of the *exec*, whether or not the set-user-ID or the set-group-ID bit of the process image file is set.

The statement about *argv*[ ] and *envp*[ ] being constants is included to make explicit to future writers of language bindings that these objects are completely constant. Due to a limitation of the ISO C standard, it is not possible to state that idea in standard C. Specifying two levels of *const–qualification* for the *argv*[ ] and *envp*[ ] parameters for the *exec* functions may seem to be the natural choice, given that these functions do not modify either the array of pointers or the characters to which the function points, but this would disallow existing correct code. Instead, only the array of pointers is noted as constant. The table of assignment compatibility for *dst=src*, derived from the ISO C standard summarizes the compatibility:

dst:	**char *[ ]**	**const char *[ ]**	**char *const[ ]**	**const char *const[ ]**
src:				
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 path name may be constructed by taking elements in the *PATH* variable and putting them together with the file name, 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].

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *atexit*( ), *chmod*( ), *close*( ), *exit*( ), *fcntl*( ), *fork*( ), *fstatvfs*( ), *getenv*( ), *getitimer*( ), *getrlimit*( ), *mmap*( ), *nice*( ), <REFERENCE UNDEFINED>(posix_trace_eventid), *posix_trace_shutdown*( ), *posix_trace_trid_eventid_open*( ), *putenv*( ), *semop*( ), *setlocale*( ), *shmat*( ), *sigaction*( ), *sigaltstack*( ), *sigpending*( ), *sigprocmask*( ), *system*( ), *times*( ), *ulimit*( ), *umask*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

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

Issue 4

The **<unistd.h>** header is added to the SYNOPSIS section.

The **const** keyword is added to identifiers of constant type (for example, *path*, *file*).

In the DESCRIPTION:

- An indication of the disposition of conversion descriptors after a call to one of the *exec* functions is added.
- A statement about the interaction between *exec* and *atexit*( ) is added.
- *usually* in the descriptions of argument pointers is removed.
- *owner ID* is changed to *user ID*.
- Shared memory is no longer optional.
- The penultimate paragraph is changed to correct an error in Issue 3. It now refers to ‘‘All other process attributes ...’’ instead of ‘‘All the process attributes ...’’

A note about the initialization of locales is added to the APPLICATION USAGE section.

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- In the ERRORS section, the description of the [ENOEXEC] error is changed to indicate that this error does not apply to *execlp*( ) and *execvp*( ), and the [ENOMEM] error is added.

The following change is incorporated for alignment with the FIPS requirements:

- In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path name component is larger that {NAME_MAX} is now defined as mandatory and marked as an extension.

Issue 4, Version 2

The following changes are incorporated for X/OPEN UNIX conformance:

- The DESCRIPTION is changed:
  - — To indicate the disposition of alternate signal stacks, the SA_ONSTACK flag, and mappings established through *mmap*( ) after a successful call to one of the *exec* functions.
  - — The effects of ST_NOSUID being set for a file system are defined.
  - — A statement is added that mappings established through *mmap*( ) are not preserved across an *exec*.
  - — The list of inherited process attributes is extended to include resource limits, the controlling terminal, and interval timers.
- In the ERRORS section:
  - — The condition whereby [ELOOP] is returned if too many symbolic links are encountered during path name resolution is defined as mandatory.
  - — A second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of path name resolution of a symbolic link.

Issue 5

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

Large File Summit extensions are added.

Issue 6

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, behavior is defined for when the process image file is not a valid executable.
- In this issue, _POSIX_SAVED_IDS is mandated, thus the effective user ID and effective group ID of the new process image shall be saved (as the saved set-user-ID and the saved set-group-ID) for use by the *setuid*( ) function.
- 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 [EINVAL] mandatory error condition is added.
- The [ELOOP] optional error condition is added.

The description of CPU-time clock semantics is added for alignment with IEEE Std. 1003.1d-1999.

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

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements. |

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

NAME

exit, _Exit, _exit — terminate a process

SYNOPSIS

```
#include <stdlib.h>

void exit(int status);

#include <unistd.h>

void _Exit(int status);
void _exit(int status);
```

DESCRIPTION

CX The functionality described on this reference page for the *exit*() function is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard are unintentional. This volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

The *exit*() function shall first call all functions registered by *atexit*(), in the reverse order of their registration, except that a function is called after any previously registered functions that had already been called at the time it was registered. Each function is called as many times as it was registered. If, during the call to any such function, a call to the *longjmp*() function is made that would terminate the call to the registered function, the behavior is undefined.

If a function registered by a call to *atexit*() fails to return, the remaining registered functions shall not be called and the rest of the *exit*() processing shall not be completed. If *exit*() is called more than once, the effects are undefined.

CX The *exit*() function then flushes all output streams, closes all open streams, and removes all files created by *tmpfile*(). Finally, control is returned to the host environment as described below. The values of *status* can be EXIT_SUCCESS or EXIT_FAILURE, as described in **<stdlib.h>**, or any CX implementation-defined value, although note that only the range 0 through 255 shall be available to a waiting parent process.

The *_Exit*() and *_exit*() functions shall be functionally identical.

CX The *_Exit*(), *_exit*(), and *exit*() functions shall terminate the calling process with the following consequences:

- XSI All of the file descriptors, directory streams, conversion descriptors, and message catalog descriptors open in the calling process are closed.
- XSI If the parent process of the calling process is executing a *wait*(), *waitid*(), or *waitpid*(), and has neither set its SA_NOCLDWAIT flag nor set SIGCHLD to SIG_IGN, it is notified of the calling process' termination and the low-order eight bits (that is, bits 0377) of *status* are made available to it. If the parent is not waiting, the child's status shall be made available to it when the parent subsequently executes *wait*(), *waitid*(), or *waitpid*(). XSI
- XSI If the parent process of the calling process is not executing a *wait*(), *waitid*(), or *waitpid*(), and has not set its SA_NOCLDWAIT flag, or set SIGCHLD to SIG_IGN, the calling process is transformed into a *zombie process*. A *zombie process* is an inactive process and it shall be deleted at some later time when its parent process executes *wait*(), *waitid*(), or *waitpid*(). XSI
- Termination of a process does not directly terminate its children. The sending of a SIGHUP signal as described below indirectly terminates children in some circumstances.
- If the implementation supports the SIGCHLD signal, a SIGCHLD shall be sent to the parent process.

XSI
- If the parent process has set its SA_NOCLDWAIT flag, or set SIGCHLD to SIG_IGN, the status shall be discarded, and the lifetime of the calling process shall end immediately. If SA_NOCLDWAIT is set, it is implementation-defined whether a SIGCHLD signal shall be sent to the parent process.

- The parent process ID of all of the calling process' existing child processes and zombie processes is set to the process ID of an implementation-defined system process. That is, these processes are inherited by a special system process.

XSI
- Each attached shared-memory segment is detached and the value of *shm_nattch* (see *shmget*( )) in the data structure associated with its shared memory ID is decremented by 1.

XSI
- For each semaphore for which the calling process has set a *semadj* value (see *semop*( )), that value is 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 the foreground process group of the controlling terminal belonging to the calling process.

- If the process is a controlling process, the controlling terminal associated with the session is disassociated from the session, allowing it to be acquired by a new controlling process.

- If the exit of the process causes a process group to become orphaned, and if any member of the newly-orphaned process group is stopped, then a SIGHUP signal followed by a SIGCONT signal shall be sent to each process in the newly-orphaned process group.

SEM
- All open named semaphores in the calling process shall be closed as if by appropriate calls to *sem_close*( ).

ML
- Any memory locks established by the process via calls to *mlockall*( ) or *mlock*( ) shall be removed. If locked pages in the address space of the calling process are also mapped into the address spaces of other processes and are locked by those processes, the locks established by the other processes shall be unaffected by the call by this process to *_Exit*( ) or *_exit*( ).

MF|SHM
- Memory mappings created in the process are unmapped before the process is destroyed.

TYM
- Any blocks of typed memory that were mapped in the calling process are unmapped, as if *munmap*( ) was implicitly called to unmap them.

MSG
- All open message queue descriptors in the calling process shall be closed as if by appropriate calls to *mq_close*( ).

AIO
- Any outstanding cancelable asynchronous I/O operations may be canceled. Those asynchronous I/O operations that are not canceled shall complete as if the *_Exit*( ) or *_exit*( ) operation had not yet occurred, but any associated signal notifications shall be suppressed. The *_Exit*( ) or *_exit*( ) operation may block awaiting such I/O completion. Whether any I/O is canceled, and which I/O may be canceled upon *_Exit*( ) or *_exit*( ), is implementation-defined.

- Threads terminated by a call to *_Exit*( ) or *_exit*( ) shall not invoke their cancelation cleanup handlers or per-thread data destructors.

TRC
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 by the *posix_trace_shutdown*( ) function, and any process' mapping of trace event names to trace event type identifiers built for these trace streams may be deallocated.

RETURN VALUE

These functions do not return.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Normally applications should use *exit*( ) rather than *_Exit*( ) or *_exit*( ).

RATIONALE

Process Termination

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

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

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

Despite the silence of this volume of IEEE Std. 1003.1-200x on this feature, applications are advised not to create files named **core** because of potential conflicts in many implementations. Some historical implementations use a different name than **core** for the file, such as by appending the process ID to the file name.

Terminating a Process

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

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

As required by the ISO C standard, using **return** from *main*( ) is equivalent to calling *exit*( ) with the same argument value. Also, reaching the end of the *main*( ) function is equivalent to using *exit*( ) with an unspecified value.

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

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

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

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

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

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

The ISO/IEC 9899:1999 standard adds the *_Exit*( ) function that results in immediate program termination without triggering signals or *atexit*( )-registered functions. In IEEE Std. 1003.1-200x,

this is equivalent to the *_exit*( ) function.

FUTURE DIRECTIONS

None.

SEE ALSO

atexit( ), *close*( ), *fclose*( ), *longjmp*( ), <REFERENCE UNDEFINED>(posix_trace_eventid), *posix_trace_shutdown*( ), *posix_trace_trid_eventid_open*( ), *semop*( ), *shmget*( ), *sigaction*( ), *wait*( ), *waitid*( ), *waitpid*( ), 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 4

The **<unistd.h>** header is added to the SYNOPSIS for *_exit*( ).

In the DESCRIPTION, text is added describing the behavior when a function registered by *atexit*( ) fails to return, and the consequences of calling *exit*( ) more than once.

The phrase ''If the implementation supports job control'' is removed from the last bullet in the DESCRIPTION. This is because job control is now defined as mandatory for all conforming implementations.

The following change is incorporated for alignment with the ISO C standard:

- In the DESCRIPTION, interactions between *exit*( ) and *atexit*( ) are defined, and it is now stated explicitly that all files created by *tmpfile*( ) are removed.

Issue 4, Version 2

The following changes to the DESCRIPTION are incorporated for X/OPEN UNIX conformance:

- References to the functions *wait3*( ) and *waitid*( ) are added in appropriate places throughout the text.
- Interactions with the SA_NOCLDWAIT flag and SIGCHLD signal are defined.
- It is specified that each mapped memory object is unmapped.

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 now 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.

10239 **NAME**

10240 exp, expf, expl — exponential function

10241 **SYNOPSIS**

10242
```
#include <math.h>
```

10243
```
double exp(double x);
```
10244
```
float expf(float x);
```
10245
```
long double expl(long double x);
```

10246 **DESCRIPTION**

10247 CX The functionality described on this reference page is aligned with the ISO C standard. Any
10248 conflict between the requirements described here and the ISO C standard is unintentional. This
10249 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

10250 These functions shall compute the exponent of x, defined as e^x.

10251 An application wishing to check for error situations should set *errno* to 0 before calling *exp*( ). If
10252 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

10253 **RETURN VALUE**

10254 Upon successful completion, these functions shall return the exponential value of x.

10255 If the correct value would cause overflow, *exp*( ) shall return HUGE_VAL and set *errno* to
10256 [ERANGE].

10257 If the correct value would cause underflow, *exp*( ) shall return 0 and may set *errno* to [ERANGE].

10258 XSI If x is NaN, NaN shall be returned and *errno* may be set to [EDOM].

10259 **ERRORS**

10260 These functions shall fail if:

10261 [ERANGE] The result overflows.

10262 These functions may fail if:

10263 XSI [EDOM] The value of x is NaN.

10264 [ERANGE] The result underflows

10265 XSI No other errors shall occur.

10266 **EXAMPLES**

10267 None.

10268 **APPLICATION USAGE**

10269 None.

10270 **RATIONALE**

10271 None.

10272 **FUTURE DIRECTIONS**

10273 None.

10274 **SEE ALSO**

10275 *isnan*( ), *log*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

10276 **CHANGE HISTORY**

10277 First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 4

References to *matherr*( ) are removed.

The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the ISO C standard and to rationalize error handling in the mathematics functions.

The return value specified for [EDOM] is marked as an extension.

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 *expf*( ) and *expl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

10288 **NAME**

10289 exp2, exp2f, exp2l — exponential base 2 functions

10290 **SYNOPSIS**

10291
```
#include <math.h>
```

10292
```
double exp2(double x);
```
10293
```
float exp2f(float x);
```
10294
```
long double exp2l(long double x);
```

10295 **DESCRIPTION**

10296 CX The functionality described on this reference page is aligned with the ISO C standard. Any
10297 conflict between the requirements described here and the ISO C standard is unintentional. This
10298 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

10299 These functions shall compute the base 2 exponent of *x*, defined as e^x.

10300 An application wishing to check for error situations should set *errno* to 0 before calling these
10301 functions. If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

10302 **RETURN VALUE**

10303 Upon successful completion, these functions shall return 2^x.

10304 If the correct value would cause overflow, these functions shall return HUGE_VAL and set *errno*
10305 to [ERANGE].

10306 If the correct value would cause underflow, these functions shall return 0 and may set *errno* to
10307 [ERANGE].

10308 If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

10309 **ERRORS**

10310 These functions shall fail if:

10311 [ERANGE] The result overflows.

10312 These functions may fail if:

10313 [EDOM] The value of *x* is NaN.

10314 [ERANGE] The result underflows

10315 No other errors shall occur.

10316 **EXAMPLES**

10317 None.

10318 **APPLICATION USAGE**

10319 None.

10320 **RATIONALE**

10321 None.

10322 **FUTURE DIRECTIONS**

10323 None.

10324 **SEE ALSO**

10325 *exp*( ), *isnan*( ), *log*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

10326 **CHANGE HISTORY**
10327 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

10328 **NAME**

10329 expm1, expm1f, expm1l — compute exponential functions

10330 **SYNOPSIS**

10331 `#include <math.h>`

10332 `double expm1(double x);`
10333 `float expm1f(float x);`
10334 `long double expm1l(long double x);`

10335 **DESCRIPTION**

10336 These functions shall compute e^x–1.0.

10337 **RETURN VALUE**

10338 If *x* is NaN, then these functions shall return NaN and *errno* may be set to [EDOM].

10339 If *x* is positive infinity, these functions shall return positive infinity.

10340 If *x* is negative infinity, these functions shall return –1.0.

10341 If the value overflows, these functions shall return HUGE_VAL and may set *errno* to [ERANGE].

10342 **ERRORS**

10343 These functions may fail if:

10344 [EDOM] The value of *x* is NaN.

10345 [ERANGE] The result overflows.

10346 **EXAMPLES**

10347 None.

10348 **APPLICATION USAGE**

10349 The value of *expm1*(*x*) may be more accurate than *exp*(*x*)–1.0 for small values of *x*.

10350 The *expm1*( ) and *log1p*( ) functions are useful for financial calculations of $((1+x)^n-1)/x$, namely:

10351 expm1(*n* * log1p(*x*))/*x*

10352 when *x* is very small (for example, when calculating small daily interest rates). These functions
10353 also simplify writing accurate inverse hyperbolic functions.

10354 **RATIONALE**

10355 None.

10356 **FUTURE DIRECTIONS**

10357 None.

10358 **SEE ALSO**

10359 *exp*( ), *ilogb*( ), *log1p*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

10360 **CHANGE HISTORY**

10361 First released in Issue 4, Version 2.

10362 **Issue 5**

10363 Moved from X/OPEN UNIX extension to BASE.

10364 **Issue 6**

10365 The *expm1f*( ) and *expm1l*( ) functions are added for alignment with the ISO/IEC 9899:1999
10366 standard.

10367 **NAME**

10368 fabs, fabsf, fabsl — absolute value function

10369 **SYNOPSIS**

10370
```
#include <math.h>
```

10371
```
double fabs(double x);
```
10372
```
float fabsf(float x);
```
10373
```
long double fabsl(long double x);
```

10374 **DESCRIPTION**

10375 CX The functionality described on this reference page is aligned with the ISO C standard. Any
10376 conflict between the requirements described here and the ISO C standard is unintentional. This
10377 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

10378 These functions shall compute the absolute value of x, $|x|$.

10379 An application wishing to check for error situations should set *errno* to 0 before calling *fabs*( ). If
10380 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

10381 **RETURN VALUE**

10382 Upon successful completion, these functions shall return the absolute value of x.

10383 XSI If x is NaN, NaN shall be returned and *errno* may be set to [EDOM].

10384 If the result underflows, 0 shall be returned and *errno* may be set to [ERANGE].

10385 **ERRORS**

10386 These functions may fail if:

10387 XSI [EDOM] The value of x is NaN.

10388 [ERANGE] The result underflows

10389 XSI No other errors shall occur.

10390 **EXAMPLES**

10391 None.

10392 **APPLICATION USAGE**

10393 None.

10394 **RATIONALE**

10395 None.

10396 **FUTURE DIRECTIONS**

10397 None.

10398 **SEE ALSO**

10399 *isnan*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

10400 **CHANGE HISTORY**

10401 First released in Issue 1. Derived from Issue 1 of the SVID.

10402 **Issue 4**

10403 References to *matherr*( ) are removed.

10404 The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the
10405 ISO C standard and to rationalize error handling in the mathematics functions.

10406 The return value specified for [EDOM] is marked as an extension.

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.

NAME

fattach — attach a STREAMS-based file descriptor to a file in the file system name space (**STREAMS**)

SYNOPSIS

XSR
```
#include <stropts.h>

int fattach(int fildes, const char *path);
```

DESCRIPTION

Notes to Reviewers

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

Re D1, XSH, ERN 111: if the original file had multiple links, the streams file still has only one? I presume that a stream is actually attached to an inode, not a file name. If so, there continue to exist multiple links to the object, even though it shows a link count of 1. If it associated the stream with a file name, then the following sentence is wrong.

The *fattach*( ) function attaches a STREAMS-based file descriptor to a file, effectively associating a path name with *fildes*. The application shall ensure that the *fildes* argument is a valid open file descriptor associated with a STREAMS file. The *path* argument points to a path name of an existing file. The application shall ensure that the process has appropriate privileges, or is the owner of the file named by *path* and has write permission. A successful call to *fattach*( ) shall cause all path names that name the file named by *path* to name the STREAMS file associated with *fildes*, until the STREAMS file is detached from the file. A STREAMS file can be attached to more than one file and can have several path names associated with it.

The attributes of the named STREAMS file shall be initialized as follows: the permissions, user ID, group ID, and times are set to those of the file named by *path*, the number of links is set to 1, and the size and device identifier are set to those of the STREAMS file associated with *fildes*. If any attributes of the named STREAMS file are subsequently changed (for example, by *chmod*( )), neither the attributes of the underlying file nor the attributes of the STREAMS file to which *fildes* refers shall be affected.

File descriptors referring to the underlying file, opened prior to an *fattach*( ) call, shall continue to refer to the underlying file.

RETURN VALUE

Upon successful completion, *fattach*( ) shall return 0. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *fattach*( ) function shall fail if:

[EACCES] Search permission is denied for a component of the path prefix, or the process is the owner of *path* but does not have write permissions on the file named by *path*.

[EBADF] The *fildes* argument is not a valid open file descriptor.

[EBUSY] The file named by *path* is currently a mount point or has a STREAMS file attached to it.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

[ENAMETOOLONG]
The size of *path* exceeds {PATH_MAX} or a component of *path* is longer than {NAME_MAX}.

[ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The effective user ID of the process is not the owner of the file named by *path* and the process does not have appropriate privilege.

The *fattach*( ) function may fail if:

[EINVAL] The *fildes* argument does not refer to a STREAMS file.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
Path name resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

[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*.

```
#include <stropts.h>
...
    int fd;
    char *filename = "/tmp/named-STREAM";
    int ret;

    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

None.

FUTURE DIRECTIONS

None.

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.

Issue 6

This function is marked as part of the XSI STREAMS Option Group.

The DESCRIPTION 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.

10502 **NAME**

10503 fchdir — change working directory

10504 **SYNOPSIS**

10505 XSI
```
#include <unistd.h>

int fchdir(int fildes);
```
10506

10507

10508 **DESCRIPTION**

10509 The *fchdir*( ) function has the same effect as *chdir*( ) except that the directory that is to be the new
10510 current working directory is specified by the file descriptor *fildes*.

10511 **RETURN VALUE**

10512 Upon successful completion, *fchdir*( ) shall return 0. Otherwise, it shall return –1 and set *errno* to
10513 indicate the error. On failure the current working directory shall remain unchanged.

10514 **ERRORS**

10515 The *fchdir*( ) function shall fail if:

10516 [EACCES] Search permission is denied for the directory referenced by *fildes*.

10517 [EBADF] The *fildes* argument is not an open file descriptor.

10518 [ENOTDIR] The open file descriptor *fildes* does not refer to a directory.

10519 The *fchdir*( ) may fail if:

10520 [EINTR] A signal was caught during the execution of *fchdir*( ).

10521 [EIO] An I/O error occurred while reading from or writing to the file system.

10522 **EXAMPLES**

10523 None.

10524 **APPLICATION USAGE**

10525 None.

10526 **RATIONALE**

10527 None.

10528 **FUTURE DIRECTIONS**

10529 None.

10530 **SEE ALSO**

10531 *chdir*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>**

10532 **CHANGE HISTORY**

10533 First released in Issue 4, Version 2.

10534 **Issue 5**

10535 Moved from X/OPEN UNIX extension to BASE.

10536 **NAME**

10537 fchmod — change mode of a file

10538 **SYNOPSIS**

10539 `#include <sys/stat.h>`

10540 `int fchmod(int fildes, mode_t mode);`

10541 **DESCRIPTION**

10542 The *fchmod*( ) function has the same effect as *chmod*( ) except that the file whose permissions are
10543 changed is specified by the file descriptor *fildes*.

10544 SHM If *fildes* references a shared memory object, the *fchmod*( ) function need only affect the S_IRUSR,
10545 S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits.

10546 TYM If *fildes* references a typed memory object, the behavior of *fchmod*( ) is unspecified. |

10547 ***Notes to Reviewers*** |

10548 *This section with side shading will not appear in the final copy. - Ed.* |

10549 D3, XSH, ERN 178 suggests adding text as follows: "If fildes refers to a STREAM (which is |
10550 fattached() into the file system name space) the call returns successfully, doing nothing. If fildes |
10551 refers to a stream, <do what?>." |

10552 **RETURN VALUE** |

10553 Upon successful completion, *fchmod*( ) shall return 0. Otherwise, it shall return –1 and set *errno* to
10554 indicate the error.

10555 **ERRORS**

10556 The *fchmod*( ) function shall fail if:

10557 [EBADF] The *fildes* argument is not an open file descriptor. |

10558 [EPERM] The effective user ID does not match the owner of the file and the process |
10559 does not have appropriate privilege.

10560 [EROFS] The file referred to by *fildes* resides on a read-only file system. |

10561 The *fchmod*( ) function may fail if:

10562 XSI [EINTR] The *fchmod*( ) function was interrupted by a signal. |

10563 XSI [EINVAL] The value of the *mode* argument is invalid. |

10564 [EINVAL] The *fildes* argument refers to a pipe and the implementation disallows
10565 execution of *fchmod*( ) on a pipe.

10566 **EXAMPLES**

10567 **Changing the Current Permissions for a File**

10568 The following example shows how to change the permissions for a file named **/home/cnd/mod1**
10569 so that the owner and group have read/write/execute permissions, but the world only has
10570 read/write permissions.

```
10571 #include <sys/stat.h>
10572 #include <fcntl.h>

10573 mode_t mode;
10574 int    fildes;
10575 ...
```

```
        fildes = open("/home/cnd/mod1", O_RDWR);
        fchmod(fildes, S_IRWXU | S_IRWXG | S_IROTH | S_IWOTH);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *chown*( ), *creat*( ), *fcntl*( ), *fstatvfs*( ), *mknod*( ), *open*( ), *read*( ), *stat*( ), *write*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/stat.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE and aligned with *fchmod*( ) in the POSIX Realtime Extension. Specifically, the second paragraph of the DESCRIPTION is added and a second instance of [EINVAL] is defined in the list of optional errors.

Issue 6

The DESCRIPTION is updated for alignment with IEEE Std. 1003.1j-2000 by stating that *fchmod*( ) behavior is unspecified for typed memory objects.

NAME

fchown — change owner and group of a file

SYNOPSIS

```
#include <unistd.h>

int fchown(int fildes, uid_t owner, gid_t group);
```

DESCRIPTION

The *fchown*( ) function has the same effect as *chown*( ) except that the file whose owner and group are changed is specified by the file descriptor *fildes*.

RETURN VALUE

Upon successful completion, *fchown*( ) shall return 0. Otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *fchown*( ) function shall fail if:

[EBADF] The *fildes* argument is not an open file descriptor.

[EPERM] The effective user ID does not match the owner of the file or the process does not have appropriate privilege and _POSIX_CHOWN_RESTRICTED indicates that such privilege is required.

[EROFS] The file referred to by *fildes* resides on a read-only file system.

The *fchown*( ) function may fail if:

[EINVAL] The owner or group ID is not a value supported by the implementation. The *fildes* argument refers to a pipe and the implementation disallows execution of *fchown*( ) on a pipe.

Notes to Reviewers

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

D3, XSH, ERN 177 states that STREAMS ignore the call, but raises a question about AF_UNIX sockets in the file system name space.

[EIO] A physical I/O error has occurred.

[EINTR] The *fchown*( ) function was interrupted by a signal which was caught.

EXAMPLES

Changing the Current Owner of a File

The following example shows how to change the owner of a file named **/home/cnd/mod1** to ‘‘jones’’ and the group to ‘‘cnd’’.

The numeric value for the user ID is obtained by extracting the user ID from the user database entry associated with ‘‘jones’’. Similarly, the numeric value for the group ID is obtained by extracting the group ID from the group database entry associated with ‘‘cnd’’. This example assumes the calling program has appropriate privileges.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <pwd.h>
#include <grp.h>
```

```
10637       struct passwd *pwd;
10638       struct group  *grp;
10639       int            fildes;
10640       ...
10641       fildes = open("/home/cnd/mod1", O_RDWR);
10642       pwd = getpwnam("jones");
10643       grp = getgrnam("cnd");
10644       fchown(fildes, pwd->pw_uid, grp->gr_gid);
```

10645 **APPLICATION USAGE**

10646 None.

10647 **RATIONALE**

10648 None.

10649 **FUTURE DIRECTIONS**

10650 None.

10651 **SEE ALSO**

10652 *chown*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>** |

10653 **CHANGE HISTORY**

10654 First released in Issue 4, Version 2.

10655 **Issue 5**

10656 Moved from X/OPEN UNIX extension to BASE.

10657 **Issue 6**

10658 The following changes were made to align with the IEEE P1003.1a draft standard:

10659 • Clarification is added that a call to *fchown*( ) may not be allowed on a pipe.

10660 The *fchwon*( ) function is now defined as mandatory. |

10661 **NAME**

10662 fclose — close a stream

10663 **SYNOPSIS**

10664 `#include <stdio.h>`

10665 `int fclose(FILE *stream);`

10666 **DESCRIPTION**

10667 CX The functionality described on this reference page is aligned with the ISO C standard. Any
10668 conflict between the requirements described here and the ISO C standard is unintentional. This
10669 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

10670 The *fclose*( ) function shall cause the stream pointed to by *stream* to be flushed and the associated
10671 file to be closed. Any unwritten buffered data for the stream shall be written to the file; any
10672 unread buffered data shall be discarded. Whether or not the call succeeds, the stream shall be
10673 disassociated from the file and any buffer set by the *setbuf*( ) or *setvbuf*( ) function shall be
10674 disassociated from the stream. If the associated buffer was automatically allocated, it shall be
10675 CX deallocated. It shall mark for update the *st_ctime* and *st_mtime* fields of the underlying file, if the
10676 stream was writable, and if buffered data remains that has not yet been written to the file. The
10677 *fclose*( ) function shall perform the equivalent of a *close*( ) on the file descriptor that is associated
10678 with the stream pointed to by *stream*.

10679 After the call to *fclose*( ), any use of *stream* results in undefined behavior.

10680 **RETURN VALUE**

10681 CX Upon successful completion, *fclose*( ) shall return 0; otherwise, it shall return EOF and set *errno* to
10682 indicate the error.

10683 **ERRORS**

10684 The *fclose*( ) function shall fail if:

10685 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
10686 process would be delayed in the write operation.

10687 CX [EBADF] The file descriptor underlying stream is not valid.

10688 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size.

10689 XSI [EFBIG] An attempt was made to write a file that exceeds the process' file size limit.

10690 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
10691 offset maximum associated with the corresponding stream.

10692 CX [EINTR] The *fclose*( ) function was interrupted by a signal.

10693 CX [EIO] The process is a member of a background process group attempting to write
10694 to its controlling terminal, TOSTOP is set, the process is neither ignoring nor
10695 blocking SIGTTOU, and the process group of the process is orphaned. This
10696 error may also be returned under implementation-defined conditions.

10697 CX [ENOSPC] There was no free space remaining on the device containing the file.

10698 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
10699 any process. A SIGPIPE signal shall also be sent to the thread.

10700 The *fclose*( ) function may fail if:

10701 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
10702 capabilities of the device.

10703 **EXAMPLES**

10704 None.

10705 **APPLICATION USAGE**

10706 None.

10707 **RATIONALE**

10708 None.

10709 **FUTURE DIRECTIONS**

10710 None.

10711 **SEE ALSO**

10712 *close*( ), *fopen*( ), *getrlimit*( ), *ulimit*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
10713 **<stdio.h>**

10714 **CHANGE HISTORY**

10715 First released in Issue 1. Derived from Issue 1 of the SVID. |

10716 **Issue 4**

10717 The last sentence of the first paragraph in the DESCRIPTION is changed to say *close*( ) instead of
10718 *fclose*( ). This was an error in Issue 3.

10719 The following paragraph is withdrawn from the DESCRIPTION (by POSIX as well as X/Open)
10720 because of the possibility of causing applications to malfunction, and the impossibility of
10721 implementing these mechanisms for pipes:

10722 > If the file is not already at EOF, and the file is one capable of seeking, the file *offset* of the
10723 > underlying open file description is adjusted so that the next operation on the open file
10724 > description deals with the byte after the last one read from or written to the stream being
10725 > closed.

10726 It is replaced with a statement that any subsequent use of *stream* is undefined.

10727 The [EFBIG] error is marked to indicate the extensions.

10728 **Issue 4, Version 2**

10729 A cross-reference to *getrlimit*( ) is added.

10730 **Issue 5**

10731 Large File Summit extensions are added.

10732 **Issue 6**

10733 Extensions beyond the ISO C standard are now marked.

10734 The following new requirements on POSIX implementations derive from alignment with the
10735 Single UNIX Specification:

10736 - The [EFBIG] error is added as part of the large file support extensions.

10737 - The [ENXIO] optional error condition is added.

10738 The DESCRIPTION is updated to note that the stream and any buffer are disassociated whether |
10739 or not the call succeeds. This is for alignment with the ISO/IEC 9899: 1999 standard. |

10740 **NAME**

10741 fcntl — file control

10742 **SYNOPSIS**

10743 OH
```
#include <unistd.h>
```
10744
```
#include <fcntl.h>
```
10745
```
int fcntl(int fildes, int cmd, ...);
```

10746 **DESCRIPTION**

10747 The *fcntl*( ) function provides for control over open files. The *fildes* argument is a file descriptor.

10748 The available values for *cmd* are defined in **<fcntl.h>**, which include:

10749 F_DUPFD Return a new file descriptor which is the lowest numbered available (that is,
10750 not already open) file descriptor greater than or equal to the third argument,
10751 *arg*, taken as an integer of type **int**. The new file descriptor refers to the same
10752 open file description as the original file descriptor, and shares any locks. The
10753 FD_CLOEXEC flag associated with the new file descriptor is cleared to keep
10754 the file open across calls to one of the *exec* functions.

10755 F_GETFD Get the file descriptor flags defined in **<fcntl.h>** that are associated with the
10756 file descriptor *fildes*. File descriptor flags are associated with a single file
10757 descriptor and do not affect other file descriptors that refer to the same file.

10758 F_SETFD Set the file descriptor flags defined in **<fcntl.h>**, that are associated with *fildes*,
10759 to the third argument, *arg*, taken as type **int**. If the FD_CLOEXEC flag in the
10760 third argument is 0, the file shall remain open across the *exec* functions;
10761 otherwise, the file shall be closed upon successful execution of one of the *exec*
10762 functions.

10763 F_GETFL Get the file status flags and file access modes, defined in **<fcntl.h>**, for the file
10764 description associated with *fildes*. The file access modes can be extracted from
10765 the return value using the mask O_ACCMODE, which is defined in **<fcntl.h>**.
10766 File status flags and file access modes are associated with the file description
10767 and do not affect other file descriptors that refer to the same file with different
10768 open file descriptions.

10769 F_SETFL Set the file status flags, defined in **<fcntl.h>**, for the file description associated
10770 with *fildes* from the corresponding bits in the third argument, *arg*, taken as
10771 type **int**. Bits corresponding to the file access mode and the *oflag* values that
10772 are set in *arg* are ignored. If any bits in *arg* other than those mentioned here are
10773 changed by the application, the result is unspecified.

10774 F_GETOWN If *fildes* refers to a socket, get the process or process group ID specified to
10775 receive SIGURG signals when out-of-band data is available. Positive values
10776 indicate a process ID; negative values, other than –1, indicate a process group
10777 ID. If *fildes* does not refer to a socket, the results are unspecified.

10778 F_SETOWN If *fildes* refers to a socket, set the process or process group ID specified to
10779 receive SIGURG signals when out-of-band data is available, using the value of
10780 the third argument, *arg*, taken as type **int**. Positive values indicate a process
10781 ID; negative values, other than –1, indicate a process group ID. If *fildes* does
10782 not refer to a socket, the results are unspecified.

10783 The following values for *cmd* are available for advisory record locking. Record locking is
10784 supported for regular files, and may be supported for other files.

F_GETLK Get the first lock which blocks the lock description pointed to by the third argument, *arg*, taken as a pointer to type **struct flock**, defined in **<fcntl.h>**. The information retrieved overwrites the information passed to *fcntl*( ) in the structure **flock**. If no lock is found that would prevent this lock from being created, then the structure shall be left unchanged except for the lock type which shall be set to F_UNLCK.

F_SETLK Set or clear a file segment lock according to the lock description pointed to by the third argument, *arg*, taken as a pointer to type **struct flock**, defined in **<fcntl.h>**. F_SETLK is used to establish shared (or read) locks (F_RDLCK) or exclusive (or write) locks (F_WRLCK), as well as to remove either type of lock (F_UNLCK). F_RDLCK, F_WRLCK, and F_UNLCK are defined in **<fcntl.h>**. If a shared or exclusive lock cannot be set, *fcntl*( ) shall return immediately with a return value of –1.

F_SETLKW This command is the same as F_SETLK except that if a shared or exclusive lock is blocked by other locks, the thread shall wait until the request can be satisfied. If a signal that is to be caught is received while *fcntl*( ) is waiting for a region, *fcntl*( ) shall be interrupted. Upon return from the signal handler, *fcntl*( ) shall return –1 with *errno* set to [EINTR], and the lock operation shall not be done.

Additional implementation-defined values for *cmd* may be defined in **<fcntl.h>**. Their names shall start with F_.

When a shared lock is set on a segment of a file, other processes shall be able to set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock shall fail if the file descriptor was not opened with read access.

An exclusive lock shall prevent any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock shall fail if the file descriptor was not opened with write access.

The structure **flock** describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), size (*l_len*), and process ID (*l_pid*) of the segment of the file to be affected.

The value of *l_whence* is {SEEK_SET}, {SEEK_CUR}, or {SEEK_END}, to indicate that the relative offset *l_start* bytes shall be measured from the start of the file, current position, or end of the file, respectively. The value of *l_len* is the number of consecutive bytes to be locked. The value of *l_len* may be negative (where the definition of **off_t** permits negative values of *l_len*). The *l_pid* field is only used with F_GETLK to return the process ID of the process holding a blocking lock. After a successful F_GETLK request, when a blocking lock is found, the values returned in the **flock** structure shall be as follows:

l_type Type of blocking lock found.

l_whence {SEEK_SET}.

l_start Start of the blocking lock.

l_len Length of the blocking lock.

l_pid Process ID of the process that holds the blocking lock.

If the command is F_SETLKW and the process must wait for another process to release a lock, then the range of bytes to be locked shall be determined before the *fcntl*( ) function blocks. If the file size or file descriptor seek offset change while *fcntl*( ) is blocked, this shall not affect the range of bytes locked.

If *l_len* is positive, the area affected starts at *l_start* and ends at *l_start*+ *l_len*−1. If *l_len* is negative, the area affected starts at *l_start*+ *l_len* and ends at *l_start*−1. Locks may start and extend beyond the current end of a file, but the application shall ensure that they are not negative relative to the beginning of the file. A lock shall be set to extend to the largest possible value of the file offset for that file by setting *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 shall be locked.

There shall be at most one type of lock set for each byte in the file. Before a successful return from an F_SETLK or an F_SETLKW request when the calling process has previously existing locks on bytes in the region specified by the request, the previous lock type for each byte in the specified region shall be replaced by the new lock type. As specified above under the descriptions of shared locks and exclusive locks, an F_SETLK or an F_SETLKW request (respectively) shall fail or block when another process has existing locks on bytes in the specified region and the type of any of those locks conflicts with the type specified in the request.

All locks associated with a file for a given process shall be removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock another process' locked region. If the system detects that sleeping until a locked region is unlocked would cause a deadlock, *fcntl*( ) shall fail with an [EDEADLK] error.

SHM When the file descriptor *fildes* refers to a shared memory object, the behavior of *fcntl*( ) shall be the same as for a regular file except the effect of the following values for the argument *cmd* shall be unspecified: F_SETFL, F_GETLK, F_SETLK, and F_SETLKW.

TYM If *fildes* refers to a typed memory object, the result of the *fcntl*( ) function is unspecified.

An unlock (F_UNLCK) request in which *l_len* is non-zero and the offset of the last byte of the requested segment is the maximum value for an object of type **off_t**, when the process has an existing lock in which *l_len* is 0 and which includes the last byte of the requested segment, shall be treated as a request to unlock from the start of the requested segment with an *l_len* equal to 0. Otherwise, an unlock (F_UNLCK) request shall attempt to unlock only the requested segment.

RETURN VALUE

Upon successful completion, the value returned shall depend on *cmd* as follows:

F_DUPFD	A new file descriptor.
F_GETFD	Value of flags defined in **<fcntl.h>**. The return value shall not be negative.
F_SETFD	Value other than –1.
F_GETFL	Value of file status flags and access modes. The return value is not negative.
F_SETFL	Value other than –1.
F_GETLK	Value other than –1.
F_SETLK	Value other than –1.
F_SETLKW	Value other than –1.
F_GETOWN	Value of the socket owner process or process group; this will not be –1.
F_SETOWN	Value other than –1.

Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *fcntl*( ) function shall fail if:

[EACCES] or [EAGAIN]
: The *cmd* argument is F_SETLK; the type of lock (*l_type*) is a shared (F_RDLCK) or exclusive (F_WRLCK) lock and the segment of a file to be locked is already exclusive-locked by another process, or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.

[EBADF]
: The *fildes* argument is not a valid open file descriptor, or the argument *cmd* is F_SETLK or F_SETLKW, the type of lock, *l_type*, is a shared lock (F_RDLCK), and *fildes* is not a valid file descriptor open for reading, or the type of lock *l_type*, is an exclusive lock (F_WRLCK), and *fildes* is not a valid file descriptor open for writing.

[EINTR]
: The *cmd* argument is F_SETLKW and the function was interrupted by a signal.

[EINVAL]
: The *cmd* argument is invalid, or the *cmd* argument is F_DUPFD and *arg* is 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 *fildes* refers to a file that does not support locking.

[EMFILE]
: The argument *cmd* is F_DUPFD and {OPEN_MAX} file descriptors are currently open in the calling process, or no file descriptors greater than or equal to *arg* are available.

[ENOLCK]
: The argument *cmd* is F_SETLK or F_SETLKW and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.

[EOVERFLOW]
: One of the values to be returned cannot be represented correctly.

[EOVERFLOW]
: 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**.

The *fcntl*( ) function may fail if:

[EDEADLK]
: The *cmd* argument is F_SETLKW, the lock is blocked by some lock from another process and putting the calling process to sleep, waiting for that lock to become free would cause a deadlock.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The ellipsis in the SYNOPSIS is the syntax specified by the ISO C standard for a variable number of arguments. It is used because System V uses pointers for the implementation of file locking functions.

The *arg* values to F_GETFD, F_SETFD, F_GETFL, and F_SETFL all represent flag values to allow for future growth. Applications using these functions should do a read-modify-write operation on them, rather than assuming that only the values defined by this volume of IEEE Std. 1003.1-200x are valid. It is a common error to forget this, particularly in the case of F_SETFD.

This volume of IEEE Std. 1003.1-200x permits concurrent read and write access to file data using the *fcntl*( ) function; this is a change from the 1984 /usr/group standard and early proposals. Without concurrency controls, this feature may not be fully utilized without occasional loss of data.

Data losses occur in several ways. One case occurs when several processes try to update the same record, without sequencing controls; several updates may occur in parallel and the last writer ''wins''. Another case is a bit-tree or other internal list-based database that is undergoing reorganization. Without exclusive use to the tree segment by the updating process, other reading processes chance getting lost in the database when the index blocks are split, condensed, inserted, or deleted. While *fcntl*( ) is useful for many applications, it is not intended to be overly general and does not handle the bit-tree example well.

This facility is only required for regular files because it is not appropriate for many devices such as terminals and network connections.

Since *fcntl*( ) works with ''any file descriptor associated with that file, however it is obtained'', the file descriptor may have been inherited through a *fork*( ) or *exec* operation and thus may affect a file that another process also has open.

The use of the open file description to identify what to lock requires extra calls and presents problems if several processes are sharing an open file description, but there are too many implementations of the existing mechanism for this volume of IEEE Std. 1003.1-200x to use different specifications.

Another consequence of this model is that closing any file descriptor for a given file (whether or not it is the same open file description that created the lock) causes the locks on that file to be relinquished for that process. Equivalently, any close for any file/process pair relinquishes the locks owned on that file for that process. But note that while an open file description may be shared through *fork*( ), locks are not inherited through *fork*( ). Yet locks may be inherited through one of the *exec* functions.

The identification of a machine in a network environment is outside of the scope of this volume of IEEE Std. 1003.1-200x. Thus, an *l_sysid* member, such as found in System V, is not included in the locking structure.

Before successful return from an F_SETLK or F_SETLKW request, the previous lock type for each byte in the specified region shall be replaced by the new lock type. This can result in a previously locked region being split into smaller regions. If this would cause the number of regions being held by all processes in the system to exceed a system-imposed limit, the *fcntl*( ) function shall return –1 with *errno* set to [ENOLCK].

Mandatory locking was a major feature of the 1984 /usr/group standard.

For advisory file record locking to be effective, all processes that have access to a file must cooperate and use the advisory mechanism before doing I/O on the file. Enforcement-mode record locking is important when it cannot be assumed that all processes are cooperating. For example, if one user uses an editor to update a file at the same time that a second user executes another process that updates the same file and if only one of the two processes is using advisory locking, the processes are not cooperating. Enforcement-mode record locking would protect against accidental collisions.

Secondly, advisory record locking requires a process using locking to bracket each I/O operation with lock (or test) and unlock operations. With enforcement-mode file and record locking, a process can lock the file once and unlock when all I/O operations have been completed. Enforcement-mode record locking provides a base that can be enhanced; for example, with sharable locks. That is, the mechanism could be enhanced to allow a process to lock a file so

10964 other processes could read it, but none of them could write it.

10965 Mandatory locks were omitted for several reasons:

10966 1. Mandatory lock setting was done by multiplexing the set-group-ID bit in most
10967 implementations; this was confusing, at best.

10968 2. The relationship to file truncation as supported in 4.2 BSD was not well specified.

10969 3. Any publicly readable file could be locked by anyone. Many historical implementations
10970 keep the password database in a publicly readable file. A malicious user could thus
10971 prohibit logins. Another possibility would be to hold open a long-distance telephone line.

10972 4. Some demand-paged historical implementations offer memory mapped files, and
10973 enforcement cannot be done on that type of file.

10974 Since sleeping on a region is interrupted with any signal, *alarm*( ) may be used to provide a
10975 timeout facility in applications requiring it. This is useful in deadlock detection. Because
10976 implementation of full deadlock detection is not always feasible, the [EDEADLK] error was |
10977 made optional.

10978 **FUTURE DIRECTIONS**

10979 None.

10980 **SEE ALSO**

10981 *close*( ), *exec*, *open*( ), *sigaction*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<fcntl.h>**, |
10982 **<signal.h>**, **<unistd.h>**

10983 **CHANGE HISTORY**

10984 First released in Issue 1. Derived from Issue 1 of the SVID. |

10985 **Issue 4**

10986 The **<sys/types.h>** and **<unistd.h>** headers are now marked as optional (OH); these headers do
10987 not need to be included on XSI-conformant systems.

10988 In the DESCRIPTION, sentences describing behavior when *l_len* is negative are marked as an
10989 extension, and the description of locks is corrected to make it a requirement on the application.

10990 The following change is incorporated for alignment with the ISO POSIX-1 standard:

10991 • In the DESCRIPTION, the meaning of a successful F_SETLK or F_SETLKW request is
10992 clarified, after a POSIX Request for Interpretation.

10993 **Issue 5**

10994 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
10995 Threads Extension.

10996 Large File Summit extensions are added.

10997 **Issue 6**

10998 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

10999 The following new requirements on POSIX implementations derive from alignment with the
11000 Single UNIX Specification:

11001 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
11002 required for conforming implementations of previous POSIX specifications, it was not
11003 required for UNIX applications.

11004 • In the DESCRIPTION, sentences describing behavior when *l_len* is negative are now
11005 mandated, and the description of unlock (F_UNLOCK) when *l_len* is non-negative is
11006 mandated.

- In the ERRORS section, the [EINVAL] error condition has the case mandated when the *cmd* is invalid, and two [EOVERFLOW] error conditions are added.

The F_GETOWN and F_SETOWN values are added for sockets.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Clarification is added that the extent of the bytes locked is determined prior to the blocking action.

The DESCRIPTION is updated for alignment with IEEE Std. 1003.1j-2000 by specifying that *fcntl*( ) results are unspecified for typed memory objects.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

11016 **NAME**

11017 fcvt — convert a floating-point number to a string (**LEGACY**)

11018 **SYNOPSIS**

11019 XSI
```
#include <stdlib.h>
```

11020 11021 11022
```
char *fcvt(double value, int ndigit, int *restrict decpt,
    int *restrict sign);
```

11023 **DESCRIPTION**

11024 Refer to *ecvt*( ).

NAME

fdatasync — synchronize the data of a file (**REALTIME**)

SYNOPSIS

SIO
```
#include <unistd.h>

int fdatasync(int fildes);
```

DESCRIPTION

The *fdatasync*( ) function shall force all currently queued I/O operations associated with the file indicated by file descriptor *fildes* to the synchronized I/O completion state.

The functionality is as described for *fsync*( ) (with the symbol _POSIX_SYNCHRONIZED_IO defined), with the exception that all I/O operations shall be completed as defined for synchronized I/O data integrity completion.

RETURN VALUE

If successful, the *fdatasync*( ) function shall return the value 0; otherwise, the function shall return the value −1 and set *errno* to indicate the error. If the *fdatasync*( ) function fails, outstanding I/O operations are not guaranteed to have been completed.

ERRORS

The *fdatasync*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid file descriptor open for writing. |

[EINVAL] This implementation does not support synchronized I/O for this file. |

In the event that any of the queued I/O operations fail, *fdatasync*( ) shall return the error conditions defined for *read*( ) and *write*( ).

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_fsync( ), *fcntl*( ), *fsync*( ), *open*( ), *read*( ), *write*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.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 Synchronized Input and Output option. |

The *fdatasync*( ) function is marked as part of the Synchronized Input and Output option. |

11064 **NAME**

11065 fdetach — detach a name from a STREAMS-based file descriptor (**STREAMS**)

11066 **SYNOPSIS**

11067 XSR
```
#include <stropts.h>
```

11068
```
int fdetach(const char *path);
```

11069

11070 **DESCRIPTION**

11071 The *fdetach*( ) function detaches a STREAMS-based file from the file to which it was attached by a
11072 previous call to *fattach*( ). The *path* argument points to the path name of the attached STREAMS
11073 file. The application shall ensure that the process has appropriate privileges or be the owner of
11074 the file. A successful call to *fdetach*( ) shall cause all path names that named the attached
11075 STREAMS file to again name the file to which the STREAMS file was attached. All subsequent
11076 operations on *path* shall operate on the underlying file and not on the STREAMS file.

11077 All open file descriptions established while the STREAMS file was attached to the file referenced
11078 by *path* shall still refer to the STREAMS file after the *fdetach*( ) has taken effect.

11079 If there are no open file descriptors or other references to the STREAMS file, then a successful
11080 call to *fdetach*( ) shall have the same effect as performing the last *close*( ) on the attached file.

11081 **RETURN VALUE**

11082 Upon successful completion, *fdetach*( ) shall return 0; otherwise, it shall return –1 and set *errno* to
11083 indicate the error.

11084 **ERRORS**

11085 The *fdetach*( ) function shall fail if:

11086 [EACCES] Search permission is denied on a component of the path prefix. |

11087 [EINVAL] The *path* argument names a file that is not currently attached. |

11088 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* |
11089 argument. |

11090 [ENAMETOOLONG] |
11091 The size of a path name exceeds {PATH_MAX} or a path name component is |
11092 longer than {NAME_MAX}. |

11093 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string. |

11094 [ENOTDIR] A component of the path prefix is not a directory. |

11095 [EPERM] The effective user ID is not the owner of *path* and the process does not have |
11096 appropriate privileges. |

11097 The *fdetach*( ) function may fail if:

11098 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during |
11099 resolution of the *path* argument. |

11100 [ENAMETOOLONG] |
11101 Path name resolution of a symbolic link produced an intermediate result
11102 whose length exceeds {PATH_MAX}.

EXAMPLES

Detaching a File

The following example detaches the STREAMS-based file **/tmp/named-STREAM** from the file to which it was attached by a previous, successful call to *fattach*( ). Subsequent calls to open this file refer to the underlying file, not to the STREAMS file.

```
#include <stropts.h>
...
    char *filename = "/tmp/named-STREAM";
    int ret;

    ret = fdetach(filename);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fattach( ), 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.

Issue 6

The DESCRIPTION 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.

11129 **NAME**

11130 fdim, fdimf, fdiml — compute positive difference between two floating-point numbers

11131 **SYNOPSIS**

11132
```
#include <math.h>
```

11133
```
double fdim(double x, double y);
```
11134
```
float fdimf(float x, float y);
```
11135
```
long double fdiml(long double x, long double y);
```

11136 **DESCRIPTION**

11137 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11138 conflict between the requirements described here and the ISO C standard is unintentional. This
11139 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

11140 These functions shall determine the positive difference between their arguments. If *x* is greater
11141 than *y*, *x*–*y* is returned. If *x* is less than or equal to *y*, +0 is returned.

11142 An application wishing to check for error situations should set *errno* to 0 before calling these
11143 functions. If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

11144 **RETURN VALUE**

11145 Upon successful completion, these functions shall return the positive difference value.

11146 If *x* or *y* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

11147 If the magnitude of the result is too large or too small, the numeric result is unspecified and *errno*
11148 may be set to [ERANGE].

11149 **ERRORS**

11150 These functions may fail if:

11151 [EDOM] The value of *x* or *y* is NaN.

11152 [ERANGE] The magnitude of the result is too large or too small.

11153 **EXAMPLES**

11154 None.

11155 **APPLICATION USAGE**

11156 None.

11157 **RATIONALE**

11158 None.

11159 **FUTURE DIRECTIONS**

11160 None.

11161 **SEE ALSO**

11162 *fmax*( ), *fmin*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

11163 **CHANGE HISTORY**

11164 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

11165 **NAME**

11166 fdopen — associate a stream with a file descriptor

11167 **SYNOPSIS**

11168 `#include <stdio.h>`

11169 `FILE *fdopen(int fildes, const char *mode);`

11170 **DESCRIPTION**

11171 The *fdopen*( ) function shall associate a stream with a file descriptor.

11172 The *mode* argument is a character string having one of the following values:

11173 *r* or *rb*	Open a file for reading.
11174 *w* or *wb*	Open a file for writing.
11175 *a* or *ab*	Open a file for writing at end of file.
11176 *r+* or *rb+* or *r+b*	Open a file for update (reading and writing).
11177 *w+* or *wb+* or *w+b*	Open a file for update (reading and writing).
11178 *a+* or *ab+* or *a+b*	Open a file for update (reading and writing) at end of file.

11179 The meaning of these flags is exactly as specified in *fopen*( ), except that modes beginning with *w*
11180 do not cause truncation of the file.

11181 Additional values for the *mode* argument may be supported by an implementation.

11182 The application shall ensure that the mode of the stream as expressed by the *type* argument is
11183 allowed by the file access mode of the open file description to which *fildes* refers. The file
11184 position indicator associated with the new stream is set to the position indicated by the file
11185 offset associated with the file descriptor.

11186 The error and end-of-file indicators for the stream shall be cleared. The *fdopen*( ) function may
11187 cause the *st_atime* field of the underlying file to be marked for update.

11188 SHM If *fildes* refers to a shared memory object, the result of the *fdopen*( ) function is unspecified.

11189 TYM If *fildes* refers to a typed memory object, the result of the *fdopen*( ) function is unspecified.

11190 The *fdopen*( ) function shall preserve the offset maximum previously set for the open file
11191 description corresponding to *fildes*.

11192 **RETURN VALUE**

11193 Upon successful completion, *fdopen*( ) shall return a pointer to a stream; otherwise, a null pointer
11194 shall be returned and *errno* set to indicate the error.

11195 **ERRORS**

11196 The *fdopen*( ) function may fail if:

11197	[EBADF]	The *fildes* argument is not a valid file descriptor. |
11198	[EINVAL]	The *mode* argument is not a valid mode. |
11199	[EMFILE]	{FOPEN_MAX} streams are currently open in the calling process. |
11200	[EMFILE]	{STREAM_MAX} streams are currently open in the calling process. |
11201	[ENOMEM]	Insufficient space to allocate a buffer. |

11202 **EXAMPLES**

11203 None.

11204 **APPLICATION USAGE**

11205 File descriptors are obtained from calls like *open*( ), *dup*( ), *creat*( ), or *pipe*( ), which open files but
11206 do not return streams.

11207 **RATIONALE**

11208 The file descriptor may have been obtained from *open*( ), *creat*( ), *pipe*( ), *dup*( ), or *fcntl*( );
11209 inherited through *fork*( ) or *exec*; or perhaps obtained by implementation-defined means, such as
11210 the 4.3 BSD *socket*( ) call.

11211 The meanings of the *type* arguments of *fdopen*( ) and *fopen*( ) differ. With *fdopen*( ), open for write
11212 (*w* or *w*+) does not truncate, and append (*a* or *a*+) cannot create for writing. There is no need for *b*
11213 in the format due to the equivalence of binary and text files in this volume of
11214 IEEE Std. 1003.1-200x. Although not explicitly required by this volume of IEEE Std. 1003.1-200x,
11215 a good implementation of append (*a*) mode would cause the O_APPEND flag to be set.

11216 **FUTURE DIRECTIONS**

11217 None.

11218 **SEE ALSO**

11219 *fclose*( ), *fopen*( ), *open*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**, Section
11220 2.5.1 (on page 535)

11221 **CHANGE HISTORY**

11222 First released in Issue 1. Derived from Issue 1 of the SVID.

11223 **Issue 4**

11224 In the DESCRIPTION, the use and settings of the *mode* argument are changed to include binary
11225 streams and are marked as extensions.

11226 All errors identified in the ERRORS section are marked as extensions, and the [EMFILE] error is
11227 added.

11228 The APPLICATION USAGE section is added.

11229 The following change is incorporated for alignment with the ISO POSIX-1 standard:

11230 • The type of argument *mode* is changed from **char*** to **const char***.

11231 **Issue 5**

11232 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

11233 Large File Summit extensions are added.

11234 **Issue 6**

11235 The following new requirements on POSIX implementations derive from alignment with the
11236 Single UNIX Specification:

11237 • In the DESCRIPTION, the use and setting of the *mode* argument are changed to include
11238 binary streams.

11239 • In the DESCRIPTION, text is added for large file support to indicate setting of the offset
11240 maximum in the open file description.

11241 • All errors identified in the ERRORS section are added.

11242 • In the DESCRIPTION, text is added that the *fdopen*( ) function may cause *st_atime* to be
11243 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.

11249 **NAME**

11250 feclearexcept — clear floating-point exception

11251 **SYNOPSIS**

11252 `#include <fenv.h>`

11253 `void feclearexcept(int excepts);`

11254 **DESCRIPTION**

11255 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11256 conflict between the requirements described here and the ISO C standard is unintentional. This
11257 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

11258 The *feclearexcept*( ) function shall clear the supported floating-point exceptions represented by
11259 *excepts*.

11260 **RETURN VALUE**

11261 None.

11262 **ERRORS**

11263 No errors are defined.

11264 **EXAMPLES**

11265 None.

11266 **APPLICATION USAGE**

11267 None.

11268 **RATIONALE**

11269 None.

11270 **FUTURE DIRECTIONS**

11271 None.

11272 **SEE ALSO**

11273 *fegetexceptflag*( ), *feraiseexcept*( ), *fesetexceptflag*( ), *fetestexcept*( ), the Base Definitions volume of
11274 IEEE Std. 1003.1-200x, **<fenv.h>**

11275 **CHANGE HISTORY**

11276 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

11277 **NAME**
11278 fegetenv, fesetenv — get and set current floating-point environment

11279 **SYNOPSIS**
11280 `#include <fenv.h>`

11281 `void fegetenv(fenv_t *`*`envp`*`);`
11282 `void fesetenv(const fenv_t *`*`envp`*`);`

11283 **DESCRIPTION**
11284 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11285 conflict between the requirements described here and the ISO C standard is unintentional. This
11286 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

11287 The *fegetenv*( ) function shall store the current floating-point environment in the object pointed to
11288 by *envp*.

11289 The *fesetenv*( ) function shall establish the floating-point environment represented by the object
11290 pointed to by *envp*. The argument *envp* shall point to an object set by a call to *fegetenv*( ) or
11291 *feholdexcept*( ), or equal a floating-point environment macro. The *fesetenv*( ) function does not
11292 raise floating-point exceptions, but only installs the state of the floating-point status flags
11293 represented through its argument.

11294 **RETURN VALUE**
11295 None.

11296 **ERRORS**
11297 No errors are defined.

11298 **EXAMPLES**
11299 None.

11300 **APPLICATION USAGE**
11301 None.

11302 **RATIONALE**
11303 None.

11304 **FUTURE DIRECTIONS**
11305 None.

11306 **SEE ALSO**
11307 *feholdexcept*( ), *feupdateenv*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<fenv.h>**

11308 **CHANGE HISTORY**
11309 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

fegetexceptflag, fesetexceptflag — get and set floating-point status flags

SYNOPSIS

```
#include <fenv.h>

void fegetexceptflag(fexcept_t *flagp, int excepts);
void fesetexceptflag(const fexcept_t *flagp, int excepts);
```

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 *fegetexceptflag*( ) function shall store an implementation-defined representation of the states of the floating-point status flags indicated by the argument *excepts* in the object pointed to by the argument *flagp*.

The *fesetexceptflag*( ) function shall set the floating-point status flags indicated by the argument excepts to the states stored in the object pointed to by *flagp*. The value pointed to by *flagp* shall have been set by a previous call to *fegetexceptflag*( ) whose second argument represented at least those floating-point exceptions represented by the argument *excepts*. This function does not raise floating-point exceptions, but only sets the state of the flags.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *feraiseexcept*( ), *fetestexcept*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<fenv.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

fegetround, fesetround — get and set current rounding direction

SYNOPSIS

```
#include <fenv.h>

int fegetround(void);
int fesetround(int round);
```

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 *fegetround*( ) function shall get the current rounding direction.

The *fesetround*( ) function shall establish the rounding direction represented by its argument *round*. If the argument is not equal to the value of a rounding direction macro, the rounding direction is not changed.

RETURN VALUE

The *fegetround*( ) function shall return the value of the rounding direction macro representing the current rounding direction or a negative value if there is no such rounding direction macro or the current rounding direction is not determinable.

The *fesetround*( ) function shall return a zero value if and only if the argument is equal to a rounding direction macro (that is, if and only if the requested rounding direction was established).

ERRORS

No errors are defined.

EXAMPLES

The following example saves, sets, and restores the rounding direction, reporting an error and aborting if setting the rounding direction fails:

```
#include <fenv.h>
#include <assert.h>
void f(int round_dir)
{
    #pragma STDC FENV_ACCESS ON
    int save_round;
    int setround_ok;
    save_round = fegetround();
    setround_ok = fesetround(round_dir);
    assert(setround_ok == 0);
    /* ... */
    fesetround(save_round);
    /* ... */
}
```

APPLICATION USAGE

None.

RATIONALE

None.

11389 **FUTURE DIRECTIONS**

11390 None.

11391 **SEE ALSO**

11392 The Base Definitions volume of IEEE Std. 1003.1-200x, **<fenv.h>**

11393 **CHANGE HISTORY**

11394 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11395 **NAME**

11396 feholdexcept — save current floating-point environment

11397 **SYNOPSIS**

11398 `#include <fenv.h>`

11399 `int feholdexcept(fenv_t *`*`envp`*`);`

11400 **DESCRIPTION**

11401 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11402 conflict between the requirements described here and the ISO C standard is unintentional. This
11403 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

11404 The *feholdexcept*( ) function shall save the current floating-point environment in the object
11405 pointed to by *envp*, clear the floating-point status flags, and then install a non-stop (continue on
11406 floating-point exceptions) mode, if available, for all floating-point exceptions.

11407 **RETURN VALUE**

11408 The *feholdexcept*( ) function shall return zero if and only if non-stop floating-point exception
11409 handling was successfully installed.

11410 **ERRORS**

11411 No errors are defined.

11412 **EXAMPLES**

11413 None.

11414 **APPLICATION USAGE**

11415 None.

11416 **RATIONALE**

11417 The *feholdexcept*( ) function should be effective on typical IEC 60559:1989 standard
11418 implementations which have the default non-stop mode and at least one other mode for trap
11419 handling or aborting. If the implementation provides only the non-stop mode, then installing the
11420 non-stop mode is trivial.

11421 **FUTURE DIRECTIONS**

11422 None.

11423 **SEE ALSO**

11424 *fegetenv*( ), *fesetenv*( ), *feupdateenv*( ), the Base Definitions volume of IEEE Std. 1003.1-200x,
11425 **<fenv.h>**

11426 **CHANGE HISTORY**

11427 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11428 **NAME**

11429 feof — test end-of-file indicator on a stream

11430 **SYNOPSIS**

11431 `#include <stdio.h>`

11432 `int feof(FILE *stream);`

11433 **DESCRIPTION**

11434 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11435 conflict between the requirements described here and the ISO C standard is unintentional. This
11436 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

11437 The *feof*( ) function shall test the end-of-file indicator for the stream pointed to by *stream*.

11438 **RETURN VALUE**

11439 The *feof*( ) function shall return non-zero if and only if the end-of-file indicator is set for *stream*.

11440 **ERRORS**

11441 No errors are defined.

11442 **EXAMPLES**

11443 None.

11444 **APPLICATION USAGE**

11445 None.

11446 **RATIONALE**

11447 None.

11448 **FUTURE DIRECTIONS**

11449 None.

11450 **SEE ALSO**

11451 *clearerr*( ), *ferror*( ), *fopen*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>** |

11452 **CHANGE HISTORY**

11453 First released in Issue 1. Derived from Issue 1 of the SVID. |

11454 **Issue 4**

11455 The ERRORS section is rewritten, such that no error return values are now defined for this |
11456 function.

11457 **NAME**

11458 feraiseexcept — raise floating-point exception

11459 **SYNOPSIS**

11460 `#include <fenv.h>`

11461 `void feraiseexcept(int excepts);`

11462 **DESCRIPTION**

11463 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11464 conflict between the requirements described here and the ISO C standard is unintentional. This
11465 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

11466 The *feraiseexcept*( ) function shall raise the supported floating-point exceptions represented by
11467 the argument *excepts*. The order in which these floating-point exceptions are raised is
11468 unspecified. Whether the *feraiseexcept*( ) function additionally raises the inexact floating-point
11469 exception whenever it raises the overflow or underflow floating-point exception is
11470 implementation-defined.

11471 **RETURN VALUE**

11472 None.

11473 **ERRORS**

11474 No errors are defined.

11475 **EXAMPLES**

11476 None.

11477 **APPLICATION USAGE**

11478 The effect is intended to be similar to that of floating-point exceptions raised by arithmetic
11479 operations. Hence, enabled traps for floating-point exceptions raised by this function are taken.

11480 **RATIONALE**

11481 Raising overflow or underflow is allowed to also raise inexact because on some architectures the
11482 only practical way to raise an exception is to execute an instruction that has the exception as a
11483 side effect. The function is not restricted to accept only valid coincident expressions for atomic
11484 operations, so the function can be used to raise exceptions accrued over several operations.

11485 **FUTURE DIRECTIONS**

11486 None.

11487 **SEE ALSO**

11488 *feclearexcept*( ), *fegetexceptflag*( ), *fesetexceptflag*( ), *fetestexcept*( ), the Base Definitions volume of
11489 IEEE Std. 1003.1-200x, **<fenv.h>**

11490 **CHANGE HISTORY**

11491 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

11492 **NAME**

11493 ferror — test error indicator on a stream

11494 **SYNOPSIS**

11495 `#include <stdio.h>`

11496 `int ferror(FILE *stream);`

11497 **DESCRIPTION**

11498 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11499 conflict between the requirements described here and the ISO C standard is unintentional. This
11500 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

11501 The *ferror*( ) function shall test the error indicator for the stream pointed to by *stream*.

11502 **RETURN VALUE**

11503 The *ferror*( ) function shall return non-zero if and only if the error indicator is set for *stream*.

11504 **ERRORS**

11505 No errors are defined.

11506 **EXAMPLES**

11507 None.

11508 **APPLICATION USAGE**

11509 None.

11510 **RATIONALE**

11511 None.

11512 **FUTURE DIRECTIONS**

11513 None.

11514 **SEE ALSO**

11515 *clearerr*( ), *feof*( ), *fopen*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>** |

11516 **CHANGE HISTORY**

11517 First released in Issue 1. Derived from Issue 1 of the SVID. |

11518 **Issue 4**

11519 The ERRORS section is rewritten, such that no error return values are now defined for this |
11520 function.

11521 **NAME**
11522 fesetenv — set current floating-point environment

11523 **SYNOPSIS**
11524 `#include <fenv.h>`

11525 `void fesetenv(const fenv_t *`*`envp`*`);`

11526 **DESCRIPTION**
11527 Refer to *fegetenv*( ).

11528 **NAME**

11529 fesetexceptflag — set floating-point status flags

11530 **SYNOPSIS**

11531 `#include <fenv.h>`

11532 `void fesetexceptflag(const fexcept_t *`*`flagp`*`, int `*`excepts`*`);`

11533 **DESCRIPTION**

11534 Refer to *fegetexceptflag*( ).

11535 **NAME**

11536 fesetround — set current rounding direction

11537 **SYNOPSIS**

11538 `#include <fenv.h>`

11539 `int fesetround(int` *`round`*`);`

11540 **DESCRIPTION**

11541 Refer to *fegetround*( ).

11542 **NAME**

11543 fetestexcept — test floating-point exception flags

11544 **SYNOPSIS**

11545
```
#include <fenv.h>
```

11546
```
int fetestexcept(int excepts);
```

11547 **DESCRIPTION**

11548 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11549 conflict between the requirements described here and the ISO C standard is unintentional. This
11550 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

11551 The *fetestexcept*( ) function shall determine which of a specified subset of the floating-point
11552 exception flags are currently set. The *excepts* argument specifies the floating-point status flags to
11553 be queried.

11554 **RETURN VALUE**

11555 The *fetestexcept*( ) function shall return the value of the bitwise-inclusive OR of the floating-point
11556 exception macros corresponding to the currently set floating-point exceptions included in
11557 *excepts*.

11558 **ERRORS**

11559 No errors are defined.

11560 **EXAMPLES**

11561 The following example calls function *f*( ) if an invalid exception is set, and then function *g*( ) if an
11562 overflow exception is set:

```
11563     #include <fenv.h>
11564     /* ... */
11565     {
11566         #pragma STDC FENV_ACCESS ON
11567         int set_excepts;
11568         feclearexcept(FE_INVALID | FE_OVERFLOW);
11569         // maybe raise exceptions
11570         set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
11571         if (set_excepts & FE_INVALID) f();
11572         if (set_excepts & FE_OVERFLOW) g();
11573         /* ... */
11574     }
```

11575 **APPLICATION USAGE**

11576 None.

11577 **RATIONALE**

11578 None.

11579 **FUTURE DIRECTIONS**

11580 None.

11581 **SEE ALSO**

11582 *feclearexcept*( ), *fegetexceptflag*( ), *feraiseexcept*( ), the Base Definitions volume of
11583 IEEE Std. 1003.1-200x, **<fenv.h>**

11584 **CHANGE HISTORY**
11585 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11586 **NAME**

11587 feupdateenv — update floating-point environment

11588 **SYNOPSIS**

11589
```
#include <fenv.h>
```

11590
```
void feupdateenv(const fenv_t *envp);
```

11591 **DESCRIPTION**

11592 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11593 conflict between the requirements described here and the ISO C standard is unintentional. This
11594 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

11595 The *feupdateenv*( ) function shall save the currently raised floating-point exceptions in its
11596 automatic storage, install the floating-point environment represented by the object pointed to by
11597 *envp*, and then raise the saved floating-point exceptions. The argument *envp* shall point to an
11598 object set by a call to *feholdexcept*( ) or *fegetenv*( ), or equal a floating-point environment macro.

11599 **RETURN VALUE**

11600 None.

11601 **ERRORS**

11602 No errors are defined.

11603 **EXAMPLES**

11604 The following example shows sample code to hide spurious underflow floating-point
11605 exceptions:

```
11606   #include <fenv.h>
11607   double f(double x)
11608   {
11609       #pragma STDC FENV_ACCESS ON
11610       double result;
11611       fenv_t save_env;
11612       feholdexcept(&save_env);
11613       // compute result
11614       if (/* test spurious underflow */)
11615       feclearexcept(FE_UNDERFLOW);
11616       feupdateenv(&save_env);
11617       return result;
11618   }
```

11619 **APPLICATION USAGE**

11620 None.

11621 **RATIONALE**

11622 None.

11623 **FUTURE DIRECTIONS**

11624 None.

11625 **SEE ALSO**

11626 *fegetenv*( ), *feholdexcept*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<fenv.h>**

11627 **CHANGE HISTORY**

11628 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

11629 **NAME**

11630 fflush — flush a stream

11631 **SYNOPSIS**

11632 `#include <stdio.h>`

11633 `int fflush(FILE *stream);`

11634 **DESCRIPTION**

11635 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11636 conflict between the requirements described here and the ISO C standard is unintentional. This
11637 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

11638 If *stream* points to an output stream or an update stream in which the most recent operation was
11639 CX not input, *fflush*( ) causes any unwritten data for that stream to be written to the file, and the
11640 *st_ctime* and *st_mtime* fields of the underlying file are marked for update.

11641 If *stream* is a null pointer, *fflush*( ) shall perform this flushing action on all streams for which the
11642 behavior is defined above.

11643 **RETURN VALUE**

11644 Upon successful completion, *fflush*( ) shall return 0; otherwise, it shall set the error indicator for |
11645 CX the stream, return EOF, and set *errno* to indicate the error. |

11646 **ERRORS**

11647 The *fflush*( ) function shall fail if:

11648 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the |
11649 process would be delayed in the write operation.

11650 CX [EBADF] The file descriptor underlying *stream* is not valid. |

11651 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size. |

11652 XSI [EFBIG] An attempt was made to write a file that exceeds the process' file size limit. |

11653 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the |
11654 offset maximum associated with the corresponding stream.

11655 CX [EINTR] The *fflush*( ) function was interrupted by a signal. |

11656 CX [EIO] The process is a member of a background process group attempting to write |
11657 to its controlling terminal, TOSTOP is set, the process is neither ignoring nor
11658 blocking SIGTTOU, and the process group of the process is orphaned. This
11659 error may also be returned under implementation-defined conditions. |

11660 CX [ENOSPC] There was no free space remaining on the device containing the file. |

11661 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by |
11662 any process. A SIGPIPE signal shall also be sent to the thread.

11663 The *fflush*( ) function may fail if:

11664 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the |
11665 capabilities of the device.

EXAMPLES

Sending Prompts to Standard Output

The following example uses *printf*( ) calls to print a series of prompts for information the user must enter from standard input. The *fflush*( ) calls force the output to standard output. The *fflush*( ) function is used because standard output is usually buffered and the prompt may not immediately be printed on the output or terminal. The *gets*( ) calls read strings from standard input and place the results in variables, for use later in the program

```
#include <stdio.h>
...
char user[100];
char oldpasswd[100];
char newpasswd[100];
...
printf("User name: ");
fflush(stdout);
gets(user);
printf("Old password: ");
fflush(stdout);
gets(oldpasswd);
printf("New password: ");
fflush(stdout);
gets(newpasswd);
...
```

APPLICATION USAGE

None.

RATIONALE

Data buffered by the system may make determining the validity of the position of the current file descriptor impractical. Thus, enforcing the repositioning of the file descriptor after *fflush*( ) on streams open for *read*( ) is not mandated by IEEE Std. 1003.1-200x.

FUTURE DIRECTIONS

None.

SEE ALSO

getrlimit( ), *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 4

The following change is incorporated for alignment with the ISO C standard:

- The DESCRIPTION is changed to describe the behavior of *fflush*( ) if *stream* is a null pointer.

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The following two paragraphs are withdrawn from the DESCRIPTION (by POSIX as well as X/Open) because of the possibility of causing applications to malfunction, and the impossibility of implementing these mechanisms for pipes:

If the stream is open for reading, any unread data buffered in the stream is discarded.

For a stream open for reading, if the file is not already at EOF, and the file is one capable of seeking, the file offset of the underlying open file description is adjusted so that the next operation on the open file description deals with the byte after the last one read from, or written to, the stream being flushed.

- The [EFBIG] error is marked to indicate the extensions.

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now 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 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.

11724 **NAME**

11725 ffs — find first set bit

11726 ***Notes to Reviewers***

11727 *This section with side shading will not appear in the final copy. - Ed.*

11728 This function or these functions are recommended to become mandatory parts of POSIX.1 in the
11729 next draft.

11730 **SYNOPSIS**

11731 XSI
```
#include <strings.h>
```

11732
```
int ffs(int i);
```

11733

11734 **DESCRIPTION**

11735 The *ffs*( ) function shall find the first bit set (beginning with the least significant bit) in *i*, and
11736 return the index of that bit. Bits are numbered starting at one (the least significant bit).

11737 **RETURN VALUE**

11738 The *ffs*( ) function shall return the index of the first bit set. If *i* is 0, then *ffs*( ) shall return 0.

11739 **ERRORS**

11740 No errors are defined.

11741 **EXAMPLES**

11742 None.

11743 **APPLICATION USAGE**

11744 None.

11745 **RATIONALE**

11746 None.

11747 **FUTURE DIRECTIONS**

11748 None.

11749 **SEE ALSO**

11750 The Base Definitions volume of IEEE Std. 1003.1-200x, **<strings.h>**

11751 **CHANGE HISTORY**

11752 First released in Issue 4, Version 2.

11753 **Issue 5**

11754 Moved from X/OPEN UNIX extension to BASE.

11755 **NAME**

11756 fgetc — get a byte from a stream

11757 **SYNOPSIS**

11758 `#include <stdio.h>`

11759 `int fgetc(FILE *stream);`

11760 **DESCRIPTION**

11761 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11762 conflict between the requirements described here and the ISO C standard is unintentional. This
11763 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

11764 If the end-of-file indicator for the input stream pointed to by *stream* is not set and a next
11765 character is present, the *fgetc*( ) function obtains the next byte (if present) as an **unsigned char**
11766 converted to an **int**, from the input stream pointed to by *stream*, and advances the associated file
11767 position indicator for the stream (if defined).

11768 CX The *fgetc*( ) function may mark the *st_atime* field of the file associated with *stream* for update. The
11769 *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ),
11770 *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns
11771 data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

11772 **RETURN VALUE**

11773 Upon successful completion, *fgetc*( ) shall return the next byte from the input stream pointed to
11774 by *stream*. If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the
11775 end-of-file indicator for the stream shall be set and *fgetc*( ) shall return EOF. If a read error occurs,
11776 CX the error indicator for the stream shall be set, *fgetc*( ) shall return EOF, and shall set *errno* to
11777 indicate the error.

11778 **ERRORS**

11779 The *fgetc*( ) function shall fail if data needs to be read and:

11780 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
11781 process would be delayed in the *fgetc*( ) operation.

11782 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
11783 reading.

11784 CX [EINTR] The read operation was terminated due to the receipt of a signal, and no data
11785 was transferred.

11786 CX [EIO] A physical I/O error has occurred, or the process is in a background process
11787 group attempting to read from its controlling terminal, and either the process
11788 is ignoring or blocking the SIGTTIN signal or the process group is orphaned.
11789 This error may also be generated for implementation-defined reasons.

11790 CX [EOVERFLOW] The file is a regular file and an attempt was made to read at or beyond the
11791 offset maximum associated with the corresponding stream.

11792 The *fgetc*( ) function may fail if:

11793 CX [ENOMEM] Insufficient storage space is available.

11794 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
11795 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*(), *fopen*(), *getchar*(), *getc*(), 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 4

In the DESCRIPTION:

- The text is changed to make it clear that the function returns a byte value.
- The list of functions that may cause the *st_atime* field to be updated is revised.

In the ERRORS section, text is added to indicate that error returns are only generated when data needs to be read into the stream buffer.

Also in the ERRORS section, in previous issues generation of the [EIO] error depended on whether or not an implementation supported Job Control. This functionality is now defined as mandatory.

The [ENXIO] and [ENOMEM] errors are marked as extensions.

In the APPLICATION USAGE section, text is added to indicate how an application can distinguish between an error condition and an end-of-file condition.

The description of [EINTR] is amended.

Issue 4, Version 2

In the ERRORS section, the description of [EIO] is updated to include the case where a physical I/O error occurs.

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now 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.

11842 **NAME**

11843 fgetpos — get current file position information

11844 **SYNOPSIS**

11845 `#include <stdio.h>`

11846 `int fgetpos(FILE *restrict stream, fpos_t *restrict pos);`

11847 **DESCRIPTION**

11848 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11849 conflict between the requirements described here and the ISO C standard is unintentional. This
11850 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

11851 The *fgetpos*( ) function shall store the current value of the file position indicator for the stream
11852 pointed to by *stream* in the object pointed to by *pos*. The value stored contains unspecified
11853 information usable by *fsetpos*( ) for repositioning the stream to its position at the time of the call
11854 to *fgetpos*( ).

11855 **RETURN VALUE**

11856 Upon successful completion, *fgetpos*( ) shall return 0; otherwise, it shall return a non-zero value
11857 and set *errno* to indicate the error.

11858 **ERRORS**

11859 The *fgetpos*( ) function shall fail if:

11860 CX [EOVERFLOW] The current value of the file position cannot be represented correctly in an
11861 object of type **fpos_t**.

11862 The *fgetpos*( ) function may fail if:

11863 CX [EBADF] The file descriptor underlying *stream* is not valid.

11864 CX [ESPIPE] The file descriptor underlying *stream* is associated with a pipe, FIFO, or socket.

11865

11866 **EXAMPLES**

11867 None.

11868 **APPLICATION USAGE**

11869 None.

11870 **RATIONALE**

11871 None.

11872 **FUTURE DIRECTIONS**

11873 None.

11874 **SEE ALSO**

11875 *fopen*( ), *ftell*( ), *rewind*( ), *ungetc*( ), the Base Definitions volume of IEEE Std. 1003.1-200x,
11876 **<stdio.h>**

11877 **CHANGE HISTORY**

11878 First released in Issue 4. Derived from the ISO C standard.

11879 **Issue 5**

11880 Large File Summit extensions are added.

11881 **Issue 6**

11882 Extensions beyond the ISO C standard are now marked.

11883 The following new requirements on POSIX implementations derive from alignment with the
11884 Single UNIX Specification:

- The [EIO] mandatory error condition is added.
- 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. |

11889 **NAME**

11890 fgets — get a string from a stream

11891 **SYNOPSIS**

11892 ```
#include <stdio.h>
```

11893 ```
char *fgets(char *restrict s, int n, FILE *restrict stream);
```

11894 **DESCRIPTION**

11895 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11896 conflict between the requirements described here and the ISO C standard is unintentional. This
11897 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

11898 The *fgets*( ) function shall read bytes from *stream* into the array pointed to by *s*, until *n*−1 bytes
11899 are read, or a <newline> character is read and transferred to *s*, or an end-of-file condition is
11900 encountered. The string is then terminated with a null byte.

11901 CX The *fgets*( ) function may mark the *st_atime* field of the file associated with *stream* for update. The
11902 *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ),
11903 *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns
11904 data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

11905 **RETURN VALUE**

11906 Upon successful completion, *fgets*( ) shall return *s*. If the stream is at end-of-file, the end-of-file
11907 indicator for the stream shall be set and *fgets*( ) shall return a null pointer. If a read error occurs,
11908 CX the error indicator for the stream shall be set, *fgets*( ) shall return a null pointer, and shall set
11909 *errno* to indicate the error.

11910 **ERRORS**

11911 Refer to *fgetc*( ).

11912 **EXAMPLES**

11913 **Reading Input**

11914 The following example uses *fgets*( ) to read each line of input. {LINE_MAX}, which defines the
11915 maximum size of the input line, is defined in the **<limits.h>** header.

11916 ```
#include <stdio.h>
11917 ...
11918 char line[LINE_MAX];
11919 ...
11920 while (fgets(line, LINE_MAX, fp) != NULL) {
11921 ...
11922 }
11923 ...
```

11924 **APPLICATION USAGE**

11925 None.

11926 **RATIONALE**

11927 None.

11928 **FUTURE DIRECTIONS**

11929 None.
```

11930 **SEE ALSO**

11931 *fopen*( ), *fread*( ), *gets*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**

11932 **CHANGE HISTORY**

11933 First released in Issue 1. Derived from Issue 1 of the SVID.

11934 **Issue 4**

11935 In the DESCRIPTION, the text is changed to make it clear that the function reads bytes rather
11936 than (possibly multi-byte) characters, and the list of functions that may cause the *st_atime* field to
11937 be updated is revised.

11938 **Issue 6**

11939 Extensions beyond the ISO C standard are now marked.

11940 The prototype for *fgets*( ) is changed for alignment with the ISO / IEC 9899: 1999 standard.

11941 **NAME**

11942 fgetwc — get a wide-character code from a stream

11943 **SYNOPSIS**

11944 ```
#include <stdio.h>
```

11945 ```
#include <wchar.h>
```

11946 ```
wint_t fgetwc(FILE *stream);
```

11947 **DESCRIPTION**

11948 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11949 conflict between the requirements described here and the ISO C standard is unintentional. This
11950 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

11951 The *fgetwc*( ) function shall obtain the next character (if present) from the input stream pointed to
11952 by *stream*, convert that to the corresponding wide-character code, and advance the associated
11953 file position indicator for the stream (if defined).

11954 If an error occurs, the resulting value of the file position indicator for the stream is
11955 indeterminate.

11956 CX The *fgetwc*( ) function may mark the *st_atime* field of the file associated with *stream* for update.
11957 The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ),
11958 *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns
11959 data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

11960 **RETURN VALUE**

11961 Upon successful completion, the *fgetwc*( ) function shall return the wide-character code of the
11962 character read from the input stream pointed to by *stream* converted to a type **wint_t**. If the
11963 stream is at end-of-file, the end-of-file indicator for the stream shall be set and *fgetwc*( ) shall
11964 return WEOF. If a read error occurs, the error indicator for the stream shall be set, *fgetwc*( ) shall
11965 CX return WEOF, and shall set *errno* to indicate the error.

11966 **ERRORS**

11967 The *fgetwc*( ) function shall fail if data needs to be read and:

11968 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the process would be delayed in the *fgetwc*( ) operation.

11970 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for reading.

11972 CX [EINTR] The read operation was terminated due to the receipt of a signal, and no data was transferred.

11974 CX [EIO] A physical I/O error has occurred, or the process is in a background process group attempting to read from its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group is orphaned. This error may also be generated for implementation-defined reasons.

11978 CX [EOVERFLOW] The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the corresponding stream.

11980 The *fgetwc*( ) function may fail if:

11981 CX [ENOMEM] Insufficient storage space is available.

11982 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.
```

11984 CX [EILSEQ] The data obtained from the input stream does not form a valid character.

11985 **EXAMPLES**

11986 None.

11987 **APPLICATION USAGE**

11988 The *ferror*( ) or *feof*( ) functions must be used to distinguish between an error condition and an
11989 end-of-file condition.

11990 **RATIONALE**

11991 None.

11992 **FUTURE DIRECTIONS**

11993 None.

11994 **SEE ALSO**

11995 *feof*( ), *ferror*( ), *fopen*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**,
11996 **<wchar.h>**

11997 **CHANGE HISTORY**

11998 First released in Issue 4. Derived from the MSE working draft.

11999 **Issue 4, Version 2**

12000 In the ERRORS section, the description of [EIO] is updated to include the case where a physical
12001 I/O error occurs.

12002 **Issue 5**

12003 The Optional Header (OH) marking is removed from **<stdio.h>**.

12004 Large File Summit extensions are added.

12005 **Issue 6**

12006 Extensions beyond the ISO C standard are now marked.

12007 The following new requirements on POSIX implementations derive from alignment with the
12008 Single UNIX Specification:

12009 - The [EIO] and [EOVERFLOW] mandatory error conditions are added.

12010 - The [ENOMEM], [ENXIO], and [EILSEQ] optional error conditions are added.

12011 **NAME**

12012 fgetws — get a wide-character string from a stream

12013 **SYNOPSIS**

12014 ```
#include <stdio.h>
```

12015 ```
#include <wchar.h>
```

12016 ```
wchar_t *fgetws(wchar_t *restrict ws, int n,
```

12017 ```
    FILE *restrict stream);
```

12018 **DESCRIPTION**

12019 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12020 conflict between the requirements described here and the ISO C standard is unintentional. This
12021 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

12022 The *fgetws*( ) function shall read characters from the *stream*, convert these to the corresponding
12023 wide-character codes, place them in the **wchar_t** array pointed to by *ws*, until *n*−1 characters are
12024 read, or a <newline> character is read, converted, and transferred to *ws*, or an end-of-file
12025 condition is encountered. The wide-character string, *ws*, is then terminated with a null wide-
12026 character code.

12027 If an error occurs, the resulting value of the file position indicator for the stream is
12028 indeterminate.

12029 CX The *fgetws*( ) function may mark the *st_atime* field of the file associated with *stream* for update.
12030 The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ),
12031 *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns
12032 data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

12033 **RETURN VALUE**

12034 Upon successful completion, *fgetws*( ) shall return *ws*. If the stream is at end-of-file, the end-of-
12035 file indicator for the stream shall be set and *fgetws*( ) shall return a null pointer. If a read error
12036 CX occurs, the error indicator for the stream shall be set, *fgetws*( ) shall return a null pointer, and
12037 shall set *errno* to indicate the error.

12038 **ERRORS**

12039 Refer to *fgetwc*( ).

12040 **EXAMPLES**

12041 None.

12042 **APPLICATION USAGE**

12043 None.

12044 **RATIONALE**

12045 None.

12046 **FUTURE DIRECTIONS**

12047 None.

12048 **SEE ALSO**

12049 *fopen*( ), *fread*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**, **<wchar.h>**

12050 **CHANGE HISTORY**

12051 First released in Issue 4. Derived from the MSE working draft.

12052 **Issue 5**

12053 The Optional Header (OH) marking is removed from **<stdio.h>**.

12054 **Issue 6**

12055 Extensions beyond the ISO C standard are now marked. |

12056 The prototype for *fgetws*( ) is changed for alignment with the ISO / IEC 9899: 1999 standard. |

12057 **NAME**

12058 fileno — map a stream pointer to a file descriptor

12059 **SYNOPSIS**

12060 `#include <stdio.h>`

12061 `int fileno(FILE *stream);`

12062 **DESCRIPTION**

12063 The *fileno*( ) function shall return the integer file descriptor associated with the stream pointed to
12064 by *stream*.

12065 **RETURN VALUE**

12066 Upon successful completion, *fileno*( ) shall return the integer value of the file descriptor
12067 associated with *stream*. Otherwise, the value –1 shall be returned and *errno* set to indicate the
12068 error.

12069 **ERRORS**

12070 The *fileno*( ) function may fail if:

12071 [EBADF] The *stream* argument is not a valid stream. |

12072 **EXAMPLES**

12073 None.

12074 **APPLICATION USAGE**

12075 None.

12076 **RATIONALE**

12077 Without some specification of which file descriptors are associated with these streams, it is
12078 impossible for an application to set up the streams for another application it starts with *fork*( )
12079 and *exec*. In particular, it would not be possible to write a portable version of the *sh* command
12080 interpreter (although there may be other constraints that would prevent that portability).

12081 **FUTURE DIRECTIONS**

12082 None.

12083 **SEE ALSO**

12084 *fdopen*( ), *fopen*( ), *stdin*, the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**, Section |
12085 2.5.1 (on page 535)

12086 **CHANGE HISTORY**

12087 First released in Issue 1. Derived from Issue 1 of the SVID. |

12088 **Issue 4**

12089 The [EBADF] error is marked as an extension.

12090 **Issue 6**

12091 The following new requirements on POSIX implementations derive from alignment with the
12092 Single UNIX Specification:

12093 • The [EBADF] optional error condition is added.

12094 **NAME**

12095 flockfile, ftrylockfile, funlockfile — stdio locking functions

12096 **SYNOPSIS**

12097 TSF
```
#include <stdio.h>

void flockfile(FILE *file);
int ftrylockfile(FILE *file);
void funlockfile(FILE *file);
```
12101

12102 **DESCRIPTION**

12103 The *flockfile*( ), *ftrylockfile*( ), and *funlockfile*( ) functions provide for explicit application-level
12104 locking of stdio (**FILE***) objects. These functions can be used by a thread to delineate a sequence
12105 of I/O statements that are executed as a unit.

12106 The *flockfile*( ) function is used by a thread to acquire ownership of a (**FILE***) object.

12107 The *ftrylockfile*( ) function is used by a thread to acquire ownership of a (**FILE***) object if the
12108 object is available; *ftrylockfile*( ) is a non-blocking version of *flockfile*( ).

12109 The *funlockfile*( ) function is used to relinquish the ownership granted to the thread. The
12110 behavior is undefined if a thread other than the current owner calls the *funlockfile*( ) function.

12111 Logically, there is a lock count associated with each (**FILE***) object. This count is implicitly
12112 initialized to zero when the (**FILE***) object is created. The (**FILE***) object is unlocked when the
12113 count is zero. When the count is positive, a single thread owns the (**FILE***) object. When the
12114 *flockfile*( ) function is called, if the count is zero or if the count is positive and the caller owns the
12115 (**FILE***) object, the count is incremented. Otherwise, the calling thread is suspended, waiting for
12116 the count to return to zero. Each call to *funlockfile*( ) decrements the count. This allows matching
12117 calls to *flockfile*( ) (or successful calls to *ftrylockfile*( )) and *funlockfile*( ) to be nested.

12118 All functions that reference (**FILE***) objects shall behave as if they use *flockfile*( ) and *funlockfile*( )
12119 internally to obtain ownership of these (**FILE***) objects.

12120 **RETURN VALUE**

12121 None for *flockfile*( ) and *funlockfile*( ). The *ftrylockfile*( ) function shall return zero for success and
12122 non-zero to indicate that the lock cannot be acquired.

12123 **ERRORS**

12124 No errors are defined.

12125 **EXAMPLES**

12126 None.

12127 **APPLICATION USAGE**

12128 Applications using these functions may be subject to priority inversion, as discussed in the Base
12129 Definitions volume of IEEE Std. 1003.1-200x, Section 3.287, Priority Inversion.

12130 **RATIONALE**

12131 The *flockfile*( ) and *funlockfile*( ) functions provide an orthogonal mutual exclusion lock for each
12132 **FILE**. The *ftrylockfile*( ) function provides a non-blocking attempt to acquire a file lock,
12133 analogous to *pthread_mutex_trylock*( ).

12134 These locks behave as if they are the same as those used internally by *stdio* for thread-safety.
12135 This both provides thread-safety of these functions without requiring a second level of internal
12136 locking and allows functions in *stdio* to be implemented in terms of other *stdio* functions.

12137 Application writers and implementors should be aware that there are potential deadlock
12138 problems on **FILE** objects. For example, the line-buffered flushing semantics of *stdio* (requested

12139 via {_IOLBF}) require that certain input operations sometimes cause the buffered contents of |
12140 implementation-defined line-buffered output streams to be flushed. If two threads each hold the |
12141 lock on the other's **FILE**, deadlock ensues. This type of deadlock can be avoided by acquiring
12142 **FILE** locks in a consistent order. In particular, the line-buffered output stream deadlock can
12143 typically be avoided by acquiring locks on input streams before locks on output streams if a
12144 thread would be acquiring both.

12145 In summary, threads sharing *stdio* streams with other threads can use *flockfile*( ) and *funlockfile*( )
12146 to cause sequences of I/O performed by a single thread to be kept bundled. The only case where
12147 the use of *flockfile*( ) and *funlockfile*( ) is required is to provide a scope protecting uses of the
12148 **_unlocked*( ) functions/macros. This moves the cost/performance tradeoff to the optimal point.

12149 **FUTURE DIRECTIONS**

12150 None.

12151 **SEE ALSO**

12152 *getc_unlocked*( ), *putc_unlocked*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>** |

12153 **CHANGE HISTORY**

12154 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

12155 **Issue 6**

12156 These functions are marked as part of the Thread-Safe Functions option. |

12157 **NAME**

12158 floor, floorf, floorl — floor function

12159 **SYNOPSIS**

12160
```
#include <math.h>
```

12161
```
double floor(double x);
```
12162
```
float floorf(float x);
```
12163
```
long double floorl(long double x);
```

12164 **DESCRIPTION**

12165 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12166 conflict between the requirements described here and the ISO C standard is unintentional. This
12167 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

12168 These functions shall compute the largest integral value not greater than *x*.

12169 An application wishing to check for error situations should set *errno* to 0 before calling *floor*(). If
12170 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

12171 **RETURN VALUE**

12172 Upon successful completion, these functions shall return the largest integral value not greater
12173 than *x*, expressed as a **double**.

12174 XSI If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

12175 If the correct value would cause overflow, −HUGE_VAL shall be returned and *errno* set to
12176 [ERANGE].

12177 XSI If *x* is ±Inf or ±0, the value of *x* shall be returned.

12178 **ERRORS**

12179 These functions shall fail if:

12180 [ERANGE] The result would cause an overflow.

12181 These functions may fail if:

12182 XSI [EDOM] The value of *x* is NaN.

12183 XSI No other errors shall occur.

12184 **EXAMPLES**

12185 None.

12186 **APPLICATION USAGE**

12187 The integral value returned by *floor*() as a **double** might not be expressible as an **int** or **long**. The
12188 return value should be tested before assigning it to an integer type to avoid the undefined results
12189 of an integer overflow.

12190 The *floor*() function can only overflow when the floating point representation has
12191 DBL_MANT_DIG > DBL_MAX_EXP.

12192 **RATIONALE**

12193 None.

12194 **FUTURE DIRECTIONS**

12195 None.

12196 **SEE ALSO**

12197 *ceil*( ), *isnan*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

12198 **CHANGE HISTORY**

12199 First released in Issue 1. Derived from Issue 1 of the SVID.

12200 **Issue 4**

12201 References to *matherr*( ) are removed.

12202 The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the
12203 ISO C standard and to rationalize handling in the mathematics functions.

12204 The word **long** has been replaced with the words **long** in the APPLICATION USAGE section.

12205 The return value specified for [EDOM] is marked as an extension.

12206 **Issue 5**

12207 The DESCRIPTION is updated to indicate how an application should check for an error. This
12208 text was previously published in the APPLICATION USAGE section.

12209 **Issue 6**

12210 The *floorf*( ) and *floorl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

NAME

fma, fmaf, fmal — floating-point multiply-add

SYNOPSIS

```
#include <math.h>

double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double 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 $(x * y) + z$, rounded as one ternary operation: they shall compute the value (as if) to infinite precision and round once to the result format, according to the rounding mode characterized by the value of FLT_ROUNDS.

An application wishing to check for error situations should set *errno* to 0 before calling these functions. If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return $(x * y) + z$, rounded as one ternary operation.

If *x*, *y*, or *z* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

ERRORS

These functions may fail if:

[EDOM] The value of *x*, *y*, or *z* is NaN.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

In many cases, clever use of floating (*fused*) multiply-add leads to much improved code; but its unexpected use by the compiler can undermine carefully written code. The FP_CONTRACT macro can be used to disallow use of floating multiply-add; and the *fma*() function guarantees its use where desired. Many current machines provide hardware floating multiply-add instructions; software implementation can be used for others.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

12250 **NAME**

12251 fmax, fmaxf, fmaxl — determine maximum numeric value of two floating-point numbers

12252 **SYNOPSIS**

12253
```
#include <math.h>
```

12254
```
double fmax(double x, double y);
```
12255
```
float fmaxf(float x, float y);
```
12256
```
long double fmaxl(long double x, long double y);
```

12257 **DESCRIPTION**

12258 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12259 conflict between the requirements described here and the ISO C standard is unintentional. This
12260 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

12261 These functions shall determine the maximum numeric value of their arguments. NaN
12262 arguments shall be treated as missing data: if one argument is a NaN and the other numeric,
12263 then the *fmax*( ), *fmaxf*( ), and *fmaxl*( ) functions shall choose the numeric value.

12264 An application wishing to check for error situations should set *errno* to 0 before calling these
12265 functions. If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

12266 **RETURN VALUE**

12267 Upon successful completion, these functions shall return the maximum numeric value of their
12268 arguments.

12269 If *x* and *y* are NaN, NaN shall be returned and *errno* may be set to [EDOM].

12270 **ERRORS**

12271 These functions may fail if:

12272 [EDOM] The value of *x* and *y* is NaN.

12273 **EXAMPLES**

12274 None.

12275 **APPLICATION USAGE**

12276 None.

12277 **RATIONALE**

12278 None.

12279 **FUTURE DIRECTIONS**

12280 None.

12281 **SEE ALSO**

12282 *fdim*( ), *fmin*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

12283 **CHANGE HISTORY**

12284 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

fmin, fminf, fminl — determine minimum numeric value of two floating-point numbers

SYNOPSIS

```
#include <math.h>

double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double 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.

These functions shall determine the minimum 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.

An application wishing to check for error situations should set *errno* to 0 before calling these functions. If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the minimum numeric value of their arguments.

If *x* and *y* are NaN, NaN shall be returned and *errno* may be set to [EDOM].

ERRORS

These functions may fail if:

[EDOM] The value of *x* and *y* is NaN.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fdim( ), *fmax*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from ISO / IEC 9899: 1999 standard.

12320 **NAME**

12321 fmod, fmodf, fmodl — floating-point remainder value function

12322 **SYNOPSIS**

12323
```
#include <math.h>
```

12324
```
double fmod(double x, double y);
```
12325
```
float fmodf(float x, float y);
```
12326
```
long double fmodl(long double x, long double y);
```

12327 **DESCRIPTION**

12328 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12329 conflict between the requirements described here and the ISO C standard is unintentional. This
12330 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

12331 These functions shall return the floating-point remainder of the division of *x* by *y*.

12332 An application wishing to check for error situations should set *errno* to 0 before calling *fmod*( ). If
12333 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

12334 **RETURN VALUE**

12335 These functions shall return the value $x-i*y$, for some integer i such that, if y is non-zero, the
12336 result has the same sign as x and magnitude less than the magnitude of y.

12337 XSI If *x* or *y* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

12338 XSI If *y* is 0, NaN shall be returned and *errno* set to [EDOM], or 0 shall be returned and *errno* may be
12339 set to [EDOM].

12340 XSI If *x* is ±Inf, either 0 shall be returned and *errno* set to [EDOM], or NaN shall be returned and *errno*
12341 may be set to [EDOM].

12342 If *y* is non-zero, *fmod*(±0,*y*) shall return the value of *x*. If *x* is not ±Inf, *fmod*(*x*,±Inf) shall return
12343 the value of *x*.

12344 If the result underflows, 0 shall be returned and *errno* may be set to [ERANGE].

12345 **ERRORS**

12346 These functions may fail if:

12347 XSI [EDOM] One or both of the arguments is NaN, or *y* is 0, or *x* is ±Inf.

12348 [ERANGE] The result underflows

12349 XSI No other errors shall occur.

12350 **EXAMPLES**

12351 None.

12352 **APPLICATION USAGE**

12353 Portable applications should not call *fmod*( ) with *y* equal to 0, because the result is
12354 implementation-defined. The application should verify *y* is non-zero before calling *fmod*( ).

12355 **RATIONALE**

12356 None.

12357 **FUTURE DIRECTIONS**

12358 None.

12359 **SEE ALSO**

12360 *isnan*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

12361 **CHANGE HISTORY**

12362 First released in Issue 1. Derived from Issue 1 of the SVID.

12363 **Issue 4**

12364 References to *matherr*( ) are removed.

12365 The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the
12366 ISO C standard and to rationalize error handling in the mathematics functions.

12367 The return value specified for [EDOM] is marked as an extension.

12368 **Issue 5**

12369 The DESCRIPTION is updated to indicate how an application should check for an error. This
12370 text was previously published in the APPLICATION USAGE section.

12371 **Issue 6**

12372 The *fmodf*( ) and *fmodl*( ) functions are added for alignment with the ISO/IEC 9899:1999
12373 standard.

12374 **NAME**

12375 fmtmsg — display a message in the specified format on standard error and/or a system console

12376 **SYNOPSIS**

12377 XSI
```
#include <fmtmsg.h>

int fmtmsg(long classification, const char *label, int severity,
    const char *text, const char *action, const char *tag);
```
12380

12381 **DESCRIPTION**

12382 The *fmtmsg*( ) function can be used to display messages in a specified format instead of the
12383 traditional *printf*( ) function.

12384 Based on a message's classification component, *fmtmsg*( ) writes a formatted message either to
12385 standard error, to the console, or to both.

12386 A formatted message consists of up to five components as defined below. The component
12387 *classification* is not part of a message displayed to the user, but defines the source of the message
12388 and directs the display of the formatted message.

12389 *classification* Contains identifiers from the following groups of major classifications and
12390 subclassifications. Any one identifier from a subclass may be used in
12391 combination with a single identifier from a different subclass. Two or more
12392 identifiers from the same subclass should not be used together, with the
12393 exception of identifiers from the display subclass. (Both display subclass
12394 identifiers may be used so that messages can be displayed to both standard
12395 error and the system console).

12396 **Major Classifications**

12397 Identifies the source of the condition. Identifiers are: MM_HARD
12398 (hardware), MM_SOFT (software), and MM_FIRM (firmware).

12399 **Message Source Subclassifications**

12400 Identifies the type of software in which the problem is detected.
12401 Identifiers are: MM_APPL (application), MM_UTIL (utility), and
12402 MM_OPSYS (operating system).

12403 **Display Subclassifications**

12404 Indicates where the message is to be displayed. Identifiers are:
12405 MM_PRINT to display the message on the standard error stream,
12406 MM_CONSOLE to display the message on the system console. One or
12407 both identifiers may be used.

12408 **Status Subclassifications**

12409 Indicates whether the application can recover from the condition.
12410 Identifiers are: MM_RECOVER (recoverable) and MM_NRECOV (non-
12411 recoverable).

12412 An additional identifier, MM_NULLMC, indicates that no classification
12413 component is supplied for the message.

12414 *label* Identifies the source of the message. The format is two fields separated by a
12415 colon. The first field is up to 10 bytes, the second is up to 14 bytes.

12416 *severity* Indicates the seriousness of the condition. Identifiers for the levels of *severity*
12417 are:

MM_HALT Indicates that the application has encountered a severe fault and is halting. Produces the string "HALT".

MM_ERROR Indicates that the application has detected a fault. Produces the string "ERROR".

MM_WARNING Indicates a condition that is out of the ordinary, that might be a problem, and should be watched. Produces the string "WARNING".

MM_INFO Provides information about a condition that is not in error. Produces the string "INFO".

MM_NOSEV Indicates that no severity level is supplied for the message.

text 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.

action Describes the first step to be taken in the error-recovery process. The *fmtmsg*( ) function precedes the action string with the prefix: "TO FIX:". The *action* string is not limited to a specific size.

tag An identifier that references on-line documentation for the message. Suggested usage is that *tag* includes the *label* and a unique identifying number. A sample *tag* is "XSI:cat:146".

The *MSGVERB* environment variable (for message verbosity) tells *fmtmsg*( ) which message components it is to select when writing messages to standard error. The value of *MSGVERB* is 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*( ) includes 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 is not 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*( ) selects all components.

MSGVERB affects only which components are selected for display to standard error. All message components are included in console messages.

RETURN VALUE

The *fmtmsg*( ) function shall return one of the following values:

MM_OK The function succeeded.

MM_NOTOK The function failed completely.

MM_NOMSG The function was unable to generate a message on standard error, but otherwise succeeded.

MM_NOCON The function was unable to generate a console message, but otherwise succeeded.

ERRORS

None.

12459 **EXAMPLES**

12460 1. The following example of *fmtmsg*( ):

12461 ```
fmtmsg(MM_PRINT, "XSI:cat", MM_ERROR, "illegal option",
12462 "refer to cat in user's reference manual", "XSI:cat:001")
```

12463 produces a complete message in the specified message format:

12464 ```
XSI:cat: ERROR: illegal option
12465 TO FIX: refer to cat in user's reference manual XSI:cat:001
```

12466 2. When the environment variable *MSGVERB* is set as follows:

12467 ```
MSGVERB=severity:text:action
```

12468 and Example 1 is used, *fmtmsg*( ) produces:

12469 ```
ERROR: illegal option
12470 TO FIX: refer to cat in user's reference manual
```

12471 **APPLICATION USAGE**

12472 One or more message components may be systematically omitted from messages generated by
12473 an application by using the null value of the argument for that component.

12474 **RATIONALE**

12475 None.

12476 **FUTURE DIRECTIONS**

12477 None.

12478 **SEE ALSO**

12479 *printf*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<fmtmsg.h>** |

12480 **CHANGE HISTORY**

12481 First released in Issue 4, Version 2.

12482 **Issue 5**

12483 Moved from X/OPEN UNIX extension to BASE.

12484 **NAME**

12485 fnmatch — match a file name or a path name

12486 **SYNOPSIS**

12487 `#include <fnmatch.h>`

12488 `int fnmatch(const char *`*`pattern`*`, const char *`*`string`*`, int `*`flags`*`);`

12489 **DESCRIPTION**

12490 The *fnmatch*( ) function shall match patterns as described in the Shell and Utilities volume of
12491 IEEE Std. 1003.1-200x, Section 2.14.1, Patterns Matching a Single Character, and Section 2.14.2,
12492 Patterns Matching Multiple Characters. It checks the string specified by the *string* argument to
12493 see if it matches the pattern specified by the *pattern* argument.

12494 The *flags* argument modifies the interpretation of *pattern* and *string*. It is the bitwise-inclusive OR
12495 of zero or more of the flags defined in **<fnmatch.h>**. If the FNM_PATHNAME flag is set in *flags*,
12496 then a slash character (`'/'`) in *string* shall be explicitly matched by a slash in *pattern*; it shall not
12497 be matched by either the asterisk or question-mark special characters, nor by a bracket
12498 expression. If the FNM_PATHNAME flag is not set, the slash character is treated as an ordinary
12499 character.

12500 If FNM_NOESCAPE is not set in *flags*, a backslash character (`'\'`) in *pattern* followed by any
12501 other character shall match that second character in *string*. In particular, `"\\"` shall match a
12502 backslash in *string*. If FNM_NOESCAPE is set, a backslash character shall be treated as an
12503 ordinary character.

12504 If FNM_PERIOD is set in *flags*, then a leading period (`'.'`) in *string* shall match a period in
12505 *pattern*; as described by rule 2 in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section
12506 2.14.3, Patterns Used for File Name Expansion where the location of ‘‘leading’’ is indicated by
12507 the value of FNM_PATHNAME:

12508 • If FNM_PATHNAME is set, a period is ‘‘leading’’ if it is the first character in *string* or if it
12509 immediately follows a slash.

12510 • If FNM_PATHNAME is not set, a period is ‘‘leading’’ only if it is the first character of *string*.

12511 If FNM_PERIOD is not set, then no special restrictions are placed on matching a period.

12512 **RETURN VALUE**

12513 If *string* matches the pattern specified by *pattern*, then *fnmatch*( ) shall return 0. If there is no
12514 match, *fnmatch*( ) shall return FNM_NOMATCH, which is defined in **<fnmatch.h>**. If an error
12515 occurs, *fnmatch*( ) shall return another non-zero value.

12516 **ERRORS**

12517 No errors are defined.

12518 **EXAMPLES**

12519 None.

12520 **APPLICATION USAGE**

12521 The *fnmatch*( ) function has two major uses. It could be used by an application or utility that
12522 needs to read a directory and apply a pattern against each entry. The *find* utility is an example of
12523 this. It can also be used by the *pax* utility to process its *pattern* operands, or by applications that
12524 need to match strings in a similar manner.

12525 The name *fnmatch*( ) is intended to imply *file name* match, rather than *path name* match. The
12526 default action of this function is to match file names, rather than path names, since it gives no
12527 special significance to the slash character. With the FNM_PATHNAME flag, *fnmatch*( ) does
12528 match path names, but without tilde expansion, parameter expansion, or special treatment for a

period at the beginning of a file name.

RATIONALE

This function replaced the REG_FILENAME flag of *regcomp*( ) in early proposals of this volume of IEEE Std. 1003.1-200x. It provides virtually the same functionality as the *regcomp*( ) and *regexec*( ) functions using the REG_FILENAME and REG_FSLASH flags (the REG_FSLASH flag was proposed for *regcomp*( ), and would have had the opposite effect from FNM_PATHNAME), but with a simpler function and less system overhead.

FUTURE DIRECTIONS

None.

SEE ALSO

glob( ), *wordexp*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<fnmatch.h>**, the Shell and Utilities volume of IEEE Std. 1003.1-200x

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 5

Moved from POSIX2 C-language Binding to BASE.

12545 **NAME**

12546 fopen — open a stream

12547 **SYNOPSIS**

12548 `#include <stdio.h>`

12549 `FILE *fopen(const char *restrict filename, const char *restrict mode);`

12550 **DESCRIPTION**

12551 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12552 conflict between the requirements described here and the ISO C standard is unintentional. This
12553 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

12554 The *fopen*( ) function shall open the file whose path name is the string pointed to by *filename*, and
12555 associates a stream with it.

12556 The argument *mode* points to a string. If the string is one of the following, the file is open in the
12557 indicated mode. Otherwise, the behavior is undefined.

12558 *r* or *rb*	Open file for reading.
12559 *w* or *wb*	Truncate to zero length or create file for writing.
12560 *a* or *ab*	Append; open or create file for writing at end-of-file.
12561 *r+* or *rb+* or *r+b*	Open file for update (reading and writing).
12562 *w+* or *wb+* or *w+b*	Truncate to zero length or create file for update.
12563 *a+* or *ab+* or *a+b*	Append; open or create file for update, writing at end-of-file.

12564 CX The character `'b'` has no effect, but is allowed for ISO C standard conformance. Opening a file
12565 with read mode (*r* as the first character in the *mode* argument) shall fail if the file does not exist or
12566 cannot be read.

12567 Opening a file with append mode (*a* as the first character in the *mode* argument) shall cause all
12568 subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening
12569 calls to *fseek*( ).

12570 When a file is opened with update mode (`'+'` as the second or third character in the *mode*
12571 argument), both input and output may be performed on the associated stream. However, the
12572 application shall ensure that output is not directly followed by input without an intervening call
12573 to *fflush*( ) or to a file positioning function (*fseek*( ), *fsetpos*( ), or *rewind*( )), and input is not directly
12574 followed by output without an intervening call to a file positioning function, unless the input
12575 operation encounters end-of-file.

12576 When opened, a stream is fully buffered if and only if it can be determined not to refer to an
12577 interactive device. The error and end-of-file indicators for the stream shall be cleared.

12578 CX If *mode* is *w*, *a*, *w+*, or *a+*, and the file did not previously exist, upon successful completion,
12579 *fopen*( ) function shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file and
12580 the *st_ctime* and *st_mtime* fields of the parent directory.

12581 If *mode* is *w* or *w+* and the file did previously exist, upon successful completion, *fopen*( ) shall
12582 mark for update the *st_ctime* and *st_mtime* fields of the file. The *fopen*( ) function shall allocate a
12583 file descriptor as *open*( ) does.

12584 XSI After a successful call to the *fopen*( ) function, the orientation of the stream shall be cleared, the
12585 encoding rule shall be cleared, and the associated **mbstate_t** object shall be set to describe an
12586 initial conversion state.

The largest value that can be represented correctly in an object of type **off_t** shall be established as the offset maximum in the open file description.

RETURN VALUE

Upon successful completion, *fopen*( ) shall return a pointer to the object controlling the stream. Otherwise, a null pointer shall be returned, and *errno* shall be set to indicate the error.

ERRORS

The *fopen*( ) function shall fail if:

[EACCES] Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *mode* are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.

[EINTR] A signal was caught during *fopen*( ).

[EISDIR] The named file is a directory and *mode* requires write access.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

[EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

[ENAMETOOLONG]
The length of the *filename* argument exceeds {PATH_MAX} or a path name component is longer than {NAME_MAX}.

[ENFILE] The maximum allowable number of files is currently open in the system.

[ENOENT] A component of *filename* does not name an existing file or *filename* is an empty string.

[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.

[ENOTDIR] A component of the path prefix is not a directory.

[ENXIO] The named file is a character special or block special file, and the device associated with this special file does not exist.

[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**.

[EROFS] The named file resides on a read-only file system and *mode* requires write access.

The *fopen*( ) function may fail if:

[EINVAL] The value of the *mode* argument is not valid.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

[EMFILE] {STREAM_MAX} streams are currently open in the calling process.

[ENAMETOOLONG]
Path name resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

12627 CX [ENOMEM] Insufficient storage space is available.

12628 CX [ETXTBSY] The file is a pure procedure (shared text) file that is being executed and *mode*
12629 requires write access.

12630 **EXAMPLES**

12631 **Opening a File**

12632 The following example tries to open the file named **file** for reading. The *fopen*( ) function returns
12633 a file pointer that is used in subsequent *fgets*( ) and *fclose*( ) calls. If the program cannot open the
12634 file, it just ignores it.

```
12635 #include <stdio.h>
12636 ...
12637 FILE *fp;
12638 ...
12639 void rgrep(const char *file)
12640 {
12641 ...
12642     if ((fp = fopen(file, "r")) == NULL)
12643         return;
12644 ...
12645 }
```

12646 **APPLICATION USAGE**

12647 None.

12648 **RATIONALE**

12649 None.

12650 **FUTURE DIRECTIONS**

12651 None.

12652 **SEE ALSO**

12653 *fclose*( ), *fdopen*( ), *freopen*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**

12654 **CHANGE HISTORY**

12655 First released in Issue 1. Derived from Issue 1 of the SVID.

12656 **Issue 4**

12657 In the DESCRIPTION, the descriptions of input and output operations on update streams are
12658 changed to be requirements on the application.

12659 The [EMFILE] error is added to the ERRORS section, and all the optional errors are marked as
12660 extensions.

12661 The following changes are incorporated for alignment with the ISO C standard:

12662 - The type of arguments *filename* and *mode* are changed from **char*** to **const char***.

12663 - In the DESCRIPTION, the use and settings of the *mode* argument are changed to support
12664 binary streams, and *setpos*( ) is added to the list of file positioning functions.

12665 The following change is incorporated for alignment with the FIPS requirements:

12666 - In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path
12667 name component is larger that {NAME_MAX} is now defined as mandatory and marked as
12668 an extension.

Issue 4, Version 2

The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- It states that [ELOOP] is returned if too many symbolic links are encountered during path name resolution.
- A second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of path name resolution of a symbolic link.

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

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 DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

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

- The prototype for *fopen*( ) is updated.
- The DESCRIPTION is updated to note that if the argument *mode* points to a string other than those listed, then the behavior is undefined.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

12698 **NAME**

12699 fork — create a new process

12700 **SYNOPSIS**

12701
```
#include <unistd.h>
```

12702
```
pid_t fork(void);
```

12703 **DESCRIPTION**

12704 The *fork*( ) function creates a new process. The new process (child process) shall be an exact copy
12705 of the calling process (parent process) except as detailed below:

12706 • The child process has a unique process ID.

12707 • The child process ID also does not match any active process group ID.

12708 • The child process has a different parent process ID (that is, the process ID of the parent
12709 process).

12710 • The child process has its own copy of the parent's file descriptors. Each of the child's file
12711 descriptors refers to the same open file description with the corresponding file descriptor of
12712 the parent.

12713 • The child process has its own copy of the parent's open directory streams. Each open
12714 directory stream in the child process may share directory stream positioning with the
12715 corresponding directory stream of the parent.

12716 XSI • The child process may have its own copy of the parent's message catalog descriptors.

12717 • The child process' values of *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* are set to 0.

12718 • The time left until an alarm clock signal is reset to zero, and the alarm, if any, is canceled; see
12719 *alarm*( ).

12720 XSI • All *semadj* values are cleared.

12721 • File locks set by the parent process are not inherited by the child process.

12722 • The set of signals pending for the child process is initialized to the empty set.

12723 XSI • Interval timers are reset in the child process.

12724 SEM • Any semaphores that are open in the parent process shall also be open in the child process.

12725 ML • The child process does not inherit any address space memory locks established by the parent
12726 process via calls to *mlockall*( ) or *mlock*( ).

12727 MF|SHM • Memory mappings created in the parent are retained in the child process. MAP_PRIVATE
12728 mappings inherited from the parent shall also be MAP_PRIVATE mappings in the child, and
12729 any modifications to the data in these mappings made by the parent prior to calling *fork*( )
12730 shall be visible to the child. Any modifications to the data in MAP_PRIVATE mappings made
12731 by the parent after *fork*( ) returns shall be visible only to the parent. Modifications to the data
12732 in MAP_PRIVATE mappings made by the child shall be visible only to the child.

12733 PS • For the SCHED_FIFO and SCHED_RR scheduling policies, the child process shall inherit the
12734 policy and priority settings of the parent process during a *fork*( ) function. For other
12735 scheduling policies, the policy and priority settings on *fork*( ) are implementation-defined.

12736 TMR • Per-process timers created by the parent are not inherited by the child process.

12737 MSG • The child process has its own copy of the message queue descriptors of the parent. Each of
12738 the message descriptors of the child refers to the same open message queue description as
12739 the corresponding message descriptor of the parent.

12740 AIO
- No asynchronous input or asynchronous output operations are inherited by the child process.

12742
- A process is created with a single thread. If a multi-threaded process calls *fork*( ), the new process contains 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.

12748 TRC TRI
- If the Trace option and the Trace Inherit option are both supported:

  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.

12756 TRC
- If the Trace option is supported, but the Trace Inherit option is not supported:

  The child process shall not be traced into any of the trace streams of its parent process.

12758
- If the Trace option is supported, the child process of a trace controller process shall not control the trace streams controlled by its parent process.

12760 CPT The initial value of the CPU-time clock of the child process shall be set to zero.

12761 TCT The initial value of the CPU-time clock of the single thread of the child process shall be set to zero.

12763 ***Notes to Reviewers***

12764 *This section with side shading will not appear in the final copy. - Ed.*

12765 Check this text is correct after new addenda are rolled in. (Ref D1, XSH, ERN 126)

12766 For process characteristics not defined by this volume of IEEE Std. 1003.1-200x, their inheritance
12767 is not defined by this volume of IEEE Std. 1003.1-200x. Process characteristics defined by this
12768 volume of IEEE Std. 1003.1-200x have their inheritance explicitly defined.

12769 After *fork*( ), both the parent and the child processes are capable of executing independently
12770 before either one terminates.

12771 **RETURN VALUE**

12772 Upon successful completion, *fork*( ) shall return 0 to the child process and shall return the
12773 process ID of the child process to the parent process. Both processes shall continue to execute
12774 from the *fork*( ) function. Otherwise, –1 shall be returned to the parent process, no child process
12775 shall be created, and *errno* shall be set to indicate the error.

12776 **ERRORS**

12777 The *fork*( ) function shall fail if:

12778 [EAGAIN] The system lacked the necessary resources to create another process, or the
12779 system-imposed limit on the total number of processes under execution
12780 system-wide or by a single user {CHILD_MAX} would be exceeded.

12781 The *fork*( ) function may fail if:

12782 [ENOMEM] Insufficient storage space is available. |

12783 **EXAMPLES**

12784 None.

12785 **APPLICATION USAGE**

12786 None.

12787 **RATIONALE**

12788 Many historical implementations have timing windows where a signal sent to a process group
12789 (for example, an interactive SIGINT) just prior to or during execution of *fork*( ) is delivered to the
12790 parent following the *fork*( ) but not to the child because the *fork*( ) code clears the child's set of
12791 pending signals. This volume of IEEE Std. 1003.1-200x does not require, or even permit, this
12792 behavior. However, it is pragmatic to expect that problems of this nature may continue to exist
12793 in implementations that appear to conform to this volume of IEEE Std. 1003.1-200x and pass
12794 available verification suites. This behavior is only a consequence of the implementation failing to
12795 make the interval between signal generation and delivery totally invisible. From the
12796 application's perspective, a *fork*( ) call should appear atomic. A signal that is generated prior to
12797 the *fork*( ) should be delivered prior to the *fork*( ). A signal sent to the process group after the
12798 *fork*( ) should be delivered to both parent and child. The implementation may actually initialize
12799 internal data structures corresponding to the child's set of pending signals to include signals
12800 sent to the process group during the *fork*( ). Since the *fork*( ) call can be considered as atomic
12801 from the application's perspective, the set would be initialized as empty and such signals would
12802 have arrived after the *fork*( ); see also **<signal.h>**.

12803 One approach that has been suggested to address the problem of signal inheritance across *fork*( )
12804 is to add an [EINTR] error, which would be returned when a signal is detected during the call. |
12805 While this is preferable to losing signals, it was not considered an optimal solution. Although it
12806 is not recommended for this purpose, such an error would be an allowable extension for an
12807 implementation.

12808 The [ENOMEM] error value is reserved for those implementations that detect and distinguish |
12809 such a condition. This condition occurs when an implementation detects that there is not enough
12810 memory to create the process. This is intended to be returned when [EAGAIN] is inappropriate |
12811 because there can never be enough memory (either primary or secondary storage) to perform the
12812 operation. Because *fork*( ) duplicates an existing process, this must be a condition where there is
12813 sufficient memory for one such process, but not for two. Many historical implementations
12814 actually return [ENOMEM] due to temporary lack of memory, a case that is not generally |
12815 distinct from [EAGAIN] from the perspective of a portable application.

12816 Part of the reason for including the optional error [ENOMEM] is because the SVID specifies it
12817 and it should be reserved for the error condition specified there. The condition is not applicable
12818 on many implementations.

12819 IEEE Std. 1003.1-1988 neglected to require concurrent execution of the parent and child of *fork*( ). |
12820 A system that single-threads processes was clearly not intended and is considered an
12821 unacceptable ''toy implementation'' of this volume of IEEE Std. 1003.1-200x. The only objection
12822 anticipated to the phrase ''executing independently'' is testability, but this assertion should be
12823 testable. Such tests require that both the parent and child can block on a detectable action of the
12824 other, such as a write to a pipe or a signal. An interactive exchange of such actions should be
12825 possible for the system to conform to the intent of this volume of IEEE Std. 1003.1-200x.

12826 The [EAGAIN] error exists to warn applications that such a condition might occur. Whether it
12827 occurs or not is not in any practical sense under the control of the application because the
12828 condition is usually a consequence of the user's use of the system, not of the application's code.
12829 Thus, no application can or should rely upon its occurrence under any circumstances, nor

should the exact semantics of what concept of ''user'' is used be of concern to the application writer. Validation writers should be cognizant of this limitation.

There are two reasons why POSIX programmers call *fork*(). One reason is to create a new thread of control within the same program (which was originally only possible in POSIX by creating a new process); the other is to create a new process running a different program. In the latter case, the call to *fork*() is soon followed by a call to one of the *exec* functions.

The general problem with making *fork*() work in a multi-threaded world is what to do with all of the threads. There are two alternatives. One is to copy all of the threads into the new process. This causes the programmer or implementation to deal with threads that are suspended on system calls or that might be about to execute system calls that should not be executed in the new process. The other alternative is to copy only the thread that calls *fork*(). This creates the difficulty that the state of process-local resources is usually held in process memory. If a thread that is not calling *fork*() holds a resource, that resource is never released in the child process because the thread whose job it is to release the resource does not exist in the child process.

When a programmer is writing a multi-threaded program, the first described use of *fork*(), creating new threads in the same program, is provided by the *pthread_create*() function. The *fork*() function is thus used only to run new programs, and the effects of calling functions that require certain resources between the call to *fork*() and the call to an *exec* function are undefined.

The addition of the *forkall*() function to the standard was considered and rejected. The *forkall*() function lets all the threads in the parent be duplicated in the child. This essentially duplicates the state of the parent in the child. This allows threads in the child to continue processing and allows locks and the state to be preserved without explicit *pthread_atfork*() code. The calling process has to ensure that the threads processing state that is shared between the parent and child (that is, file descriptors or MAP_SHARED memory) behaves properly after *forkall*(). For example, if a thread is reading a file descriptor in the parent when *forkall*() is called, then two threads (one in the parent and one in the child) are reading the file descriptor after the *forkall*(). If this is not desired behavior, the parent process has to synchronize with such threads before calling *forkall*().

When *forkall*() is called, threads, other than the calling thread, that are in POSIX System Interfaces functions that can return with an [EINTR] error may have those functions return [EINTR] if the implementation cannot ensure that the function behaves correctly in the parent and child. In particular, *pthread_cond_wait*() and *pthread_cond_timedwait*() need to return in order to ensure that the condition has not changed. These functions can be awakened by a spurious condition wakeup rather than returning [EINTR].

FUTURE DIRECTIONS

None.

SEE ALSO

alarm(), *exec*, *fcntl*(), *posix_trace_attr_getinherited*(), *posix_trace_trid_eventid_open*(), *semop*(), *signal*(), *times*(), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

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

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The argument list is explicitly defined as **void**.
- Though functionally identical to Issue 3, the DESCRIPTION has been reorganized to improve clarity and to align more closely with the ISO POSIX-1 standard.
- The description of the [EAGAIN] error is updated to indicate that this error can also be returned if a system lacks the resources to create another process.

Issue 4, Version 2

The DESCRIPTION is changed for X/OPEN UNIX conformance to identify that interval timers are reset in the child process.

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.

12898 **NAME**

12899 fpathconf, pathconf — get configurable path name variables

12900 **SYNOPSIS**

12901 `#include <unistd.h>`

12902 `long fpathconf(int fildes, int name);` |

12903 `long pathconf(const char *path, int name);` |

12904 **DESCRIPTION** |

12905 The *fpathconf*( ) and *pathconf*( ) functions provide a method for the application to determine the
12906 current value of a configurable limit or option (*variable*) that is associated with a file or directory.

12907 For *pathconf*( ), the *path* argument points to the path name of a file or directory.

12908 For *fpathconf*( ), the *fildes* argument is an open file descriptor.

12909 The *name* argument represents the variable to be queried relative to that file or directory.
12910 Implementations shall support all of the variables listed in the following table and may support
12911 others. The variables in the following table come from **<limits.h>** or **<unistd.h>** and the
12912 symbolic constants, defined in **<unistd.h>**, are the corresponding values used for *name*. Support
12913 for some path name configuration variables is dependent on implementation options (see
12914 shading and margin codes in the table below). Where an implementation option is not
12915 supported, the variable need not be supported.

12916

12917

	Variable	Value of *name*	Notes
12918	{FILESIZEBITS}	_PC_FILESIZEBITS	3, 4
12919	{LINK_MAX}	_PC_LINK_MAX	1
12920	{MAX_CANON}	_PC_MAX_CANON	2
12921	{MAX_INPUT}	_PC_MAX_INPUT	2
12922	{NAME_MAX}	_PC_NAME_MAX	3, 4
12923	{PATH_MAX}	_PC_PATH_MAX	4, 5
12924	{PIPE_BUF}	_PC_PIPE_BUF	6
12925 ADV	{POSIX_ALLOC_SIZE_MIN}	_PC_ALLOC_SIZE_MIN	
12926 ADV	{POSIX_REC_INCR_XFER_SIZE}	_PC_REC_INCR_XFER_SIZE	
12927 ADV	{POSIX_REC_MAX_XFER_SIZE}	_PC_REC_MAX_XFER_SIZE	
12928 ADV	{POSIX_REC_MIN_XFER_SIZE}	_PC_REC_MIN_XFER_SIZE	
12929 ADV	{POSIX_REC_XFER_ALIGN}	_PC_REC_XFER_ALIGN	
12930	{SYMLINK_MAX}	_PC_SYMLINK_MAX	4,9
12931	_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED	7
12932	_POSIX_NO_TRUNC	_PC_NO_TRUNC	3, 4
12933	_POSIX_VDISABLE	_PC_VDISABLE	2
12934	_POSIX_ASYNC_IO	_PC_ASYNC_IO	8
12935	_POSIX_PRIO_IO	_PC_PRIO_IO	8
12936	_POSIX_SYNC_IO	_PC_SYNC_IO	8

12937 **Notes:**

12938 1. If *path* or *fildes* refers to a directory, the value returned applies to the directory
12939 itself.

12940 2. If *path* or *fildes* does not refer to a terminal file, it is unspecified whether an
12941 implementation supports an association of the variable name with the specified
12942 file.

3. If *path* or *fildes* refers to a directory, the value returned applies to file names within the directory.
4. If *path* or *fildes* does not refer to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.
5. If *path* or *fildes* refers to a directory, the value returned is the maximum length of a relative path name when the specified directory is the working directory.
6. If *path* refers to a FIFO, or *fildes* refers to a pipe or FIFO, the value returned applies to the referenced object. If *path* or *fildes* refers to a directory, the value returned applies to any FIFO that exists or can be created within the directory. If *path* or *fildes* refers to any other type of file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
7. If *path* or *fildes* refers to a directory, the value returned applies to any files, other than directories, that exist or can be created within the directory.
8. If *path* or *fildes* refers to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.
9. If *path* or *fildes* refers to a directory, the value returned is the maximum length of the string that a symbolic link in that directory can contain.

RETURN VALUE

If *name* is an invalid value, both *pathconf*( ) and *fpathconf*( ) shall return –1 and set *errno* to indicate the error.

If the variable corresponding to *name* has no limit for the *path* or file descriptor, both *pathconf*( ) and *fpathconf*( ) shall return –1 without changing *errno*. If the implementation needs to use *path* to determine the value of *name* and the implementation does not support the association of *name* with the file specified by *path*, or if the process did not have appropriate privileges to query the file specified by *path*, or *path* does not exist, *pathconf*( ) shall return –1 and set *errno* to indicate the error.

If the implementation needs to use *fildes* to determine the value of *name* and the implementation does not support the association of *name* with the file specified by *fildes*, or if *fildes* is an invalid file descriptor, *fpathconf*( ) shall return –1 and set *errno* to indicate the error.

Otherwise, *pathconf*( ) or *fpathconf*( ) shall return the current variable value for the file or directory without changing *errno*. The value returned shall not be more restrictive than the corresponding value available to the application when it was compiled with the implementation's **<limits.h>** or **<unistd.h>**.

ERRORS

The *pathconf*( ) function shall fail if:

[EINVAL] The value of *name* is not valid.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

The *pathconf*( ) function may fail if:

[EACCES] Search permission is denied for a component of the path prefix.

[EINVAL] The implementation does not support an association of the variable *name* with the specified file.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG] |
The length of the *path* argument exceeds {PATH_MAX} or a path name component is longer than {NAME_MAX}. |

[ENAMETOOLONG] |
As a result of encounering a symbolic link in resolution of the *path* argument, the length of the substituted path name string exceeded {PATH_MAX}. |

[ENOENT] A component of *path* does not name an existing file or *path* is an empty string. |

[ENOTDIR] A component of the path prefix is not a directory. |

The *fpathconf*( ) function shall fail if:

[EINVAL] The value of *name* is not valid. |

The *fpathconf*( ) function may fail if:

[EBADF] The *fildes* argument is not a valid file descriptor. |

[EINVAL] The implementation does not support an association of the variable *name* with the specified file. |

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *pathconf*( ) function was proposed immediately after the *sysconf*( ) function when it was realized that some configurable values may differ across file system, directory, or device boundaries.

For example, {NAME_MAX} frequently changes between System V and BSD-based file systems; System V uses a maximum of 14, BSD 255. On an implementation that provides both types of file systems, an application would be forced to limit all path name components to 14 bytes, as this would be the value specified in **<limits.h>** on such a system.

Therefore, various useful values can be queried on any path name or file descriptor, assuming that the appropriate permissions are in place.

The value returned for the variable {PATH_MAX} indicates the longest relative path name that could be given if the specified directory is the process' current working directory. A process may not always be able to generate a name that long and use it if a subdirectory in the path name crosses into a more restrictive file system.

The value returned for the variable _POSIX_CHOWN_RESTRICTED also applies to directories that do not have file systems mounted on them. The value may change when crossing a mount point, so applications that need to know should check for each directory. (An even easier check is to try the *chown*( ) function and look for an error in case it happens.)

Unlike the values returned by *sysconf*( ), the path name-oriented variables are potentially more volatile and are not guaranteed to remain constant throughout the process' lifetime. For example, in between two calls to *pathconf*( ), the file system in question may have been unmounted and remounted with different characteristics.

Also note that most of the errors are optional. If one of the variables always has the same value on an implementation, the implementation need not look at *path* or *fildes* to return that value and is, therefore, not required to detect any of the errors except the meaning of [EINVAL] that indicates that the value of *name* is not valid for that variable.

If the value of any of the limits are indeterminate (logically infinite), they are defined in **<limits.h>** and the *pathconf*( ) and *fpathconf*( ) functions return –1 without changing *errno*. This can be distinguished from the case of giving an unrecognized *name* argument because *errno* is set to [EINVAL] in this case.

Since –1 is a valid return value for the *pathconf*( ) and *fpathconf*( ) functions, applications should set *errno* to zero before calling them and check *errno* only if the return value is –1.

For the case of {SYMLINK_MAX}, since both *pathconf*( ) and *open*( ) follow symbolic links, there is no way that *path* or *fildes* could refer to a symbolic link.

FUTURE DIRECTIONS

None.

SEE ALSO

confstr( ), *sysconf*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<limits.h>**, **<unistd.h>**, the Shell and Utilities volume of IEEE Std. 1003.1-200x

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The *fpathconf*( ) function now has the full **long** return type in the SYNOPSIS section.

The following changes gave been made for alignment with the ISO POSIX-1 standard:

- The type of argument *path* is changed from **char*** to **const char***. Also, the return value of both functions is changed from **long** to **long**.
- In the DESCRIPTION, the words ‘‘The behavior is undefined if’’ have been replaced by ‘‘it is unspecified whether an implementation supports an association of the variable name with the specified file’’ in notes 2, 4, and 6.
- In the RETURN VALUE section, errors associated with the use of *path* and *fildes*, when an implementation does not support the requested association, are now specified separately.
- The requirement that *errno* be set to indicate the error is added.

The following change is incorporated for alignment with the FIPS requirements:

- In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path name component is larger that {NAME_MAX} is now defined as mandatory and marked as an extension.

Issue 4, Version 2

The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- It states that [ELOOP] is returned if too many symbolic links are encountered during path name resolution.
- A second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of path name resolution of a symbolic link.

Issue 5

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

Large File Summit extensions are added.

Issue 6

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to include {FILESIZEBITS}.
- 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 _PC_SYMLINK_MAX entry is added to the table in the DESCRIPTION.

The *pathconf*( ) variables {POSIX_ALLOC_SIZE_MIN}, {POSIX_REC_INCR_XFER_SIZE}, {POSIX_REC_MAX_XFER_SIZE}, {POSIX_REC_MIN_XFER_SIZE}, {POSIX_REC_XFER_ALIGN} and their associated names are added for alignment with IEEE Std. 1003.1d-1999.

13088 **NAME**

13089 fpclassify — classify real floating type

13090 **SYNOPSIS**

13091 `#include <math.h>`

13092 `int fpclassify(real-floating x);`

13093 **DESCRIPTION**

13094 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13095 conflict between the requirements described here and the ISO C standard is unintentional. This
13096 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

13097 The *fpclassify*( ) macro shall classify its argument value as NaN, infinite, normal, subnormal,
13098 zero, or into another implementation-defined category. First, an argument represented in a
13099 format wider than its semantic type is converted to its semantic type. Then classification is based
13100 on the type of the argument.

13101 **RETURN VALUE**

13102 The *fpclassify*( ) macro shall return the value of the number classification macro appropriate to
13103 the value of its argument.

13104 **ERRORS**

13105 No errors are defined.

13106 **EXAMPLES**

13107 None.

13108 **APPLICATION USAGE**

13109 None.

13110 **RATIONALE**

13111 None.

13112 **FUTURE DIRECTIONS**

13113 None.

13114 **SEE ALSO**

13115 *isfinite*( ), *isinf*( ), *isnan*( ), *isnormal*( ), *signbit*( ), the Base Definitions volume of
13116 IEEE Std. 1003.1-200x, **<math.h>**

13117 **CHANGE HISTORY**

13118 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

13119 **NAME**

13120 fprintf, printf, snprintf, sprintf — print formatted output

13121 **SYNOPSIS**

13122 `#include <stdio.h>`

13123 `int fprintf(FILE *restrict stream, const char *restrict format, ...);`
13124 `int printf(const char *restrict format, ...);`
13125 `int snprintf(char *restrict s, size_t n, const char *restrict format, ...);`
13126 `int sprintf(char *restrict s, const char *restrict format, ...);`

13127 **DESCRIPTION**

13128 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13129 conflict between the requirements described here and the ISO C standard is unintentional. This
13130 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

13131 The *fprintf*( ) function places output on the named output *stream*. The *printf*( ) function places
13132 output on the standard output stream *stdout*. The *sprintf*( ) function places output followed by
13133 the null byte, `'\0'`, in consecutive bytes starting at **s*; it is the user's responsibility to ensure that
13134 enough space is available.

13135 XSI The *snprintf*( ) function is identical to *sprintf*( ) with the addition of the *n* argument, which states
13136 the size of the buffer referred to by *s*. If *n* is greater than zero, but not large enough to hold all
13137 output bytes specified by the format, output bytes beyond the *n*-1st are discarded rather than
13138 being written into the array.

13139 If copying takes place between objects that overlap as a result of a call to *sprintf*( ) or *snprintf*( ),
13140 the results are undefined.

13141 Each of these functions converts, formats, and prints its arguments under control of the *format*.
13142 The *format* is a character string, beginning and ending in its initial shift state, if any. The *format* is
13143 composed of zero or more directives: *ordinary characters*, which are simply copied to the output
13144 stream, and *conversion specifications*, each of which results in the fetching of zero or more
13145 arguments. The results are undefined if there are insufficient arguments for the *format*. If the
13146 *format* is exhausted while arguments remain, the excess arguments are evaluated but are
13147 otherwise ignored.

13148 XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
13149 to the next unused argument. In this case, the conversion character `'%'` (see below) is replaced
13150 by the sequence `"%n$"`, where *n* is a decimal integer in the range [1,{NL_ARGMAX}], giving the
13151 position of the argument in the argument list. This feature provides for the definition of format
13152 strings that select arguments in an order appropriate to specific languages (see the EXAMPLES
13153 section).

13154 In format strings containing the `"%n$"` form of conversion specifications, numbered arguments
13155 in the argument list can be referenced from the format string as many times as required.

13156 In format strings containing the `'%'` form of conversion specifications, each argument in the
13157 argument list is used exactly once.

13158 All forms of the *fprintf*( ) functions allow for the insertion of a language-dependent radix
13159 character in the output string. The radix character is defined in the program's locale (category
13160 *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the
13161 radix character defaults to a period (`'.'`).

13162 XSI Each conversion specification is introduced by the `'%'` character or by the character sequence
13163 `"%n$"`, after which the following appear in sequence:

- Zero or more *flags* (in any order), which modify the meaning of the conversion specification.
- An optional minimum *field width*. If the converted value has fewer bytes than the field width, it shall be padded with spaces by default on the left; it shall be padded on the right, if the left-adjustment flag ('−'), described below, is given to the field width. The field width takes the form of an asterisk ('*'), described below, or a decimal integer.
- An optional *precision* that gives the minimum number of digits to appear for the *d*, *i*, *o*, *u*, *x*, and *X* conversions; the number of digits to appear after the radix character for the *e*, *E*, and *f* conversions; the maximum number of significant digits for the *g* and *G* conversions; or the maximum number of bytes to be printed from a string in *s* and *S* conversions. The precision takes the form of a period ('.') followed either by an asterisk ('*'), described below, or an optional decimal digit string, where a null digit string is treated as 0. If a precision appears with any other conversion character, the behavior is undefined.
- An optional length modifier that specifies the size of the argument.
- A *conversion character* that indicates the type of conversion to be applied.

A field width, or precision, or both, may be indicated by an asterisk ('*'). In this case an argument of type **int** supplies the field width or precision. Applications shall ensure that arguments specifying field width, or precision, or both appear in that order before the argument, if any, to be converted. A negative field width is taken as a '−' flag followed by a positive field width. A negative precision is taken as if the precision were omitted. In format strings containing the "%n$" form of a conversion specification, a field width or precision may be indicated by the sequence "*m$", where *m* is a decimal integer in the range [1,{NL_ARGMAX}] giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision, for example:

```
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

The *format* can contain either numbered argument specifications (that is, "%n$" and "*m$"), or unnumbered argument specifications (that is, '%' and '*'), but normally not both. The only exception to this is that "%%" can be mixed with the "%n$" form. The results of mixing numbered and unnumbered argument specifications in a *format* string are undefined. When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the (*N*–*1*)th, are specified in the format string.

The flag characters and their meanings are:

' The integer portion of the result of a decimal conversion (%*i*, %*d*, %*u*, %*f*, %*g*, or %*G*) shall be formatted with thousands' grouping characters. For other conversions the behavior is undefined. The non-monetary grouping character is used.

– The result of the conversion shall be left-justified within the field. The conversion is right-justified if this flag is not specified.

+ The result of a signed conversion shall always begin with a sign ('+' or '−'). The conversion shall begin with a sign only when a negative value is converted if this flag is not specified.

<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.

This flag 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 0. For *x* or *X* conversions, a non-zero result shall have 0x (or 0X) prefixed to it. For *e*, *E*, *f*, *g*, or *G* conversions, the result shall always contain a radix character, even if no

13210 digits follow the radix character. Without this flag, a radix character appears in the
13211 result of these conversions only if a digit follows it. For *g* and *G* conversions, trailing
13212 zeros shall *not* be removed from the result as they normally are. For other conversions,
13213 the behavior is undefined.

13214 0 For *d*, *i*, *o*, *u*, *x*, *X*, *e*, *E*, *f*, *g*, and *G* conversions, leading zeros (following any indication of
13215 sign or base) are used to pad to the field width; no space padding is performed. If the |
13216 `'0'` and `'−'` flags both appear, the `'0'` flag is ignored. For *d*, *i*, *o*, *u*, *x*, and *X* |
13217 XSI conversions, if a precision is specified, the `'0'` flag is ignored. If the `'0'` and `'\''` |
13218 flags both appear, the grouping characters are inserted before zero padding. For other
13219 conversions, the behavior is undefined. |

13220 The length modifiers and their meanings are: |

13221 *hh* Specifies that a following *d*, *i*, *o*, *u*, *x*, or *X* conversion specifier applies to a **signed char** |
13222 or **unsigned char** argument (the argument will have been promoted according to the |
13223 integer promotions, but its value shall be converted to **signed char** or **unsigned char** |
13224 before printing); or that a following *n* conversion specifier applies to a pointer to a |
13225 **signed char** argument. |

13226 *h* Specifies that a following *d*, *i*, *o*, *u*, *x*, or *X* conversion specifier applies to a **short** or |
13227 **unsigned short** argument (the argument will have been promoted according to the |
13228 integer promotions, but its value shall be converted to **short** or **unsigned short** before |
13229 printing); or that a following *n* conversion specifier applies to a pointer to a **short** |
13230 argument. |

13231 *l* (ell) Specifies that a following *d*, *i*, *o*, *u*, *x*, or *X* conversion specifier applies to a **long** or |
13232 **unsigned long** argument; that a following *n* conversion specifier applies to a pointer to |
13233 a **long** argument; that a following *c* conversion specifier applies to a **wint_t** argument; |
13234 that a following *s* conversion specifier applies to a pointer to a **wchar_t** argument; or |
13235 has no effect on a following *a*, *A*, *e*, *E*, *f*, *F*, *g*, or *G* conversion specifier. |

13236 *ll* (ell-ell) Specifies that a following *d*, *i*, *o*, *u*, *x*, or *X* conversion specifier applies to a **long long** or |
13237 **unsigned long long** argument; or that a following *n* conversion specifier applies to a |
13238 pointer to a **long long** argument. |

13239 *j* Specifies that a following *d*, *i*, *o*, *u*, *x*, or *X* conversion specifier applies to an **intmax_t** or |
13240 **uintmax_t** argument; or that a following *n* conversion specifier applies to a pointer to |
13241 an **intmax_t** argument. |

13242 *z* Specifies that a following *d*, *i*, *o*, *u*, *x*, or *X* conversion specifier applies to a **size_t** or the |
13243 corresponding signed integer type argument; or that a following *n* conversion specifier |
13244 applies to a pointer to a signed integer type corresponding to **size_t** argument. |

13245 *t* Specifies that a following *d*, *i*, *o*, *u*, *x*, or *X* conversion specifier applies to a **ptrdiff_t** or |
13246 the corresponding **unsigned** type argument; or that a following n conversion specifier |
13247 applies to a pointer to a **ptrdiff_t** argument. |

13248 *L* Specifies that a following *a*, *A*, *e*, *E*, *f*, *F*, *g*, or*G* conversion specifier applies to a **long** |
13249 **double** argument. |

13250 If a length modifier appears with any conversion specifier other than as specified above, the |
13251 behavior is undefined. |

13252 The conversion characters and their meanings are: |

13253 *d*, *i* The **int** argument is converted to a signed decimal in the style **[**−**]***dddd*. The precision
13254 specifies the minimum number of digits to appear; if the value being converted can be
13255 represented in fewer digits, it shall be expanded with leading zeros. The default

13256 precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.

13257 *o* The **unsigned** argument is converted to unsigned octal format in the style *dddd*. The
13258 precision specifies the minimum number of digits to appear; if the value being
13259 converted can be represented in fewer digits, it shall be expanded with leading zeros.
13260 The default precision is 1. The result of converting 0 with an explicit precision of 0 is no
13261 characters.

13262 *u* The **unsigned** argument is converted to unsigned decimal format in the style *dddd*. The
13263 precision specifies the minimum number of digits to appear; if the value being
13264 converted can be represented in fewer digits, it shall be expanded with leading zeros.
13265 The default precision is 1. The result of converting 0 with an explicit precision of 0 is no
13266 characters.

13267 *x* The **unsigned** argument is converted to unsigned hexadecimal format in the style *dddd*;
13268 the letters "abcdef" are used. The precision specifies the minimum number of digits
13269 to appear; if the value being converted can be represented in fewer digits, it shall be
13270 expanded with leading zeros. The default precision is 1. The result of converting 0 with
13271 an explicit precision of 0 is no characters.

13272 *X* Behaves the same as the *x* conversion character except that letters "ABCDEF" are used
13273 instead of "abcdef".

13274 *f, F* The **double** argument is converted to decimal notation in the style **[–]***ddd.ddd*, where
13275 the number of digits after the radix character is equal to the precision specification. If
13276 the precision is missing, it is taken as 6; if the precision is explicitly 0 and no '#' flag is
13277 present, no radix character appears. If a radix character appears, at least one digit
13278 appears before it. The value is rounded to the appropriate number of digits.

13279 A **double** argument representing an infinity is converted in one of the styles [–]inf or
13280 [–]infinity; which style is implementation-defined. A **double** argument representing a
13281 NaN is converted in one of the styles [–]nan or [–]nan(*n-char-sequence*); which style, and
13282 the meaning of any *n-char-sequence*, is implementation-defined. The *F* conversion
13283 specifier produces INF, INFINITY, or NAN instead of inf, infinity, or nan, respectively.

13284 *e, E* The **double** argument is converted in the style **[–]***d.ddde±dd*, where there is one digit
13285 before the radix character (which is non-zero if the argument is non-zero) and the
13286 number of digits after it is equal to the precision; if the precision is missing, it is taken
13287 as 6; if the precision is 0 and no '#' flag is present, no radix character appears. The
13288 value is rounded to the appropriate number of digits. The *E* conversion character will
13289 produce a number with *E* instead of *e* introducing the exponent. The exponent always
13290 contains at least two digits. If the value is 0, the exponent is 0.

13291 A **double** argument representing an infinity or NaN is converted in the style of an *f* or *F*
13292 conversion specifier.

13293 *g, G* The **double** argument is converted in the style *f* or *e* (or in the style *E* in the case of a *G*
13294 conversion character), with the precision specifying the number of significant digits. If
13295 an explicit precision is 0, it is taken as 1. The style used depends on the value
13296 converted; style *e* (or *E*) shall be used only if the exponent resulting from such a
13297 conversion is less than –4 or greater than or equal to the precision. Trailing zeros are
13298 removed from the fractional portion of the result; a radix character appears only if it is
13299 followed by a digit.

13300 A **double** argument representing an infinity or NaN is converted in the style of an *f* or *F*
13301 conversion specifier.

a, A A **double** argument representing a floating-point number is converted in the style [–]0xh.hhhh p1d, where there is one hexadecimal digit (which is non-zero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character 235) and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and FLT_RADIX is a power of 2, then the precision is 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 is sufficient to distinguish 236) 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 character appears. The letters "abcdef" are used for a conversion and the letters "ABCDEF" for *A* conversion. The *A* conversion specifier produces a number with 'X' and 'P' instead of 'x' and 'p'. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

A **double** argument representing an infinity or NaN is converted in the style of an *f* or *F* conversion specifier.

c The **int** argument is converted to an **unsigned char**, and the resulting byte is written.

If an *l* (ell) qualifier is present, the **wint_t** argument is converted as if by an *ls* conversion specification with no precision and an argument that points to a two-element array of type **wchar_t**, the first element of which contains the **wint_t** argument to the *ls* conversion specification and the second element contains a null wide character.

s The argument shall be a pointer to an array of **char**. Bytes from the array are written up to (but not including) any terminating null byte. If the precision is specified, no more than that many bytes shall be written. If the precision is not specified or is greater than the size of the array, the application shall ensure that the array contains a null byte.

If an *l* (ell) qualifier is present, the argument shall be a pointer to an array of type **wchar_t**. Wide characters from the array are converted to characters (each as if by a call to the *wcrtomb*( ) function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting characters shall be written up to (but not including) the terminating null character (byte). If no precision is specified, the application shall ensure that the array contains a null wide character. If a precision is specified, no more than that many characters (bytes) shall be written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case is a partial character written.

p The argument shall be a pointer to **void**. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.

n The argument shall be a pointer to an integer into which is written the number of bytes written to the output so far by this call to one of the *fprintf*( ) functions. No argument is converted.

XSI *C* Same as *lc*.

XSI *S* Same as *ls*.

% Print a '%'; no argument is converted. The entire conversion specification shall be "%%".

If a conversion specification does not match one of the above forms, the behavior is undefined. If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *fprintf*( ) and *printf*( ) are printed as if *fputc*( ) had been called.

For *a* and *A* conversions, if FLT_RADIX is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

If FLT_RADIX is not a power of 2, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

For *e*, *E*, *f*, *F*, *g*, and *G* conversions, if the number of significant decimal digits is at most DECIMAL_DIG, then the result should be correctly rounded. If the number of significant decimal digits is more than DECIMAL_DIG but the source value is exactly representable with DECIMAL_DIG digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings `"L < U"`, both having DECIMAL_DIG significant digits; the value of the resultant decimal string `"D"` should satisfy `"L <= D <= U"`, with the extra stipulation that the error should have a correct sign for the current rounding direction.

CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the call to a successful execution of *fprintf*( ) or *printf*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same stream or a call to *exit*( ) or *abort*( ).

RETURN VALUE

Upon successful completion, the *fprintf*( ) and *printf*( ) functions shall return the number of bytes transmitted.

Upon successful completion, the *sprintf*( ) function shall return the number of bytes written to *s*, excluding the terminating null byte.

Upon successful completion, the *snprintf*( ) function shall return the number of bytes that would be written to *s* had *n* been sufficiently large excluding the terminating null byte.

If an output error was encountered, these functions shall return a negative value.

If the value of *n* is zero on a call to *snprintf*( ), nothing shall be written, the number of bytes that would have been written had *n* been sufficiently large excluding the terminating null shall be returned, and *s* may be a null pointer.

ERRORS

For the conditions under which *fprintf*( ) and *printf*( ) fail and may fail, refer to *fputc*( ) or *fputwc*( ).

In addition, all forms of *fprintf*( ) may fail if:

XSI [EILSEQ] A wide-character code that does not correspond to a valid character has been detected.

XSI [EINVAL] There are insufficient arguments.

The *printf*( ) and *fprintf*( ) functions may fail if:

XSI [ENOMEM] Insufficient storage space is available.

The *snprintf*( ) function shall fail if:

XSI [EOVERFLOW] The value of *n* is greater than {INT_MAX} or the number of bytes needed to hold the output excluding the terminating null is greater than {INT_MAX}.

EXAMPLES

Printing Language-Independent Date and Time

The following statement can be used to print date and time using a language-independent format:

```
printf(format, weekday, month, day, hour, min);
```

For American usage, *format* could be a pointer to the following string:

```
"%s, %s %d, %d:%.2d\n"
```

This example would produce the following message:

```
Sunday, July 3, 10:02
```

For German usage, *format* could be a pointer to the following string:

```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

This definition of *format* would produce the following message:

```
Sonntag, 3. Juli, 10:02
```

Printing File Information

The following example prints information about the type, permissions, and number of links of a specific file in a directory.

The first two calls to *printf*( ) use data decoded from a previous *stat*( ) call. The user-defined *strperm*( ) function shall return a string similar to the one at the beginning of the output for the following command:

```
ls -l
```

The next call to *printf*( ) outputs the owner's name if it is found using *getpwuid*( ); the *getpwuid*( ) function shall return a **passwd** structure from which the name of the user is extracted. If the user name is not found, the program instead prints out the numeric value of the user ID.

The next call prints out the group name if it is found using *getgrgid*( ); *getgrgid*( ) is very similar to *getpwuid*( ) except that it shall return group information based on the group number. Once again, if the group is not found, the program prints the numeric value of the group for the entry.

The final call to *printf*( ) prints the size of the file.

```
#include <stdio.h>
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>

char *strperm (mode_t);
...
struct stat statbuf;
struct passwd *pwd;
struct group *grp;
...
printf("%10.10s", strperm (statbuf.st_mode));
```

```
13432       printf("%4d", statbuf.st_nlink);
13433       if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
13434           printf(" %-8.8s", pwd->pw_name);
13435       else
13436           printf(" %-8ld", (long) statbuf.st_uid);
13437       if ((grp = getgrgid(statbuf.st_gid)) != NULL)
13438           printf(" %-8.8s", grp->gr_name);
13439       else
13440           printf(" %-8ld", (long) statbuf.st_gid);
13441       printf("%9jd", (intmax_t) statbuf.st_size);
13442       ...
```

13443 **Printing a Localized Date String**

13444 The following example gets a localized date string. The *nl_langinfo*( ) function shall return the
13445 localized date string, which specifies the order and layout of the date. The *strftime*( ) function
13446 takes this information and, using the **tm** structure for values, places the date and time
13447 information into *datestring*. The *printf*( ) function then outputs *datestring* and the name of the
13448 entry.

```
13449       #include <stdio.h>
13450       #include <time.h>
13451       #include <langinfo.h>
13452       ...
13453       struct dirent *dp;
13454       struct tm *tm;
13455       char datestring[256];
13456       ...
13457       strftime(datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm);
13458       printf(" %s %s\n", datestring, dp->d_name);
13459       ...
```

13460 **Printing Error Information**

13461 The following example uses *fprintf*( ) to write error information to stadard error.

13462 In the first group of calls, the program tries to open the password lock file named **LOCKFILE**. If
13463 the file already exists, this is an error, as indicated by the O_EXCL flag on the *open*( ) function. If
13464 the call fails, the program assumes that someone else is updating the password file, and the
13465 program exits.

13466 The next group of calls saves a new password file as the current password file by creating a link
13467 between **LOCKFILE** and the new password file **PASSWDFILE**.

```
13468       #include <sys/types.h>
13469       #include <sys/stat.h>
13470       #include <fcntl.h>
13471       #include <stdio.h>
13472       #include <stdlib.h>
13473       #include <unistd.h>
13474       #include <string.h>
13475       #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);
}
...
```

Printing Usage Information

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.

```
#include <stdio.h>
#include <stdlib.h>
...
char *Options = "hdbtl";
...
if (argc < 2) {
    fprintf(stderr, "Usage: %s -%s <file\n", argv[0], Options); exit(1);
}
...
```

Formatting a Decimal String

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.

```
#include <stdio.h>
...
long i;
char *keystr;
int elementlen, len;
...
while (len < elementlen) {
...
    printf("%s Element%0*ld\n", keystr, elementlen, i);
...
}
```

13520 **Creating a File Name**

13521 The following example creates a file name using information from a previous *getpwnam*( )
13522 function that returned the HOME directory of the user.

```
13523 #include <stdio.h>
13524 #include <sys/types.h>
13525 #include <unistd.h>
13526 ...
13527 char filename[PATH_MAX+1];
13528 struct passwd *pw;
13529 ...
13530 sprintf(filename, "%s/%d.out", pw->pw_dir, getpid());
13531 ...
```

13532 **Reporting an Event**

13533 The following example loops until an event has timed out. The *pause*( ) function waits forever
13534 unless it receives a signal. The *fprintf*( ) statement should never occur due to the possible return
13535 values of *pause*( ).

```
13536 #include <stdio.h>
13537 #include <unistd.h>
13538 #include <string.h>
13539 #include <errno.h>
13540 ...
13541 while (!event_complete) {
13542 ...
13543     if (pause() != -1 || errno != EINTR)
13544         fprintf(stderr, "pause: unknown error: %s\n", strerror(errno));
13545 }
13546 ...
```

13547 **Printing Monetary Information**

13548 The following example uses *strfmon*( ) to convert a number and store it as a formatted monetary
13549 string named *convbuf*. If the first number is printed, the program prints the format and the
13550 description; otherwise, it just prints the number.

```
13551 #include <monetary.h>
13552 #include <stdio.h>
13553 ...
13554 struct tblfmt {
13555     char *format;
13556     char *description;
13557 };
13558 struct tblfmt table[] = {
13559     { "%n", "default formatting" },
13560     { "%11n", "right align within an 11 character field" },
13561     { "%#5n", "aligned columns for values up to 99,999" },
13562     { "%=*#5n", "specify a fill character" },
13563     { "%=0#5n", "fill characters do not use grouping" },
13564     { "%^#5n", "disable the grouping separator" },
13565     { "%^#5.0n", "round off to whole units" },
```

```
        { "%^#5.4n", "increase the precision" },
        { "%(#5n", "use an alternative pos/neg style" },
        { "%!(#5n", "disable the currency symbol" },
    };
    ...
    float input[3];
    int i, j;
    char convbuf[100];
    ...
    strfmon(convbuf, sizeof(convbuf), table[i].format, input[j]);
    if (j == 0) {
        printf("%s%s%s\n", table[i].format,
            convbuf, table[i].description);
    }
    else {
        printf("%s\n", convbuf);
    }
    ...
```

APPLICATION USAGE

If the application calling *fprintf*( ) has any objects of type **wint_t** or **wchar_t**, it must also include the **<wchar.h>** header to have these objects defined.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fputc( ), *fscanf*( ), *setlocale*( ), *wcrtomb*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

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

Issue 4

In the DESCRIPTION, references to *langinfo* data are marked as extensions. The reference to *langinfo* data is removed from the description of the radix character.

The ''' (single-quote) flag is added to the list of flag characters and marked as an extension. This flag directs that numeric conversion is formatted with the decimal grouping character.

The detailed description of these functions is provided here instead of under *printf*( ).

The information in the APPLICATION USAGE section is moved to the DESCRIPTION. A new APPLICATION USAGE section is added.

The [EILSEQ] error is added to the ERRORS section and all errors are marked as extensions.

The following changes are incorporated for alignment with the ISO C standard:

- The type of the *format* arguments is changed from **char*** to **const char***.
- The DESCRIPTION is reworded or presented differently in a number of places for alignment with the ISO C standard, and also for clarity. There are no functional changes, except as noted elsewhere in this CHANGE HISTORY section.

The following changes are incorporated for alignment with the MSE working draft:

- The *C* and *S* conversion characters are added, indicating respectively a wide character of type **wchar_t** and pointer to a wide-character string of type **wchar_t*** in the argument list.

Issue 4, Version 2

The [ENOMEM] error is added to the ERRORS section as an optional error.

Issue 5

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the *l* (ell) qualifier can now be used with *c* and *s* conversion characters.

The *snprintf*( ) function is new in Issue 5.

Issue 6

Extensions beyond the ISO C standard are now marked.

The description of *snprintf*( ) has been aligned with The Open Group Base Resolution bwg98-006. This aligns *snprintf*( ) with historic BSD behavior.

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

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

- The prototypes for *fprintf*( ), *printf*( ), *snprintf*( ), and *sprintf*( ) are updated, and the XSI shading is removed from *snprintf*( ).
- The DESCRIPTION is updated.

13628 **NAME**

13629 fputc — put a byte on a stream

13630 **SYNOPSIS**

13631 `#include <stdio.h>`

13632 `int fputc(int c, FILE *stream);`

13633 **DESCRIPTION**

13634 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13635 conflict between the requirements described here and the ISO C standard is unintentional. This
13636 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

13637 The *fputc*( ) function shall write the byte specified by *c* (converted to an **unsigned char**) to the
13638 output stream pointed to by *stream*, at the position indicated by the associated file-position
13639 indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot
13640 support positioning requests, or if the stream was opened with append mode, the byte shall be
13641 appended to the output stream.

13642 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
13643 execution of *fputc*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same
13644 stream or a call to *exit*( ) or *abort*( ).

13645 **RETURN VALUE**

13646 Upon successful completion, *fputc*( ) shall return the value it has written. Otherwise, it shall
13647 CX return EOF, the error indicator for the stream shall be set, and *errno* shall be set to indicate the
13648 error.

13649 **ERRORS**

13650 The *fputc*( ) function shall fail if either the *stream* is unbuffered or the *stream*'s buffer needs to be
13651 flushed, and:

13652 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation. |

13654 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for writing. |

13656 CX [EFBIG] An attempt was made to write to a file that exceeds the maximum file size. |

13657 XSI [EFBIG] An attempt was made to write to a file that exceeds the process' file size limit. |

13658 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the offset maximum. |

13660 CX [EINTR] The write operation was terminated due to the receipt of a signal, and no data was transferred. |

13662 CX [EIO] A physical I/O error has occurred, or the process is a member of a background process group attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions. |

13667 CX [ENOSPC] There was no free space remaining on the device containing the file. |

13668 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal shall also be sent to the thread. |

13670 The *fputc*( ) function may fail if:

CX [ENOMEM] Insufficient storage space is available.

CX [ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ferror( ), *fopen*( ), *getrlimit*( ), *putc*( ), *puts*( ), *setbuf*( ), *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 4

In the DESCRIPTION, the text is changed to make it clear that the function writes byte values, rather than (possibly multi-byte) character values.

In the ERRORS section, text is added to indicate that error returns are only generated when either the stream is unbuffered, or if the stream buffer needs to be flushed.

Also in the ERRORS section, in previous issues generation of the [EIO] error depended on whether or not an implementation supported Job Control. This functionality is now defined as mandatory.

The [ENXIO] error is moved to the list of optional errors, and all the optional errors are marked as extensions.

The description of [EINTR] is amended.

The [EFBIG] error is marked to show extensions.

Issue 4, Version 2

In the ERRORS section, the description of [EIO] is updated to include the case where a physical I/O error occurs.

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] and [EFBIG] mandatory error conditions are added.
- The [ENOMEM] and [ENXIO] optional error conditions are added.

13710 **NAME**

13711 fputs — put a string on a stream

13712 **SYNOPSIS**

13713 ```
#include <stdio.h>
```

13714 ```
int fputs(const char *restrict s, FILE *restrict stream);
```

13715 **DESCRIPTION**

13716 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13717 conflict between the requirements described here and the ISO C standard is unintentional. This
13718 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

13719 The *fputs*( ) function shall write the null-terminated string pointed to by *s* to the stream pointed
13720 to by *stream*. The terminating null byte shall not be written.

13721 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
13722 execution of *fputs*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same
13723 stream or a call to *exit*( ) or *abort*( ).

13724 **RETURN VALUE**

13725 Upon successful completion, *fputs*( ) shall return a non-negative number. Otherwise, it shall
13726 CX return EOF, set an error indicator for the stream, and set *errno* to indicate the error.

13727 **ERRORS**

13728 Refer to *fputc*( ).

13729 **EXAMPLES**

13730 **Printing to Standard Output**

13731 The following example gets the current time, converts it to a string using *localtime*( ) and
13732 *asctime*( ), and prints it to standard output using *fputs*( ). It then prints the number of minutes to
13733 an event for which it is waiting.

```
13734 #include <time.h>
13735 #include <stdio.h>
13736 ...
13737 time_t now;
13738 int minutes_to_event;
13739 ...
13740 time(&now);
13741 printf("The time is ");
13742 fputs(asctime(localtime(&now)), stdout);
13743 printf("There are still %d minutes to the event.\n",
13744     minutes_to_event);
13745 ...
```

13746 **APPLICATION USAGE**

13747 The *puts*( ) function appends a <newline> character while *fputs*( ) does not.

13748 **RATIONALE**

13749 None.

13750 **FUTURE DIRECTIONS**

13751 None.

13752 **SEE ALSO**

13753 *fopen*( ), *putc*( ), *puts*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**

13754 **CHANGE HISTORY**

13755 First released in Issue 1. Derived from Issue 1 of the SVID.

13756 **Issue 4**

13757 In the DESCRIPTION, the words ‘‘null character’’ are replaced by ‘‘null byte’’, to make it clear
13758 that this function deals solely in byte values.

13759 The following change is incorporated for alignment with the ISO C standard:

13760 • The type of argument *s* is changed from **char*** to **const char***.

13761 **Issue 6**

13762 Extensions beyond the ISO C standard are now marked.

13763 The *fputs*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

13764 **NAME**

13765 fputwc — put a wide-character code on a stream

13766 **SYNOPSIS**

13767 ```
#include <stdio.h>
```

13768 ```
#include <wchar.h>
```

13769 ```
wint_t fputwc(wchar_t wc, FILE *stream);
```

13770 **DESCRIPTION**

13771 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13772 conflict between the requirements described here and the ISO C standard is unintentional. This
13773 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

13774 The *fputwc*( ) function shall write the character corresponding to the wide-character code *wc* to
13775 the output stream pointed to by *stream*, at the position indicated by the associated file-position
13776 indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot
13777 support positioning requests, or if the stream was opened with append mode, the character is
13778 appended to the output stream. If an error occurs while writing the character, the shift state of |
13779 the output file is left in an undefined state.

13780 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
13781 execution of *fputwc*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the
13782 same stream or a call to *exit*( ) or *abort*( ).

13783 **RETURN VALUE**

13784 Upon successful completion, *fputwc*( ) shall return *wc*. Otherwise, it shall return WEOF, the error
13785 CX indicator for the stream shall be set set, and *errno* shall be set to indicate the error.

13786 **ERRORS**

13787 The *fputwc*( ) function shall fail if either the stream is unbuffered or data in the *stream*'s buffer
13788 needs to be written, and:

13789 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the |
13790 process would be delayed in the write operation.

13791 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for |
13792 writing.

13793 CX [EFBIG] An attempt was made to write to a file that exceeds the maximum file size or |
13794 the process' file size limit.

13795 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the |
13796 offset maximum associated with the corresponding stream.

13797 CX [EINTR] The write operation was terminated due to the receipt of a signal, and no data |
13798 was transferred.

13799 CX [EIO] A physical I/O error has occurred, or the process is a member of a |
13800 background process group attempting to write to its controlling terminal,
13801 TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the
13802 process group of the process is orphaned. This error may also be returned
13803 under implementation-defined conditions. |

13804 CX [ENOSPC] There was no free space remaining on the device containing the file. |

13805 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by |
13806 any process. A SIGPIPE signal shall also be sent to the thread.
```

The *fputwc*( ) function may fail if:

CX [ENOMEM] Insufficient storage space is available.

CX [ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

CX [EILSEQ] The wide-character code *wc* does not correspond to a valid character.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ferror( ), *fopen*( ), *setbuf*( ), *ulimit*( ), 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 4, Version 2

In the ERRORS section, the description of [EIO] is updated to include the case where a physical I/O error occurs.

Issue 5

Aligned with ISO/IEC 9899: 1990/Amendment 1: 1995 (E). Specifically, the type of argument *wc* is changed from **wint_t** to **wchar_t**.

The Optional Header (OH) marking is removed from **<stdio.h>**.

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EFBIG] and [EIO] mandatory error conditions are added.

13838 **NAME**

13839 fputws — put a wide-character string on a stream

13840 **SYNOPSIS**

13841 ```
#include <stdio.h>
```

13842 ```
#include <wchar.h>
```

13843 ```
int fputws(const wchar_t *restrict ws, FILE *restrict stream);
```

13844 **DESCRIPTION**

13845 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13846 conflict between the requirements described here and the ISO C standard is unintentional. This
13847 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

13848 The *fputws*( ) function shall write a character string corresponding to the (null-terminated)
13849 wide-character string pointed to by *ws* to the stream pointed to by *stream*. No character
13850 corresponding to the terminating null wide-character code shall be written.

13851 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
13852 execution of *fputws*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the
13853 same stream or a call to *exit*( ) or *abort*( ).

13854 **RETURN VALUE**

13855 Upon successful completion, *fputws*( ) shall return a non-negative number. Otherwise, it shall
13856 CX return –1, set an error indicator for the stream, and set *errno* to indicate the error.

13857 **ERRORS**

13858 Refer to *fputwc*( ).

13859 **EXAMPLES**

13860 None.

13861 **APPLICATION USAGE**

13862 The *fputws*( ) function does not append a <newline> character.

13863 **RATIONALE**

13864 None.

13865 **FUTURE DIRECTIONS**

13866 None.

13867 **SEE ALSO**

13868 *fopen*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**, **<wchar.h>**

13869 **CHANGE HISTORY**

13870 First released in Issue 4. Derived from the MSE working draft.

13871 **Issue 5**

13872 The Optional Header (OH) marking is removed from **<stdio.h>**.

13873 **Issue 6**

13874 Extensions beyond the ISO C standard are now marked.

13875 The *fputws*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.
```

NAME

fread — binary input

SYNOPSIS

```
#include <stdio.h>

size_t fread(void *restrict ptr, size_t size, size_t nitems,
    FILE *restrict 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 *fread*( ) function shall read into the array pointed to by *ptr* up to *nitems* members whose size is specified by *size* in bytes, from the stream pointed to by *stream*. For each object, size calls are made to the *fgetc*( ) function and the results stored, in the order read, in an array of **unsigned char** exactly overlaying the object. The file position indicator for the stream (if defined) is advanced by the number of bytes successfully read. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. If a partial member is read, its value is indeterminate.

CX The *fread*( ) function may mark the *st_atime* field of the file associated with *stream* for update. The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ), *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

RETURN VALUE

Upon successful completion, *fread*( ) shall return the number of members successfully read which is less than *nitems* only if a read error or end-of-file is encountered. If *size* or *nitems* is 0, *fread*( ) shall return 0 and the contents of the array and the state of the stream remain unchanged.

CX Otherwise, if a read error occurs, the error indicator for the stream shall be set, and *errno* shall be set to indicate the error.

ERRORS

Refer to *fgetc*( ).

EXAMPLES

Reading from a Stream

The following example reads a single element from the *fp* stream into the array pointed to by *buf*.

```
#include <stdio.h>
...
size_t bytes_read;
char buf[100];
FILE *fp;
...
bytes_read = fread(buf, sizeof(buf), 1, fp);
...
```

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 member 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 4

The list of functions that may cause the *st_atime* field to be updated is revised.

The following change is incorporated for alignment with the ISO C standard:

- In the RETURN VALUE section, the behavior if *size* or *nitems* is 0 is defined.

Issue 6

Extensions beyond the ISO C standard are now 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.

NAME

free — free allocated memory

SYNOPSIS

```
#include <stdlib.h>

void free(void *ptr);
```

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 *free*( ) function shall cause the space pointed to by *ptr* to be deallocated; that is, made available for further allocation. If *ptr* is a null pointer, no action shall occur. Otherwise, if the argument does not match a pointer earlier returned by the *calloc*( ), *malloc*( ), *posix_memalign*( ), or *realloc*( ) function, or if the space is deallocated by a call to *free*( ) or *realloc*( ), the behavior is undefined.

Any use of a pointer that refers to freed space results in undefined behavior.

RETURN VALUE

The *free*( ) function shall return no value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

There is now no requirement for the implementation to support the inclusion of **<malloc.h>**.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

calloc( ), *malloc*( ), *realloc*( ), 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 4

The APPLICATION USAGE section is changed to record that **<malloc.h>** need no longer be supported on XSI-conformant systems.

The following change is incorporated for alignment with the ISO C standard:

- The DESCRIPTION now states that the behavior is undefined if any use is made of a pointer that refers to freed space. This was implied but not stated explicitly in Issue 3.

Issue 4, Version 2

The DESCRIPTION is updated for X/OPEN UNIX conformance to indicate that the *free*( ) function can also be used to free memory allocated by *valloc*( ).

13980 **Issue 6**

13981 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 frees 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 is 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.

The *freeaddrinfo*( ) and *getaddrinfo*( ) functions shall be thread-safe.

The *nodename* and *servname* arguments are either null pointers or pointers to null-terminated strings. One or both of these two arguments shall be supplied by the application as a non-null pointer.

The format of a valid name depends on the protocol family or families. If a specific family is not given and the name could be interpreted as valid within multiple supported families, the implementation shall attempt to resolve the name in all supported families and, in absence of errors, one or more results shall be returned.

If the *nodename* argument is not null, it can be a descriptive name or can be an address string. If IP6 the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, valid descriptive names include host names. If the specified address family is AF_INET or AF_UNSPEC, address strings using Internet standard dot notation as specified in *inet_addr*( ) are valid.

IP6 If the specified address family is AF_INET6 or AF_UNSPEC, standard IPv6 text forms described in *inet_ntop*( ) are valid.

If *nodename* is not null, the requested service location is named by *nodename*; otherwise, the requested service location is local to the caller.

If *servname* is null, the call shall return network-level addresses for the specified *nodename*. If *servname* is not null, it is a null-terminated character string identifying the requested service. This can be either a descriptive name or a numeric representation suitable for use with the address IP6 family or families. If the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, the service can be specified as a string specifying a decimal port number.

If the *hints* argument is not null, it refers to a structure containing input values that may direct the operation by providing options and by limiting the returned information to a specific socket type, address family and/or protocol. In this *hints* structure every member other than *ai_flags*, *ai_family*, *ai_socktype*, and *ai_protocol* shall be set to zero or a null pointer. A value of AF_UNSPEC for *ai_family* means that the caller shall accept any protocol family. A value of zero for *ai_socktype* means that the caller shall accept any socket type. A value of zero for *ai_protocol* means that the caller shall accept any protocol. If *hints* is a null pointer, the behavior shall be as if it referred to a structure containing the value zero for the *ai_flags*, *ai_socktype*, and *ai_protocol* fields, and AF_UNSPEC for the *ai_family* field.

Notes:

1. 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.
2. If the caller handles only IPv4 and not IPv6, then the *ai_family* member of the *hints* structure should be set to PF_INET when *getaddrinfo*( ) is called.

The *ai_flags* field to which the *hints* parameter points shall be set to zero or be the bitwise-inclusive OR of one or more of the values AI_PASSIVE, AI_CANONNAME, and AI_NUMERICHOST.

If the AI_PASSIVE flag is specified, the returned address information shall be suitable for use in binding a socket for accepting incoming connections for the specified service. In this case, if the *nodename* argument is null, then the IP address portion of the socket address structure shall be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address. If the AI_PASSIVE flag is not specified, the returned address information shall be suitable for a call to *connect*( ) (for a connection-mode protocol) or for a call to *connect*( ), *sendto*( ), or *sendmsg*( ) (for a connectionless protocol). In this case, if the *nodename* argument is null, then the IP address portion of the socket address structure shall be set to the loopback address.

If the AI_CANONNAME flag is specified and the *nodename* argument is not null, the function attempts to determine the canonical name corresponding to *nodename* (for example, if *nodename* is an alias or shorthand notation for a complete name).

If the AI_NUMERICHOST flag is specified, then a non-null *nodename* string supplied shall be a numeric host address string. Otherwise, an [EAI_NONAME] error is returned. This flag prevents any type of name resolution service (for example, the DNS) from being invoked.

If the AI_NUMERICSERV flag is specified, then a non-null *servname* string supplied shall be a numeric port string. Otherwise, an [EAI_NONAME] error is returned. This flag prevents any type of name resolution service (for example, NIS+) from being invoked.

The *ai_socktype* field to which argument *hints* points specifies the socket type for the service, as defined in *socket*( ). If a specific socket type is not given (for example, a value of zero) and the service name could be interpreted as valid with multiple supported socket types, the implementation shall attempt to resolve the service name for all supported socket types and, in the absence of errors, all possible results shall be returned. A non-zero socket type value shall limit the returned information to values with the specified socket type.

If the *ai_family* field to which *hints* points has the value AF_UNSPEC, addresses are returned for use with any protocol family that can be used with the specified *nodename* and/or *servname*. Otherwise, addresses are returned for use only with the specified protocol family. If *ai_family* is not AF_UNSPEC and *ai_protocol* is not zero, then addresses are returned for use only with the specified protocol family and protocol; the value of *ai_protocol* is interpreted as in a call to the *socket*( ) function with the corresponding values of *ai_family* and *ai_protocol*.

RETURN VALUE

A zero return value for *getaddrinfo*( ) indicates successful completion; a non-zero return value indicates failure. The possible values for the failures are listed in the ERRORS section.

Upon successful return of *getaddrinfo*( ), the location to which *res* points refers to a linked list of **addrinfo** structures, each of which specifies a socket address and information for use in creating a socket with which to use that socket address. The list shall include at least one **addrinfo** structure. The *ai_next* field of each structure contains a pointer to the next structure on the list, or a null pointer if it is the last structure on the list. Each structure on the list includes values for use with a call to the *socket*( ) function, and a socket address for use with the *connect*( ) function or, if

the AI_PASSIVE flag was specified, for use with the *bind*( ) function. The fields *ai_family*, *ai_socktype*, and *ai_protocol* are usable as the arguments to the *socket*( ) function to create a socket suitable for use with the returned address. The fields *ai_addr* and *ai_addrlen* are usable as the arguments to the *connect*( ) or *bind*( ) functions with such a socket, according to the AI_PASSIVE flag.

If *nodename* is not null, and if requested by the AI_CANONNAME flag, the *ai_canonname* field of the first returned **addrinfo** structure points to a null-terminated string containing the canonical name corresponding to the input *nodename*; if the canonical name is not available, then *ai_canonname* refers to the *nodename* argument or a string with the same contents. The contents of the *ai_flags* field of the returned structures are undefined.

All fields in socket address structures returned by *getaddrinfo*( ) that are not filled in through an explicit argument (for example, *sin6_flowinfo* and *sin_zero*) shall be set to zero.

Note: This makes it easier to compare socket address structures.

ERRORS

The *getaddrinfo*( ) function shall fail and return the corresponding value if:

[EAI_AGAIN] The name could not be resolved at this time. Future attempts may succeed.

[EAI_BADFLAGS] The *flags* parameter had an invalid value.

[EAI_FAIL] A non-recoverable error occurred when attempting to resolve the name.

[EAI_FAMILY] The address family was not recognized.

[EAI_MEMORY] There was a memory allocation failure when trying to allocate storage for the return value.

[EAI_NONAME] The name does not resolve for the supplied parameters.

Neither *nodename* nor *servname* were supplied. At least one of these shall be supplied.

[EAI_SERVICE] The service passed was not recognized for the specified socket type.

[EAI_SOCKTYPE] The intended socket type was not recognized.

[EAI_SYSTEM] A system error occurred; the error code can be found in *errno*.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

connect( ), *gethostbyname*( ), *getipnodebyname*( ), *getnameinfo*( ), *getservbyname*( ), *socket*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<netdb.h>**, **<sys/socket.h>**

14115 **CHANGE HISTORY**

14116 First released in Issue 6. Derived from the XNS, Issue 5.2 specification. |

14117 The **restrict** keyword is added to the *getaddrinfo*() prototype for alignment with the |
14118 ISO/IEC 9899:1999 standard. |

14119 **NAME**

14120 freehostent — network host database functions

14121 **SYNOPSIS**

14122 `#include <netdb.h>`

14123 `void freehostent(struct hostent *`*`ptr`*`);`

14124 **DESCRIPTION**

14125 Refer to *endhostent*( ).

14126 **NAME**

14127 freopen — open a stream

14128 **SYNOPSIS**

14129 `#include <stdio.h>`

14130 `FILE *freopen(const char *restrict filename, const char *restrict mode,`
14131 `FILE *restrict stream);`

14132 **DESCRIPTION**

14133 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14134 conflict between the requirements described here and the ISO C standard is unintentional. This
14135 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

14136 The *freopen*( ) function shall first attempt to flush the stream and close any file descriptor
14137 associated with *stream*. Failure to flush or close the file successfully shall be ignored. The error
14138 and end-of-file indicators for the stream shall be cleared.

14139 The *freopen*( ) function shall open the file whose path name is the string pointed to by *filename*
14140 and associate the stream pointed to by *stream* with it. The *mode* argument shall be used just as in
14141 *fopen*( ).

14142 The original stream shall be closed regardless of whether the subsequent open succeeds.

14143 If *filename* is a null pointer, the *freopen*( ) function shall attempt to change the mode of the stream
14144 to that specified by *mode*, as if the name of the file currently associated with the stream had been
14145 used. It is implementation-defined which changes of mode are permitted (if any), and under
14146 what circumstances.

14147 XSI After a successful call to the *freopen*( ) function, the orientation of the stream shall be cleared, the
14148 encoding rule shall be cleared, and the associated **mbstate_t** object shall be set to describe an
14149 initial conversion state.

14150 CX The largest value that can be represented correctly in an object of type **off_t** shall be established
14151 as the offset maximum in the open file description.

14152 **RETURN VALUE**

14153 Upon successful completion, *freopen*( ) shall return the value of *stream*. Otherwise, a null pointer
14154 CX shall be returned, and *errno* shall be set to indicate the error.

14155 **ERRORS**

14156 The *freopen*( ) function shall fail if:

14157 CX [EACCES] Search permission is denied on a component of the path prefix, or the file
14158 exists and the permissions specified by *mode* are denied, or the file does not
14159 exist and write permission is denied for the parent directory of the file to be
14160 created.

14161 CX [EINTR] A signal was caught during *freopen*( ).

14162 CX [EISDIR] The named file is a directory and *mode* requires write access.

14163 CX [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
14164 argument.

14165 CX [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

14166 CX [ENAMETOOLONG]
14167 The length of the *filename* argument exceeds {PATH_MAX} or a path name
14168 component is longer than {NAME_MAX}.

14169 CX [ENFILE] The maximum allowable number of files is currently open in the system.

14170 CX [ENOENT] A component of *filename* does not name an existing file or *filename* is an empty
14171 string.

14172 CX [ENOSPC] The directory or file system that would contain the new file cannot be
14173 expanded, the file does not exist, and it was to be created.

14174 CX [ENOTDIR] A component of the path prefix is not a directory.

14175 CX [ENXIO] The named file is a character special or block special file, and the device
14176 associated with this special file does not exist.

14177 CX [EOVERFLOW] The named file is a regular file and the size of the file cannot be represented
14178 correctly in an object of type **off_t**.

14179 CX [EROFS] The named file resides on a read-only file system and *mode* requires write
14180 access.

14181 The *freopen*( ) function may fail if:

14182 CX [EINVAL] The value of the *mode* argument is not valid.

14183 CX [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
14184 resolution of the *path* argument.

14185 CX [ENAMETOOLONG]
14186 Path name resolution of a symbolic link produced an intermediate result
14187 whose length exceeds {PATH_MAX}.

14188 CX [ENOMEM] Insufficient storage space is available.

14189 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
14190 capabilities of the device.

14191 CX [ETXTBSY] The file is a pure procedure (shared text) file that is being executed and *mode*
14192 requires write access.

14193 **EXAMPLES**

14194 **Directing Standard Output to a File**

14195 The following example logs all standard output to the **/tmp/logfile** file.

```
14196 #include <stdio.h>
14197 ...
14198 FILE *fp;
14199 ...
14200 fp = freopen ("/tmp/logfile", "a+", stdout);
14201 ...
```

14202 **APPLICATION USAGE**

14203 The *freopen*( ) function is typically used to attach the preopened *streams* associated with *stdin*,
14204 *stdout*, and *stderr* to other files.

14205 **RATIONALE**

14206 None.

14207 **FUTURE DIRECTIONS**

14208 None.

14209 **SEE ALSO**

14210 *fclose*( ), *fopen*( ), *fdopen*( ), *mbsinit*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
14211 **<stdio.h>**

14212 **CHANGE HISTORY**

14213 First released in Issue 1. Derived from Issue 1 of the SVID. |

14214 **Issue 4**

14215 In the DESCRIPTION, the word ''name'' is replaced by ''path name'', to make it clear that the
14216 function is not limited to accepting file names only.

14217 In the ERRORS section:

14218 • The description of the [EMFILE] error has been changed to refer to {OPEN_MAX} file
14219 descriptors rather than {FOPEN_MAX} file descriptors, directories, and message catalogs.

14220 • The errors [EINVAL], [ENOMEM], and [ETXTBSY] are marked as extensions.

14221 • The [ENXIO] error is added in the ''may fail'' section and marked as an extension.

14222 The following change is incorporated for alignment with the ISO C standard:

14223 • The type of arguments *filename* and *mode* are changed from **char*** to **const char***.

14224 The following change is incorporated for alignment with the FIPS requirements:

14225 • In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path
14226 name component is larger that {NAME_MAX} is now defined as mandatory and marked as
14227 an extension.

14228 **Issue 4, Version 2**

14229 The ERRORS section is updated for X/OPEN UNIX conformance as follows:

14230 • It states that [ELOOP] is returned if too many symbolic links are encountered during path
14231 name resolution.

14232 • A second [ENAMETOOLONG] condition is defined that may report excessive length of an
14233 intermediate result of path name resolution of a symbolic link.

14234 **Issue 5**

14235 The DESCRIPTION is updated to indicate that the orientation of the stream is cleared and the
14236 conversion state of the stream is set to an initial conversion state by a successful call to the
14237 *freopen*( ) function.

14238 Large File Summit extensions are added.

14239 **Issue 6**

14240 Extensions beyond the ISO C standard are now marked.

14241 The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

14242 • The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC.
14243 This is since behavior may vary from one file system to another.

14244 The following new requirements on POSIX implementations derive from alignment with the
14245 Single UNIX Specification:

14246 • In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file
14247 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.

14258 **NAME**

14259 frexp, frexpf, frexpl — extract mantissa and exponent from a double precision number

14260 **SYNOPSIS**

14261 `#include <math.h>`

14262 `double frexp(double num, int *exp);`
14263 `float frexpf(float value, int *exp);`
14264 `long double frexpl(long double value, int *exp);`

14265 **DESCRIPTION**

14266 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14267 conflict between the requirements described here and the ISO C standard is unintentional. This
14268 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

14269 These functions breaks a floating-point number into a normalized fraction and an integral power
14270 of 2. It stores the integer exponent in the **int** object pointed to by *exp*.

14271 An application wishing to check for error situations should set *errno* to 0 before calling *frexp*( ). If
14272 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

14273 **RETURN VALUE**

14274 These functions shall return the value *x*, such that *x* has a magnitude in the interval [½,1) or 0,
14275 and *num* equals *x* times 2 raised to the power **exp*.

14276 If *num* is 0, both parts of the result shall be 0.

14277 XSI If *num* is NaN, NaN shall be returned, *errno* may be set to [EDOM], and the value of **exp* shall be
14278 unspecified.

14279 If *num* is ±Inf, *num* shall be returned, *errno* may be set to [EDOM], and the value of **exp* shall be
14280 unspecified.

14281 **ERRORS**

14282 These functions may fail if:

14283 XSI [EDOM] The value of *num* is NaN or ±Inf.

14284 XSI No other errors shall occur.

14285 **EXAMPLES**

14286 None.

14287 **APPLICATION USAGE**

14288 None.

14289 **RATIONALE**

14290 None.

14291 **FUTURE DIRECTIONS**

14292 None.

14293 **SEE ALSO**

14294 *isnan*( ), *ldexp*( ), *modf*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

14295 **CHANGE HISTORY**

14296 First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 4

References to *matherr*( ) are removed.

The name of the first argument is changed from *value* to *num*.

The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the ISO C standard and to rationalize error handling in the mathematics functions.

The return value specified for [EDOM] is marked as an extension.

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 *frexpf*( ) and *frexpl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

14309 **NAME**

14310 fscanf, scanf, sscanf — convert formatted input

14311 **SYNOPSIS**

14312 `#include <stdio.h>`

14313 `int fscanf(FILE *restrict stream, const char *restrict format, ... );`
14314 `int scanf(const char *restrict format, ... );`
14315 `int sscanf(const char *restrict s, const char *restrict format, ... );`

14316 **DESCRIPTION**

14317 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14318 conflict between the requirements described here and the ISO C standard is unintentional. This
14319 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

14320 The *fscanf*( ) function reads from the named input *stream*. The *scanf*( ) function reads from the
14321 standard input stream *stdin*. The *sscanf*( ) function reads from the string *s*. Each function reads
14322 bytes, interprets them according to a format, and stores the results in its arguments. Each
14323 expects, as arguments, a control string *format* described below, and a set of *pointer* arguments
14324 indicating where the converted input should be stored. The result is undefined if there are
14325 insufficient arguments for the format. If the format is exhausted while arguments remain, the
14326 excess arguments are evaluated but are otherwise ignored.

14327 XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
14328 to the next unused argument. In this case, the conversion character '%' (see below) is replaced
14329 by the sequence "%n$", where *n* is a decimal integer in the range [1,{NL_ARGMAX}]. This
14330 feature provides for the definition of format strings that select arguments in an order
14331 appropriate to specific languages. In format strings containing the "%n$" form of conversion
14332 specifications, it is unspecified whether numbered arguments in the argument list can be
14333 referenced from the format string more than once.

14334 The *format* can contain either form of a conversion specification—that is, '%' or "%n$"—but the
14335 two forms cannot normally be mixed within a single *format* string. The only exception to this is
14336 that "%%" or "%*" can be mixed with the "%n$" form.

14337 The *fscanf*( ) function in all its forms allows for detection of a language-dependent radix
14338 character in the input string. The radix character is defined in the program's locale (category
14339 *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the
14340 radix character defaults to a period ('.').

14341 The format is a character string, beginning and ending in its initial shift state, if any, composed
14342 of zero or more directives. Each directive is composed of one of the following: one or more
14343 white-space characters (<space>, <tab>, <newline>, <vertical-tab>, or <form-feed> characters);
14344 an ordinary character (neither '%' nor a white-space character); or a conversion specification.
14345 XSI Each conversion specification is introduced by the character '%' or the character sequence
14346 "%n$", after which the following appear in sequence:

14347 • An optional assignment-suppressing character '*'.

14348 • An optional non-zero decimal integer that specifies the maximum field width.

14349 • An option length modifier that specifies the size of the receiving object.

14350 • A conversion character that specifies the type of conversion to be applied. The valid
14351 conversion characters are described below.

14352 The *fscanf*( ) functions execute each directive of the format in turn. If a directive fails, as detailed
14353 below, the function shall return. Failures are described as input failures (due to the
14354 unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space characters is executed by reading input until no more valid input can be read, or up to the first byte which is not a white-space character, which remains unread.

A directive that is an ordinary character is executed as follows: the next byte is read from the input and compared with the byte that comprises the directive; if the comparison shows that they are not equivalent, the directive fails, and the differing and subsequent bytes remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive fails.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion character. A conversion specification is executed in the following steps:

Input white-space characters (as specified by *isspace*()) are skipped, unless the conversion specification includes a '[', *c*, *C*, or *n* conversion character.

An item is read from the input, unless the conversion specification includes an *n* conversion character. An input item is defined as the longest sequence of input bytes (up to any specified maximum field width, which may be measured in characters or bytes dependent on the conversion character) which is an initial subsequence of a matching sequence. The first byte, if any, after the input item remains unread. If the length of the input item is 0, the execution of the conversion specification fails; this condition is a matching failure, unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a '%' conversion character, the input item (or, in the case of a %*n* conversion specification, the count of input bytes) is converted to a type appropriate to the conversion character. If the input item is not a matching sequence, the execution of the conversion specification fails; this condition is a matching failure. Unless assignment suppression was indicated by a '*', the result of the conversion is placed in the object pointed to by the first argument following the *format* argument that has not already received a conversion result if the conversion specification is introduced by '%', or in the *n*th argument if introduced by the character sequence "%n$". If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

XSI

The length modifiers and their meanings are:

hh Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an argument with type pointer to **signed char** or **unsigned char**.

h Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an argument with type pointer to **short** or **unsigned short**.

l (ell) Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an argument with type pointer to **long** or **unsigned long**; that a following *a*, *A*, *e*, *E*, *f*, *F*, *g*, or *G* conversion specifier applies to an argument with type pointer to **double**; or that a following *c*, *s*, or '[' conversion specifier applies to an argument with type pointer to **wchar_t**.

ll (ell-ell) Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an argument with type pointer to **long long** or **unsigned long long**.

j Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an argument with type pointer to **intmax_t** or **uintmax_t**.

z Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an argument with type pointer to **size_t** or the corresponding signed integer type.

14401 *t* Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an argument
14402 with type pointer to **ptrdiff_t** or the corresponding **unsigned** type.

14403 *L* Specifies that a following *a*, *A*, *e*, *E*, *f*, *F*, *g*, or *G* conversion specifier applies to an
14404 argument with type pointer to **long double**.

14405 If a length modifier appears with any conversion specifier other than as specified above, the
14406 behavior is undefined.

14407 The following conversion characters are valid:

14408 *d* Matches an optionally signed decimal integer, whose format is the same as expected for
14409 the subject sequence of *strtol*( ) with the value 10 for the *base* argument. In the absence
14410 of a size modifier, the application shall ensure that the corresponding argument is a
14411 pointer to **int**.

14412 *i* Matches an optionally signed integer, whose format is the same as expected for the
14413 subject sequence of *strtol*( ) with 0 for the *base* argument. In the absence of a size
14414 modifier, the application shall ensure that the corresponding argument is a pointer to
14415 **int**.

14416 *o* Matches an optionally signed octal integer, whose format is the same as expected for
14417 the subject sequence of *strtoul*( ) with the value 8 for the *base* argument. In the absence
14418 of a size modifier, the application shall ensure that the corresponding argument is a
14419 pointer to **unsigned**.

14420 *u* Matches an optionally signed decimal integer, whose format is the same as expected for
14421 the subject sequence of *strtoul*( ) with the value 10 for the *base* argument. In the absence
14422 of a size modifier, the application shall ensure that the corresponding argument is a
14423 pointer to **unsigned**.

14424 *x* Matches an optionally signed hexadecimal integer, whose format is the same as
14425 expected for the subject sequence of *strtoul*( ) with the value 16 for the *base* argument. In
14426 the absence of a size modifier, the application shall ensure that the corresponding
14427 argument is a pointer to **unsigned**.

14428 *a*, *e*, *f*, *g* Matches an optionally signed floating-point number, infinity, or NaN, whose format is
14429 the same as expected for the subject sequence of *strtod*( ). In the absence of a size
14430 modifier, the application shall ensure that the corresponding argument is a pointer to
14431 **float**.

14432 If the *fprintf*( ) family of functions generates character string representations for infinity
14433 and NaN (a symbolic entity encoded in floating-point format) to support
14434 IEEE Std. 754-1985, the *fscanf*( ) family of functions shall recognize them as input.

14435 *s* Matches a sequence of bytes that are not white-space characters. The application shall
14436 ensure that the corresponding argument is a pointer to the initial byte of an array of
14437 **char**, **signed char**, or **unsigned char** large enough to accept the sequence and a
14438 terminating null character code, which shall be added automatically.

14439 If an *l* (ell) qualifier is present, the input is a sequence of characters that begins in the
14440 initial shift state. Each character is converted to a wide character as if by a call to the
14441 *mbrtowc*( ) function, with the conversion state described by an **mbstate_t** object
14442 initialized to zero before the first character is converted. The application shall ensure
14443 that the corresponding argument is a pointer to an array of **wchar_t** large enough to
14444 accept the sequence and the terminating null wide character, which shall be added
14445 automatically.

[Matches a non-empty sequence of bytes from a set of expected bytes (the *scanset*). The normal skip over white-space characters is suppressed in this case. The application shall ensure that the corresponding argument is a pointer to the initial byte of an array of **char**, **signed char**, or **unsigned char** large enough to accept the sequence and a terminating null byte, which shall be added automatically.

If an *l* (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character in the sequence is converted to a wide character as if by a call to the *mbrtowc*( ) function, with the conversion state described by an **mbstate_t** object initialized to zero before the first character is converted. The application shall ensure that the corresponding argument is a pointer to an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which shall be added automatically.

The conversion specification includes all subsequent bytes in the *format* string up to and including the matching right square bracket (']'). The bytes between the square brackets (the *scanlist*) comprise the scanset, unless the byte after the left square bracket is a circumflex ('^'), in which case the scanset contains all bytes that do not appear in the scanlist between the circumflex and the right square bracket. If the conversion specification begins with "[]" or "[^]", the right square bracket is included in the scanlist and the next right square bracket is the matching right square bracket that ends the conversion specification; otherwise, the first right square bracket is the one that ends the conversion specification. If a '–' is in the scanlist and is not the first character, nor the second where the first character is a '^', nor the last character, the behavior is implementation-defined.

c Matches a sequence of bytes of the number specified by the field width (1 if no field width is present in the conversion specification). The application shall ensure that the corresponding argument is a pointer to the initial byte of an array of **char**, **signed char**, or **unsigned char** large enough to accept the sequence. No null byte is added. The normal skip over white-space characters is suppressed in this case.

If an *l* (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character in the sequence is converted to a wide character as if by a call to the *mbrtowc*( ) function, with the conversion state described by an **mbstate_t** object initialized to zero before the first character is converted. The application shall ensure that the corresponding argument is a pointer to an array of **wchar_t** large enough to accept the resulting sequence of wide characters. No null wide character is added.

p Matches an implementation-defined set of sequences, which shall be the same as the set of sequences that is produced by the %*p* conversion of the corresponding *fprintf*( ) functions. The application shall ensure that the corresponding argument is a pointer to a pointer to **void**. The interpretation of the input item is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise, the behavior of the %*p* conversion is undefined.

n No input is consumed. The application shall ensure that the corresponding argument is a pointer to the integer into which is to be written the number of bytes read from the input so far by this call to the *fscanf*( ) functions. Execution of a %*n* conversion specification does not increment the assignment count returned at the completion of execution of the function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.

14495 XSI *C* Same as *lc*.

14496 XSI *S* Same as *ls*.

14497 % Matches a single '%'; no conversion or assignment occurs. The complete conversion
14498 specification shall be "%%".

14499 If a conversion specification is invalid, the behavior is undefined.

14500 The conversion characters *A*, *E*, *F*, *G*, and *X* are also valid and behave the same as, respectively, *a*,
14501 *e*, *f*, *g*, and *x*.

14502 If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before
14503 any bytes matching the current conversion specification (except for %*n*) have been read (other
14504 than leading white-space characters, where permitted), execution of the current conversion
14505 specification terminates with an input failure. Otherwise, unless execution of the current
14506 conversion specification is terminated with a matching failure, execution of the following
14507 conversion specification (if any) is terminated with an input failure.

14508 Reaching the end of the string in *sscanf*( ) is equivalent to encountering end-of-file for *fscanf*( ).

14509 If conversion terminates on a conflicting input, the offending input is left unread in the input.
14510 Any trailing white space (including newline characters) is left unread unless matched by a
14511 conversion specification. The success of literal matches and suppressed assignments is only
14512 directly determinable via the %*n* conversion specification.

14513 The *fscanf*( ) and *scanf*( ) functions may mark the *st_atime* field of the file associated with *stream*
14514 for update. The *st_atime* field shall be marked for update by the first successful execution of
14515 *fgetc*( ), *fgets*( ), *fread*( ), *getc*( ), *getchar*( ), *gets*( ), *fscanf*( ), or *fscanf*( ) using *stream* that returns data
14516 not supplied by a prior call to *ungetc*( ).

14517 **RETURN VALUE**

14518 Upon successful completion, these functions shall return the number of successfully matched
14519 and assigned input items; this number can be 0 in the event of an early matching failure. If the
14520 input ends before the first matching failure or conversion, EOF shall be returned. If a read error
14521 CX occurs, the error indicator for the stream is set, EOF shall be returned, and *errno* shall be set to
14522 indicate the error.

14523 **ERRORS**

14524 For the conditions under which the *fscanf*( ) functions fail and may fail, refer to *fgetc*( ) or
14525 *fgetwc*( ).

14526 In addition, *fscanf*( ) may fail if:

14527 XSI [EILSEQ] Input byte sequence does not form a valid character.

14528 XSI [EINVAL] There are insufficient arguments.

14529 **EXAMPLES**

14530 The call:

```
14531 int i, n; float x; char name[50];
14532 n = scanf("%d%f%s", &i, &x, name);
```

14533 with the input line:

```
14534 25 54.32E-1 Hamster
```

14535 assigns to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* contains the string
14536 "Hamster".

The call:

```
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

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'`.

Reading Data into an Array

The following call uses *fscanf*( ) to read three floating point numbers from standard input into the *input* array.

```
float input[3];
fscanf (stdin, "%f %f %f", input, input+1, input+2);
```

APPLICATION USAGE

If the application calling *fscanf*( ) has any objects of type **wint_t** or **wchar_t**, it must also include the **<wchar.h>** header to have these objects defined.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getc( ), *printf*( ), *setlocale*( ), *strtod*( ), *strtol*( ), *strtoul*( ), *wcrtomb*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<langinfo.h>**, **<stdio.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

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

Issue 4

Use of the terms ‘‘byte’’ and ‘‘character’’ is rationalized to make it clear when single-byte and multi-byte values can be used. Similarly, use of the terms ‘‘conversion specification’’ and ‘‘conversion character’’ is now more precise.

Various errors are corrected. For example, the description of the *d* conversion character contained an erroneous reference to *strtod*( ) in Issue 3. This is replaced in this issue by reference to *strtol*( ).

The DESCRIPTION is updated in a number of places to indicate further implications of the `"%n$"` form of a conversion. All references to this functionality, which is not specified in the ISO C standard, are marked as extensions.

The ERRORS section is changed to refer to the entries for *fgetc*( ) and *fgetwc*( ), the [EINVAL] error is marked as an extension, and the [EILSEQ] error is added and marked as an extension.

The detailed description of this function including the CHANGE HISTORY section for *scanf*( ) is provided here instead of under *scanf*( ).

The APPLICATION USAGE section is amended to record the need for **<sys/types.h>** or **<stddef.h>** if type **wchar_t** is required.

The following changes are incorporated for alignment with the ISO C standard:

- The type of the argument *format* for all functions, and the type of argument *s* for *sscanf*( ), are changed from **char*** to **const char***.
- The description is updated in various places to align more closely with the text of the ISO C standard. In particular, this issue fully defines the *L* conversion character, allows for the support of multi-byte coded character sets (although these are not mandated by X/Open), and fills in a number of gaps in the definition (for example, by defining termination conditions for *sscanf*( ).
- Following an ANSI interpretation, the effect of conversion specifications that consume no input is better defined, and is no longer marked as an extension.

The following change is incorporated for alignment with the MSE working draft.

- The *C* and *S* conversion characters are added, indicating a pointer in the argument list to the initial wide-character code of an array large enough to accept the input sequence.

Issue 5

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the *l* (ell) qualifier is now defined for the *c*, *s*, and ′[′ conversion characters.

The DESCRIPTION is updated to indicate that if infinity and NaN can be generated by the *fprintf*( ) family of functions, then they are recognized by the *fscanf*( ) family.

Issue 6

The Open Group corrigenda items U021/7 and U028/10 have been applied. These correct several occurrences of ‘‘characters’’ in the text which have been replaced with the term ‘‘bytes’’.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

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

- The prototypes for *fscanf*( ), *scanf*( ), and *sscanf*( ) are updated.
- The DESCRIPTION is updated.

14603 **NAME**

14604 fseek, fseeko — reposition a file-position indicator in a stream

14605 **SYNOPSIS**

14606 ```
#include <stdio.h>
```

14607 ```
int fseek(FILE *stream, long offset, int whence);
```

14608 CX ```
int fseeko(FILE *stream, off_t offset, int whence);
```

14609

14610 **DESCRIPTION**

14611 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14612 conflict between the requirements described here and the ISO C standard is unintentional. This
14613 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

14614 The *fseek*( ) function shall set the file-position indicator for the stream pointed to by *stream*. If a
14615 read or write error occurs, the error indicator for the stream shall be set and *fseek*( ) fails.

14616 The new position, measured in bytes from the beginning of the file, shall be obtained by adding
14617 *offset* to the position specified by *whence*. The specified point is the beginning of the file for
14618 {SEEK_SET}, the current value of the file-position indicator for {SEEK_CUR}, or end-of-file for
14619 {SEEK_END}.

14620 If the stream is to be used with wide-character input/output functions, the application shall
14621 ensure that *offset* is either 0 or a value returned by an earlier call to *ftell*( ) on the same stream and
14622 *whence* is {SEEK_SET}.

14623 A successful call to *fseek*( ) shall clear the end-of-file indicator for the stream and undo any effects
14624 of *ungetc*( ) and *ungetwc*( ) on the same stream. After an *fseek*( ) call, the next operation on an
14625 update stream may be either input or output.

14626 If the most recent operation, other than *ftell*( ), on a given stream is *fflush*( ), the file offset in the
14627 underlying open file description shall be adjusted to reflect the location specified by *fseek*( ).

14628 The *fseek*( ) function shall allow the file-position indicator to be set beyond the end of existing
14629 data in the file. If data is later written at this point, subsequent reads of data in the gap shall
14630 return bytes with the value 0 until data is actually written into the gap.

14631 CX The behavior of *fseek*( ) on devices which are incapable of seeking is implementation-defined.
14632 The value of the file offset associated with such a device is undefined.

14633 If the stream is writable and buffered data had not been written to the underlying file, *fseek*( )
14634 shall cause the unwritten data to be written to the file and shall mark the *st_ctime* and *st_mtime*
14635 fields of the file for update.

14636 In a locale with state-dependent encoding, whether *fseek*( ) restores the stream's shift state is
14637 implementation-defined.

14638 The *fseeko*( ) function is equivalent to the *fseek*( ) function except that the *offset* argument is of
14639 type **off_t**.

14640 **RETURN VALUE**

14641 CX The *fseek*( ) and *fseeko*( ) functions shall return 0 if they succeed.

14642 CX Otherwise, they shall return –1 and set *errno* to indicate the error.

14643 **ERRORS**

14644 CX The *fseek*( ) and *fseeko*( ) functions shall fail if, either the *stream* is unbuffered or the *stream*'s
14645 CX buffer needed to be flushed, and the call to *fseek*( ) or *fseeko*( ) causes an underlying *lseek*( ) or
14646 *write*( ) to be invoked:
```

14647 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the process would be delayed in the write operation.

14649 CX [EBADF] The file descriptor underlying the stream file is not open for writing or the stream's buffer needed to be flushed and the file is not open.

14651 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size.

14652 XSI [EFBIG] An attempt was made to write a file that exceeds the process' file size limit.

14653 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.

14655 CX [EINTR] The write operation was terminated due to the receipt of a signal, and no data was transferred.

14657 CX [EINVAL] The *whence* argument is invalid. The resulting file-position indicator would be set to a negative value.

14659 CX [EIO] A physical I/O error has occurred, or the process is a member of a background process group attempting to perform a *write*() to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.

14664 CX [ENOSPC] There was no free space remaining on the device containing the file.

14665 CX [EOVERFLOW] For *fseek*(), the resulting file offset would be a value which cannot be represented correctly in an object of type **long**.

14667 CX [EOVERFLOW] For *fseeko*(), the resulting file offset would be a value which cannot be represented correctly in an object of type **off_t**.

14669 CX [EPIPE] The file descriptor underlying *stream* is associated with a pipe or FIFO.

14670 CX [EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading by any process; a SIGPIPE signal shall also be sent to the thread.

14672 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

14674 **EXAMPLES**

14675 None.

14676 **APPLICATION USAGE**

14677 None.

14678 **RATIONALE**

14679 None.

14680 **FUTURE DIRECTIONS**

14681 None.

14682 **SEE ALSO**

14683 *fopen*(), *fsetpos*(), *ftell*(), *getrlimit*(), *rewind*(), *ulimit*(), *ungetc*(), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**

14685 **CHANGE HISTORY**

14686 First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 4

In the DESCRIPTION, the words ‘‘The *seek*( ) function does not, by itself, extend the size of a file’’ are deleted.

In the RETURN VALUE section, the value –1 is marked as an extension. This is because the ISO POSIX-1 standard only requires that a non-zero value is returned.

In the ERRORS section, text is added to indicate that error returns are only generated when either the stream is unbuffered, or if the stream buffer needs to be flushed.

The ‘‘fail’’ and ‘‘may fail’’ parts of the ERRORS section are revised for consistency with *lseek*( ) and *write*( ).

Text associated with the [EIO] error is expanded and the [ENXIO] error is added.

Text is added to explain how *fseek*( ) is used with wide-character input/output; this is marked as a WP extension.

The [EFBIG] error is marked to show extensions.

The APPLICATION USAGE section is added.

The following change is incorporated for alignment with the ISO C standard:

- The type of argument *offset* is now defined in full as **long** instead of **long**.

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.

Issue 4, Version 2

In the ERRORS section, the description of [EIO] is updated to include the case where a physical I/O error occurs.

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 now 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 DESCRIPTION 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.

14726 **NAME**

14727 fsetpos — set current file position

14728 **SYNOPSIS**

14729 `#include <stdio.h>`

14730 `int fsetpos(FILE *stream, const fpos_t *pos);`

14731 **DESCRIPTION**

14732 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14733 conflict between the requirements described here and the ISO C standard is unintentional. This
14734 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

14735 The *fsetpos*( ) function shall set the file position and state indicators for the stream pointed to by
14736 *stream* according to the value of the object pointed to by *pos*, which the application shall ensure
14737 is a value obtained from an earlier call to *fgetpos*( ) on the same stream. If a read or write error |
14738 occurs, the error indicator for the stream shall be set and *fsetpos*( ) fails. |

14739 A successful call to the *fsetpos*( ) function shall clear the end-of-file indicator for the stream and
14740 undo any effects of *ungetc*( ) on the same stream. After an *fsetpos*( ) call, the next operation on an
14741 update stream may be either input or output. |

14742 CX The behavior of *fsetpos*( ) on devices which are incapable of seeking is implementation-defined. |
14743 The value of the file offset associated with such a device is undefined. |

14744 **RETURN VALUE**

14745 The *fsetpos*( ) function shall return 0 if it succeeds; otherwise, it shall return a non-zero value and
14746 set *errno* to indicate the error.

14747 **ERRORS**

14748 The *fsetpos*( ) function may fail if:

14749 CX [EBADF] The file descriptor underlying *stream* is not valid. |

14750 CX [ESPIPE] The file descriptor underlying *stream* is associated with a pipe, FIFO, or socket. |

14751

14752 **EXAMPLES**

14753 None.

14754 **APPLICATION USAGE**

14755 None.

14756 **RATIONALE**

14757 None.

14758 **FUTURE DIRECTIONS**

14759 None.

14760 **SEE ALSO**

14761 *fopen*( ), *ftell*( ), *rewind*( ), *ungetc*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
14762 **<stdio.h>**

14763 **CHANGE HISTORY**

14764 First released in Issue 4. Derived from the ISO C standard. |

14765 **Issue 6**

14766 Extensions beyond the ISO C standard are now marked.

14767 An additional [ESPIPE] error condition is added for sockets.

14768 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements. |

14769 The DESCRIPTION is updated to clarify that the error indicator is set for the stream on a read or |
14770 write error. This is for alignment with the ISO/IEC 9899:1999 standard. |

14771 **NAME**

14772 fstat — get file status

14773 **SYNOPSIS**

14774 `#include <sys/stat.h>`

14775 `int fstat(int fildes, struct stat *buf);`

14776 **DESCRIPTION**

14777 The *fstat*( ) function obtains information about an open file associated with the file descriptor
14778 *fildes*, and writes it to the area pointed to by *buf*.

14779 SHM If *fildes* references a shared memory object, the implementation need update in the **stat** structure
14780 pointed to by the *buf* argument only the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the
14781 S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be
14782 valid.

14783 TYM If *fildes* references a typed memory object, the implementation need update in the **stat** structure
14784 pointed to by the *buf* argument only the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the
14785 S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be
14786 valid.

14787 The *buf* argument is a pointer to a **stat** structure, as defined in **<sys/stat.h>**, into which
14788 information is placed concerning the file.

14789 The structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime*, and *st_mtime*
14790 shall have meaningful values for all file types defined in this volume of IEEE Std. 1003.1-200x.
14791 The value of the member *st_nlink* shall be set to the number of links to the file.

14792 An implementation that provides additional or alternative file access control mechanisms may, |
14793 under implementation-defined conditions, cause *fstat*( ) to fail. |

14794 The *fstat*( ) function updates any time-related fields as described in **File Times Update** (see the |
14795 Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 6, Character Set), before writing into |
14796 the **stat** structure. |

14797 **RETURN VALUE**

14798 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
14799 indicate the error.

14800 **ERRORS**

14801 The *fstat*( ) function shall fail if:

14802 [EBADF] The *fildes* argument is not a valid file descriptor. |

14803 [EIO] An I/O error occurred while reading from the file system. |

14804 [EOVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file |
14805 serial number cannot be represented correctly in the structure pointed to by
14806 *buf*. |

14807 The *fstat*( ) function may fail if:

14808 [EOVERFLOW] One of the values is too large to store into the structure pointed to by the *buf* |
14809 argument. |

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. The **/home/cnd/mod1** file is opened with read/write privileges and is passed to the open file descriptor *fildes*.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

struct stat buffer;
int         status;
...
fildes = open("/home/cnd/mod1", O_RDWR);
status = fstat(fildes, &buffer);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

lstat( ), *stat*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

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

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

The following changes are incorporated in the DESCRIPTION for alignment with the ISO POSIX-1 standard:

- A paragraph defining the contents of **stat** structure members is added.
- The words ‘‘extended security controls’’ are replaced by ‘‘additional or alternative file access control mechanisms’’.

Issue 4, Version 2

The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- The [EIO] error is added as a mandatory error indicated the occurrence of an I/O error.
- The [EOVERFLOW] error is added as an optional error indicating that one of the values is too large to store in the area pointed to by *buf*.

Issue 5

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

Large File Summit extensions are added.

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.
- The [EIO] mandatory error condition is added.
- The [EOVERFLOW] mandatory error condition is added. This change is to support large files.
- The [EOVERFLOW] optional error condition is added.

The DESCRIPTION is updated for alignment with IEEE Std. 1003.1j-2000 by specifying that shared memory object semantics apply to typed memory objects.

14863 **NAME**

14864 fstatvfs, statvfs — get file system information

14865 **SYNOPSIS**

14866 XSI
```
#include <sys/statvfs.h>

int fstatvfs(int fildes, struct statvfs *buf);
int statvfs(const char *restrict path, struct statvfs *restrict buf);
```

14869

14870 **DESCRIPTION**

14871 The *fstatvfs*( ) function obtains information about the file system containing the file referenced by
14872 *fildes*.

14873 The following flags can be returned in the *f_flag* member:

14874 ST_RDONLY Read-only file system.

14875 ST_NOSUID Setuid/setgid bits ignored by *exec*.

14876 The *statvfs*( ) function obtains descriptive information about the file system containing the file
14877 named by *path*.

14878 For both functions, the *buf* argument is a pointer to a **statvfs** structure that shall be filled. Read,
14879 write, or execute permission of the named file is not required.

14880 It is unspecified whether all members of the **statvfs** structure have meaningful values on all file
14881 systems.

14882 **RETURN VALUE**

14883 Upon successful completion, *statvfs*( ) shall return 0. Otherwise, it shall return –1 and set *errno* to
14884 indicate the error.

14885 **ERRORS**

14886 The *fstatvfs*( ) and *statvfs*( ) functions shall fail if:

14887 [EIO] An I/O error occurred while reading the file system.

14888 [EINTR] A signal was caught during execution of the function.

14889 [EOVERFLOW] One of the values to be returned cannot be represented correctly in the
14890 structure pointed to by *buf*.

14891 The *fstatvfs*( ) function shall fail if:

14892 [EBADF] The *fildes* argument is not an open file descriptor.

14893 The *statvfs*( ) function shall fail if:

14894 [EACCES] Search permission is denied on a component of the path prefix.

14895 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
14896 argument.

14897 [ENAMETOOLONG]
14898 The length of a path name exceeds {PATH_MAX} or a path name component
14899 is longer than {NAME_MAX}.

14900 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

14901 [ENOTDIR] A component of the path prefix of *path* is not a directory.

14902 The *statvfs*( ) function may fail if:

14903 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during |
14904 resolution of the *path* argument. |

14905 [ENAMETOOLONG] |
14906 Path name resolution of a symbolic link produced an intermediate result
14907 whose length exceeds {PATH_MAX}.

14908 **EXAMPLES**

14909 **Obtaining File System Information Using fstatvfs()**

14910 The following example shows how to obtain file system information for the file system upon
14911 which the file named **/home/cnd/mod1** resides, using the *fstatvfs*() function. The
14912 **/home/cnd/mod1** file is opened with read/write privileges and the open file descriptor is passed
14913 to the *fstatvfs*() function.

```
14914 #include <statvfs.h>
14915 #include <fcntl.h>

14916 struct statvfs buffer;
14917 int            status;
14918 ...
14919 fildes = open("/home/cnd/mod1", O_RDWR);
14920 status  = fstatvfs(fildes, &buffer);
```

14921 **Obtaining File System Information Using statvfs()**

14922 The following example shows how to obtain file system information for the file system upon
14923 which the file named **/home/cnd/mod1** resides, using the *statvfs*() function.

```
14924 #include <statvfs.h>

14925 struct statvfs buffer;
14926 int            status;
14927 ...
14928 status = statvfs("/home/cnd/mod1", &buffer);
```

14929 **APPLICATION USAGE**
14930 None.

14931 **RATIONALE**
14932 None.

14933 **FUTURE DIRECTIONS**
14934 None.

14935 **SEE ALSO**
14936 *chmod*(), *chown*(), *creat*(), *dup*(), *exec*, *fcntl*(), *link*(), *mknod*(), *open*(), *pipe*(), *read*(), *time*(),
14937 *unlink*(), *utime*(), *write*(), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/statvfs.h>** |

14938 **CHANGE HISTORY**
14939 First released in Issue 4, Version 2.

14940 **Issue 5**
14941 Moved from X/OPEN UNIX extension to BASE.

14942 Large File Summit extensions are added.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *statvfs*() prototype for alignment with the ISO/IEC 9899:1999 standard.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

14949 **NAME**

14950 fsync — synchronize changes to a file

14951 **SYNOPSIS**

14952 FSC
```
#include <unistd.h>
```

14953
```
int fsync(int fildes);
```

14954

14955 **DESCRIPTION**

14956 The *fsync*( ) function can be used by an application to indicate that all data for the open file
14957 description named by *fildes* is to be transferred to the storage device associated with the file
14958 described by *fildes* in an implementation-defined manner. The *fsync*( ) function shall not return |
14959 until the system has completed that action or until an error is detected.

14960 SIO If _POSIX_SYNCHRONIZED_IO is defined, the *fsync*( ) function shall force all currently queued
14961 I/O operations associated with the file indicated by file descriptor *fildes* to the synchronized I/O
14962 completion state. All I/O operations shall be completed as defined for synchronized I/O file
14963 integrity completion.

14964 **RETURN VALUE**

14965 Upon successful completion, *fsync*( ) shall return 0. Otherwise, –1 shall be returned and *errno* set
14966 to indicate the error. If the *fsync*( ) function fails, outstanding I/O operations are not guaranteed
14967 to have been completed.

14968 **ERRORS**

14969 The *fsync*( ) function shall fail if:

14970 [EBADF] The *fildes* argument is not a valid descriptor. |

14971 [EINTR] The *fsync*( ) function was interrupted by a signal. |

14972 [EINVAL] The *fildes* argument does not refer to a file on which this operation is possible. |

14973 [EIO] An I/O error occurred while reading from or writing to the file system. |

14974 In the event that any of the queued I/O operations fail, *fsync*( ) shall return the error conditions
14975 defined for *read*( ) and *write*( ).

14976 **EXAMPLES**

14977 None.

14978 **APPLICATION USAGE**

14979 The *fsync*( ) function should be used by programs which require modifications to a file to be
14980 completed before continuing; for example, a program which contains a simple transaction
14981 facility might use it to ensure that all modifications to a file or files caused by a transaction are
14982 recorded.

14983 **RATIONALE**

14984 The *fsync*( ) function is intended to force a physical write of data from the buffer cache, and to
14985 assure that after a system crash or other failure that all data up to the time of the *fsync*( ) call is
14986 recorded on the disk. Since the concepts of ‘‘buffer cache’’, ‘‘system crash’’, ‘‘physical write’’, and
14987 ‘‘non-volatile storage’’ are not defined here, the wording has to be more abstract.

14988 If _POSIX_SYNCHRONIZED_IO is not defined, the wording relies heavily on the conformance
14989 document to tell the user what can be expected from the system. It is explicitly intended that a
14990 null implementation is permitted. This could be valid in the case where the system cannot assure
14991 non-volatile storage under any circumstances or when the system is highly fault-tolerant and the
14992 functionality is not required. In the middle ground between these extremes, *fsync*( ) might or
14993 might not actually cause data to be written where it is safe from a power failure. The

conformance document should identify at least that one configuration exists (and how to obtain that configuration) where this can be assured for at least some files that the user can select to use for critical data. It is not intended that an exhaustive list is required, but rather sufficient information is provided to let the user determine that if he or she has critical data he or she can configure her system to allow it to be written to non-volatile storage.

It is reasonable to assert that the key aspects of *fsync*( ) are unreasonable to test in a test suite. That does not make the function any less valuable, just more difficult to test. A formal conformance test should probably force a system crash (power shutdown) during the test for this condition, but it needs to be done in such a way that automated testing does not require this to be done except when a formal record of the results is being made. It would also not be unreasonable to omit testing for *fsync*( ), allowing it to be treated as a quality-of-implementation issue.

FUTURE DIRECTIONS

None.

SEE ALSO

sync( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 4

The **<unistd.h>** header is added to the SYNOPSIS section.

In the APPLICATION USAGE section, the words ‘‘require a file to be in a known state’’ are replaced by ‘‘require modifications to a file to be completed before continuing’’.

Issue 5

Aligned with *fsync*( ) in the POSIX Realtime Extension. Specifically, the DESCRIPTION and RETURN VALUE sections are much expanded, and the ERRORS section is updated to indicate that *fsync*( ) can return the error conditions defined for *read*( ) and *write*( ).

Issue 6

This function is marked as part of the File Synchronization option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EINVAL] and [EIO] mandatory error conditions are added.

15025 **NAME**

15026 ftell, ftello — return a file offset in a stream

15027 **SYNOPSIS**

15028 `#include <stdio.h>`

15029 `long ftell(FILE *stream);` |

15030 CX `off_t ftello(FILE *stream);` |

15031

15032 **DESCRIPTION**

15033 CX The functionality described on this reference page is aligned with the ISO C standard. Any
15034 conflict between the requirements described here and the ISO C standard is unintentional. This
15035 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

15036 The *ftell*( ) function shall obtain the current value of the file-position indicator for the stream
15037 pointed to by *stream*.

15038 CX The *ftello*( ) function is identical to *ftell*( ) except that the return value is of type **off_t**.

15039 **RETURN VALUE**

15040 CX Upon successful completion, *ftell*( ) and *ftello*( ) shall return the current value of the file-position
15041 indicator for the stream measured in bytes from the beginning of the file.

15042 CX Otherwise, *ftell*( ) and *ftello*( ) shall return –1, cast to **long** and **off_t** respectively, and set *errno* to
15043 indicate the error.

15044 **ERRORS**

15045 CX The *ftell*( ) and *ftello*( ) functions shall fail if:

15046 CX [EBADF] The file descriptor underlying *stream* is not an open file descriptor. |

15047 CX [EOVERFLOW] For *ftell*( ), the current file offset cannot be represented correctly in an object of |
15048 type **long**.

15049 CX [EOVERFLOW] For *ftello*( ), the current file offset cannot be represented correctly in an object |
15050 of type **off_t**.

15051 CX [ESPIPE] The file descriptor underlying *stream* is associated with a pipe or FIFO. |

15052 The *ftell*( ) function may fail if:

15053 CX [ESPIPE] The file descriptor underlying *stream* is associated with a socket.

15054 **EXAMPLES**

15055 None.

15056 **APPLICATION USAGE**

15057 None.

15058 **RATIONALE**

15059 None.

15060 **FUTURE DIRECTIONS**

15061 None.

15062 **SEE ALSO**

15063 *fgetpos*( ), *fopen*( ), *fseek*( ), *lseek*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>** |

15064 **CHANGE HISTORY**

15065 First released in Issue 1. Derived from Issue 1 of the SVID.

15066 **Issue 4**

15067 The following change is incorporated for alignment with the ISO C standard:

15068 • The function return value is now defined in full as **long**. It was previously defined as **long**.

15069 **Issue 5**

15070 Large File Summit extensions are added.

15071 **Issue 6**

15072 Extensions beyond the ISO C standard are now marked.

15073 The following new requirements on POSIX implementations derive from alignment with the
15074 Single UNIX Specification:

15075 • The *ftello*( ) function is added.

15076 • The [EOVERFLOW] error conditions are added.

15077 An additional [ESPIPE] error condition is added for sockets.

15078 **NAME**

15079 ftime — get date and time (**LEGACY**)

15080 **SYNOPSIS**

15081 XSI
```
#include <sys/timeb.h>
```
15082
```
int ftime(struct timeb *tp);
```
15083

15084 **DESCRIPTION**

15085 The *ftime*( ) function shall set the *time* and *millitm* members of the **timeb** structure pointed to by
15086 *tp* to contain the seconds and milliseconds portions, respectively, of the current time in seconds
15087 since the Epoch. The contents of the *timezone* and *dstflag* members of *tp* after a call to *ftime*( ) are
15088 unspecified.

15089 The system clock need not have millisecond granularity. Depending on any granularity
15090 (particularly a granularity of one) renders code non-portable.

15091 **RETURN VALUE**

15092 Upon successful completion, the *ftime*( ) function shall return 0; otherwise, –1 shall be returned.

15093 **ERRORS**

15094 No errors are defined.

15095 **EXAMPLES**

15096 **Getting the Current Time and Date**

15097 The following example shows how to get the current system time values using the *ftime*( )
15098 function. The **timeb** structure pointed to by *tp* is filled with the current system time values for
15099 *time* and *millitm*.

15100
```
#include <sys/timeb.h>
```
15101
```
struct timeb tp;
```
15102
```
int          status;
```
15103
```
...
```
15104
```
status = ftime(&tp);
```

15105 **APPLICATION USAGE**

15106 For applications portability, the *time*( ) function should be used to determine the current time
15107 instead of *ftime*( ).

15108 **RATIONALE**

15109 None.

15110 **FUTURE DIRECTIONS**

15111 This function may be withdrawn in a future version.

15112 **SEE ALSO**

15113 *ctime*( ), *gettimeofday*( ), *time*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
15114 **<sys/timeb.h>**

15115 **CHANGE HISTORY**

15116 First released in Issue 4, Version 2.

15117 **Issue 5**

15118 Moved from X/OPEN UNIX extension to BASE.

15119 Normative text previously in the APPLICATION USAGE section is moved to the
15120 DESCRIPTION.

Issue 6

This function is marked LEGACY.

The DESCRIPTION is updated to refer to ‘‘seconds since the Epoch’’ rather than ‘‘seconds since 00:00:00 UTC (Coordinated Universal Time), January 1 1970’’ for consistency with other *time* functions.

15126 **NAME**

15127 ftok — generate an IPC key

15128 **SYNOPSIS**

15129 XSI
```
#include <sys/ipc.h>
```

15130
```
key_t ftok(const char *path, int id);
```

15131

15132 **DESCRIPTION**

15133 The *ftok*( ) function shall return a key based on *path* and *id* that is usable in subsequent calls to
15134 *msgget*( ), *semget*( ), and *shmget*( ). The application shall ensure that the *path* argument is the path
15135 name of an existing file that the process is able to *stat*( ).

15136 The *ftok*( ) function shall return the same key value for all paths that name the same file, when
15137 called with the same *id* value, and return different key values when called with different *id*
15138 values or with paths that name different files existing on the same file system at the same time. It
15139 is unspecified whether *ftok*( ) shall return the same key value when called again after the file
15140 named by *path* is removed and recreated with the same name.

15141 Only the low order 8-bits of *id* are significant. The behavior of *ftok*( ) is unspecified if these bits
15142 are 0.

15143 **RETURN VALUE**

15144 Upon successful completion, *ftok*( ) shall return a key. Otherwise, *ftok*( ) shall return (**key_t**)–1
15145 and set *errno* to indicate the error.

15146 **ERRORS**

15147 The *ftok*( ) function shall fail if:

15148 [EACCES] Search permission is denied for a component of the path prefix. |

15149 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* |
15150 argument. |

15151 [ENAMETOOLONG] |
15152 The length of the *path* argument exceeds {PATH_MAX} or a path name
15153 component is longer than {NAME_MAX}.

15154 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string. |

15155 [ENOTDIR] A component of the path prefix is not a directory. |

15156 The *ftok*( ) function may fail if:

15157 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during |
15158 resolution of the *path* argument. |

15159 [ENAMETOOLONG] |
15160 Path name resolution of a symbolic link produced an intermediate result
15161 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 path name **/tmp**.

```
#include <sys/ipc.h>
...
key_t key;
char *path = "/tmp";
int id = 'S';

key = ftok(path, id);
```

Saving an IPC Key

The following example gets a unique key based on the path name **/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*( ).

```
#include <sys/ipc.h>
...
key_t semkey;

if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
    perror("IPC error: ftok"); exit(1);
}
```

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 DESCRIPTION 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.

15199 **NAME**

15200 ftruncate — truncate a file to a specified length

15201 **SYNOPSIS**

15202 `#include <unistd.h>`

15203 `int ftruncate(int` *`fildes`*`, off_t` *`length`*`);`

15204 **DESCRIPTION**

15205 If *fildes* is not a valid file descriptor open for writing, the *ftruncate*( ) function shall fail.

15206 If *fildes* refers to a regular file, the *ftruncate*( ) function shall cause the size of the file to be
15207 truncated to *length*. If the size of the file previously exceeded *length*, the extra data shall no
15208 longer be available to reads on the file. If the file previously was smaller than this size,
15209 XSI *ftruncate*( ) shall either increase the size of the file or fail. XSI-conformant systems shall increase
15210 the size of the file. If the file size is increased, the extended area shall appear as if it were zero-
15211 filled. The value of the seek pointer shall not be modified by a call to *ftruncate*( ).

15212 Upon successful completion, if *fildes* refers to a regular file, the *ftruncate*( ) function shall mark
15213 for update the *st_ctime* and *st_mtime* fields of the file and the S_ISUID and S_ISGID bits of the file
15214 mode may be cleared. If the *ftruncate*( ) function is unsuccessful, the file is unaffected.

15215 XSI If the request would cause the file size to exceed the soft file size limit for the process, the
15216 request shall fail and the implementation shall generate the SIGXFSZ signal for the process.

15217 If *fildes* refers to a directory, *ftruncate*( ) shall fail.

15218 If *fildes* refers to any other file type, except a shared memory object, the result is unspecified.

15219 SHM If *fildes* refers to a shared memory object, *ftruncate*( ) shall set the size of the shared memory
15220 object to *length*.

15221 MF|SHM If the effect of *ftruncate*( ) is to decrease the size of a shared memory object or memory mapped
15222 file and whole pages beyond the new end were previously mapped, then the whole pages
15223 beyond the new end shall be discarded.

15224 MPR If the Memory Protection option is supported, references to discarded pages shall result in the
15225 generation of a SIGBUS signal; otherwise, the result of such references is undefined.

15226 MF|SHM If the effect of *ftruncate*( ) is to increase the size of a shared memory object, it is unspecified if the
15227 contents of any mapped pages between the old end-of-file and the new are flushed to the
15228 underlying object.

15229 **RETURN VALUE**

15230 Upon successful completion, *ftruncate*( ) shall return 0; otherwise, –1 shall be returned and *errno*
15231 set to indicate the error.

15232 **ERRORS**

15233 The *ftruncate*( ) function shall fail if:

15234 [EINTR] A signal was caught during execution.

15235 [EINVAL] The *length* argument was less than 0.

15236 [EFBIG] or [EINVAL]
15237 The *length* argument was greater than the maximum file size.

15238 XSI [EFBIG] The file is a regular file and *length* is greater than the offset maximum
15239 established in the open file description associated with *fildes*.

15240 [EIO] An I/O error occurred while reading from or writing to a file system.

15241 [EBADF] or [EINVAL] |
15242 The *fildes* argument is not a file descriptor open for writing.

15243 [EINVAL] The *fildes* argument references a file that was opened without write |
15244 permission.

15245 [EROFS] The named file resides on a read-only file system. |

15246 **EXAMPLES**

15247 None.

15248 **APPLICATION USAGE**

15249 None.

15250 **RATIONALE**

15251 The *ftruncate*( ) function is part of IEEE Std. 1003.1-200x as it was deemed to be more useful than |
15252 *truncate*( ). The *truncate*( ) function is provided as an XSI extension.

15253 **FUTURE DIRECTIONS**

15254 None.

15255 **SEE ALSO**

15256 *open*( ), *truncate*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>** |

15257 **CHANGE HISTORY**

15258 First released in Issue 4, Version 2.

15259 **Issue 5**

15260 Moved from X/OPEN UNIX extension to BASE and aligned with *ftruncate*( ) in the POSIX
15261 Realtime Extension. Specifically, the DESCRIPTION is extensively reworded and [EROFS] is
15262 added to the list of mandatory errors that can be returned by *ftruncate*( ).

15263 Large File Summit extensions are added.

15264 **Issue 6**

15265 The *truncate*( ) function has been split out into a separate reference page.

15266 The following new requirements on POSIX implementations derive from alignment with the |
15267 Single UNIX Specification:

15268 • The DESCRIPTION is change to indicate that if the file size is changed, and if the file is a
15269 regular file, the S_ISUID and S_ISGID bits in the file mode may be cleared.

15270 The following changes were made to align with the IEEE P1003.1a draft standard:

15271 • The DESCRIPTION text is updated.

15272 XSI-conformant systems are required to increase the size of the file if the file was previously |
15273 smaller than the size requested. |

15274 **NAME**

15275 ftrylockfile — stdio locking functions

15276 **SYNOPSIS**

15277 TSF
```
#include <stdio.h>
```

15278
```
int ftrylockfile(FILE *file);
```

15279

15280 **DESCRIPTION**

15281 Refer to *flockfile* ( ).

NAME

ftw — traverse (walk) a file tree

SYNOPSIS

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 recursively descends 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, and an integer. 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 of 1 to {OPEN_MAX}.

The tree traversal shall continue until 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.

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 value that is possible when a directory is opened or when one of the *stat* functions is executed on a directory or file.

15323 **ERRORS**

15324 The *ftw*( ) function shall fail if:

15325 [EACCES] Search permission is denied for any component of *path* or read permission is
15326 denied for *path*.

15327 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
15328 argument.

15329 [ENAMETOOLONG]
15330 The length of the *path* argument exceeds {PATH_MAX} or a path name
15331 component is longer than {NAME_MAX}.

15332 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

15333 [ENOTDIR] A component of *path* is not a directory.

15334 The *ftw*( ) function may fail if:

15335 [EINVAL] The value of the *ndirs* argument is invalid.

15336 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
15337 resolution of the *path* argument.

15338 [ENAMETOOLONG]
15339 Path name resolution of a symbolic link produced an intermediate result
15340 whose length exceeds {PATH_MAX}.

15341 In addition, if the function pointed to by *fn* encounters system errors, *errno* may be set
15342 accordingly.

15343 **EXAMPLES**

15344 **Walking a Directory Structure**

15345 The following example walks the current directory structure, calling the *fn* function for every
15346 directory entry, using at most 10 file descriptors:

```
15347 #include <ftw.h>
15348 ...
15349 if (ftw(".", fn, 10) != 0) {
15350     perror("ftw"); exit(2);
15351 }
```

15352 **APPLICATION USAGE**

15353 The *ftw*( ) function may allocate dynamic storage during its operation. If *ftw*( ) is forcibly
15354 terminated, such as by *longjmp*( ) or *siglongjmp*( ) being executed by the function pointed to by *fn*
15355 or an interrupt routine, *ftw*( ) does not have a chance to free that storage, so it remains
15356 permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has
15357 occurred, and arrange to have the function pointed to by *fn* return a non-zero value at its next
15358 invocation.

15359 **RATIONALE**

15360 None.

15361 **FUTURE DIRECTIONS**

15362 None.

15363 **SEE ALSO**

15364 *longjmp*( ), *lstat*( ), *malloc*( ), *nftw*( ), *opendir*( ), *siglongjmp*( ), *stat*( ), the Base Definitions volume of |
15365 IEEE Std. 1003.1-200x, **<ftw.h>**, **<sys/stat.h>** |

15366 **CHANGE HISTORY**

15367 First released in Issue 1. Derived from Issue 1 of the SVID. |

15368 **Issue 4**

15369 The type of argument *path* is changed from **char*** to **const char***. The argument list for *fn* has
15370 also been defined.

15371 In the DESCRIPTION, the words ‘‘other than [EACCES]’’ are added to the paragraph describing
15372 termination conditions for tree traversal.

15373 The following change is incorporated for alignment with the FIPS requirements:

15374 • In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path
15375 name component is larger that {NAME_MAX} is now defined as mandatory and marked as
15376 an extension.

15377 **Issue 4, Version 2**

15378 The following changes are incorporated for X/OPEN UNIX conformance:

15379 • The DESCRIPTION is updated to describe the use of the FTW_SL and FTW_NS values for a
15380 symbolic link.

15381 • The DESCRIPTION states that *ftw*( ) uses at most one file descriptor for each level in the tree.

15382 • The DESCRIPTION constrains *ndirs* to the range from 1 to {OPEN_MAX}.

15383 • The RETURN VALUE section is updated to describe the case where *ftw*( ) encounters an error
15384 other than [EACCES].

15385 • In the ERRORS section, a second [ENAMETOOLONG] condition is defined that may report
15386 excessive length of an intermediate result of path name resolution of a symbolic link.

15387 **Issue 5**

15388 UX codings in the DESCRIPTION, RETURN VALUE, and ERRORS sections have been changed
15389 to EX.

15390 **Issue 6**

15391 The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

15392 • The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC.
15393 This is since behavior may vary from one file system to another.

15394 The wording of the mandatory [ELOOP] error condition is updated, and a second optional |
15395 [ELOOP] error condition is added. |

15396 **NAME**

15397 funlockfile — stdio locking functions

15398 **SYNOPSIS**

15399 TSF
```
#include <stdio.h>
```

15400
```
void funlockfile(FILE *file);
```

15401

15402 **DESCRIPTION**

15403 Refer to *flockfile* ( ).

15404 **NAME**

15405 fwide — set stream orientation

15406 **SYNOPSIS**

15407 `#include <stdio.h>`

15408 `#include <wchar.h>`

15409 `int fwide(FILE *stream, int mode);`

15410 **DESCRIPTION**

15411 CX The functionality described on this reference page is aligned with the ISO C standard. Any
15412 conflict between the requirements described here and the ISO C standard is unintentional. This
15413 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

15414 The *fwide*( ) function shall determine the orientation of the stream pointed to by *stream*. If *mode* is
15415 greater than zero, the function first attempts to make the stream wide-oriented. If *mode* is less
15416 than zero, the function first attempts to make the stream byte-oriented. Otherwise, *mode* is zero
15417 and the function does not alter the orientation of the stream.

15418 If the orientation of the stream has already been determined, *fwide*( ) shall not change it.

15419 Because no return value is reserved to indicate an error, an application wishing to check for error
15420 situations should set *errno* to 0, then call *fwide*( ), then check *errno*, and if it is non-zero, assume
15421 an error has occurred.

15422 **RETURN VALUE**

15423 The *fwide*( ) function shall return a value greater than zero if, after the call, the stream has wide-
15424 orientation, a value less than zero if the stream has byte-orientation, or zero if the stream has no
15425 orientation.

15426 **ERRORS**

15427 The *fwide*( ) function may fail if:

15428 CX [EBADF] The *stream* argument is not a valid stream.

15429 **EXAMPLES**

15430 None.

15431 **APPLICATION USAGE**

15432 A call to *fwide*( ) with *mode* set to zero can be used to determine the current orientation of a
15433 stream.

15434 **RATIONALE**

15435 None.

15436 **FUTURE DIRECTIONS**

15437 None.

15438 **SEE ALSO**

15439 The Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>**

15440 **CHANGE HISTORY**

15441 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
15442 (E).

15443 **Issue 6**

15444 Extensions beyond the ISO C standard are now marked.

15445 **NAME**

15446 fwprintf, swprintf, wprintf — print formatted wide-character output

15447 **SYNOPSIS**

15448 ```
#include <stdio.h>
```

15449 ```
#include <wchar.h>
```

15450 ```
int fwprintf(FILE *restrict stream, const wchar_t *restrict format, ...);
```

15451 ```
int swprintf(wchar_t *restrict ws, size_t n, const wchar_t *restrict format, ...);
```

15452 ```
int wprintf(const wchar_t *restrict format, ...);
```

15453 **DESCRIPTION**

15454 CX The functionality described on this reference page is aligned with the ISO C standard. Any
15455 conflict between the requirements described here and the ISO C standard is unintentional. This
15456 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

15457 The *fwprintf*( ) function places output on the named output *stream*. The *wprintf*( ) function places
15458 output on the standard output stream *stdout*. The *swprintf*( ) function places output followed by
15459 the null wide character in consecutive wide characters starting at **ws*; no more than *n* wide
15460 characters are written, including a terminating null wide character, which is always added
15461 (unless *n* is zero).

15462 Each of these functions converts, formats, and prints its arguments under control of the *format*
15463 wide-character string. The *format* is composed of zero or more directives: *ordinary wide-*
15464 *characters*, which are simply copied to the output stream, and *conversion specifications*, each of
15465 which results in the fetching of zero or more arguments. The results are undefined if there are
15466 insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the
15467 excess arguments are evaluated but are otherwise ignored.

15468 XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
15469 to the next unused argument. In this case, the conversion wide character '%' (see below) is
15470 replaced by the sequence "%n$", where *n* is a decimal integer in the range [1,{NL_ARGMAX}],
15471 giving the position of the argument in the argument list. This feature provides for the definition
15472 of *format* wide-character strings that select arguments in an order appropriate to specific
15473 languages (see the EXAMPLES section).

15474 In *format* wide-character strings containing the "%n$" form of conversion specifications,
15475 numbered arguments in the argument list can be referenced from the *format* wide-character
15476 string as many times as required.

15477 In *format* wide-character strings containing the '%' form of conversion specifications, each
15478 argument in the argument list is used exactly once.

15479 All forms of the *fwprintf*( ) function allow for the insertion of a locale-dependent radix character
15480 in the output string, output as a wide-character value. The radix character is defined in the
15481 program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix
15482 character is not defined, the radix character defaults to a period ('.').

15483 XSI Each conversion specification is introduced by the '%' wide character or by the wide-character
15484 sequence "%n$", after which the following appear in sequence:

15485 - Zero or more *flags* (in any order), which modify the meaning of the conversion specification.

15486 - An optional minimum *field width*. If the converted value has fewer wide characters than the
15487 field width, it shall be padded with spaces by default on the left; it shall be padded on the
15488 right, if the left-adjustment flag ('−'), described below, is given to the field width. The field
15489 width takes the form of an asterisk ('*'), described below, or a decimal integer.
```

- An optional *precision* that gives the minimum number of digits to appear for the *d*, *i*, *o*, *u*, *x*, and *X* conversions; the number of digits to appear after the radix character for the *e*, *E*, and *f* conversions; the maximum number of significant digits for the *g* and *G* conversions; or the maximum number of wide characters to be printed from a string in *s* conversions. The precision takes the form of a period (`'.'`) followed either by an asterisk (`'*'`), described below, or an optional decimal digit string, where a null digit string is treated as 0. If a precision appears with any other conversion wide character, the behavior is undefined.
- An optional length modifier that specifies the size of the argument.
- A *conversion wide character* that indicates the type of conversion to be applied.

A field width, or precision, or both, may be indicated by an asterisk (`'*'`). In this case an argument of type **int** supplies the field width or precision. The application shall ensure that arguments specifying field width, or precision, or both appear in that order before the argument, if any, to be converted. A negative field width is taken as a `'-'` flag followed by a positive field width. A negative precision is taken as if the precision were omitted. In format wide-character strings containing the `"%n$"` form of a conversion specification, a field width or precision may be indicated by the sequence `"*m$"`, where *m* is a decimal integer in the range [1,{NL_ARGMAX}] giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision, for example:

```
wprintf(L"%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

The *format* can contain either numbered argument specifications (that is, `"%n$"` and `"*m$"`), or unnumbered argument specifications (that is, `'%'` and `'*'`), but normally not both. The only exception to this is that `"%%"` can be mixed with the `"%n$"` form. The results of mixing numbered and unnumbered argument specifications in a *format* wide-character string are undefined. When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the (*N*–1)th, are specified in the format wide-character string.

The flag wide characters and their meanings are:

`'` The integer portion of the result of a decimal conversion (%*i*, %*d*, %*u*, %*f*, %*g*, or %*G*) shall be formatted with thousands' grouping wide characters. For other conversions, the behavior is undefined. The numeric grouping wide character is used.

– The result of the conversion shall be left-justified within the field. The conversion shall be right-justified if this flag is not specified.

+ The result of a signed conversion shall always begin with a sign (`'+'` or `'-'`). The conversion shall begin with a sign only when a negative value is converted if this flag is not specified.

<space> If the first wide character of a signed conversion is not a sign, or if a signed conversion results in no wide 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.

This flag 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 0. For *x* or *X* conversions, a non-zero result ahsll have 0*x* (or 0*X*) prefixed to it. For *e*, *E*, *f*, *g*, or *G* conversions, the result shall always contain a radix character, even if no digits follow it. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For *g* and *G* conversions, trailing zeros shall *not* be removed from the result as they normally are. For other conversions, the behavior is undefined.

0 For *d*, *i*, *o*, *u*, *x*, *X*, *e*, *E*, *f*, *g*, and *G* conversions, 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 shall be ignored. For *d*, *i*, *o*, *u*, *x*, and *X* conversions, if a precision is specified, the 0 flag shall be ignored. If the 0 and ''' flags both appear, the grouping wide characters are inserted before zero padding. For other conversions, the behavior is undefined.

The length modifiers and their meanings are:

hh Specifies that a following *d*, *i*, *o*, *u*, *x*, or *X* conversion specifier applies to a **signed char** or **unsigned char** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **signed char** or **unsigned char** before printing); or that a following *n* conversion specifier applies to a pointer to a **signed char** argument.

h Specifies that a following *d*, *i*, *o*, *u*, *x*, or *X* conversion specifier applies to a **short** or **unsigned short** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **short** or **unsigned short** before printing); or that a following *n* conversion specifier applies to a pointer to a **short** argument.

l (ell) Specifies that a following *d*, *i*, *o*, *u*, *x*, or *X* conversion specifier applies to a **long** or **unsigned long** argument; that a following *n* conversion specifier applies to a pointer to a **long** argument; that a following *c* conversion specifier applies to a **wint_t** argument; that a following *s* conversion specifier applies to a pointer to a **wchar_t** argument; or has no effect on a following *a*, *A*, *e*, *E*, *f*, *F*, *g*, or *G* conversion specifier.

ll (ell-ell) Specifies that a following *d*, *i*, *o*, *u*, *x*, or *X* conversion specifier applies to a **long long** or **unsigned long long** argument; or that a following *n* conversion specifier applies to a pointer to a **long long** argument.

j Specifies that a following *d*, *i*, *o*, *u*, *x*, or *X* conversion specifier applies to an **intmax_t** or **uintmax_t** argument; or that a following *n* conversion specifier applies to a pointer to an **intmax_t** argument.

z Specifies that a following *d*, *i*, *o*, *u*, *x*, or *X* conversion specifier applies to a **size_t** 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 **size_t** argument.

t Specifies that a following *d*, *i*, *o*, *u*, *x*, or *X* conversion specifier applies to a **ptrdiff_t** or the corresponding **unsigned** type argument; or that a following *n* conversion specifier applies to a pointer to a **ptrdiff_t** argument.

L Specifies that a following *a*, *A*, *e*, *E*, *f*, *F*, *g*, or *G* conversion specifier applies to a **long double** argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion wide characters and their meanings are:

d, *i* The **int** argument is converted to a signed decimal in the style **[–]***dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide characters.

o The **unsigned** argument is converted to unsigned octal format in the style *dddd*. The precision specifies the minimum number of digits to appear; if the value being

converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide characters.

u The **unsigned** argument is converted to unsigned decimal format in the style *dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide characters.

x The **unsigned** argument is converted to unsigned hexadecimal format in the style *dddd*; the letters "abcdef" are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide characters.

X Behaves the same as the *x* conversion wide character except that letters "ABCDEF" are used instead of "abcdef".

f, F The **double** argument is converted to decimal notation in the style **[–]***ddd.ddd*, where the number of digits after the radix character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly 0 and no '#' flag is present, no radix character appears. If a radix character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

A **double** argument representing an infinity is converted in one of the styles [–]inf or [–]infinity; which style is implementation-defined. A **double** argument representing a NaN is converted in one of the styles [–]nan or [–]nan(*n-char-sequence*); which style, and the meaning of any *n-char-sequence*, is implementation-defined. The *F* conversion specifier produces INF, INFINITY, or NAN instead of inf, infinity, or nan, respectively.

e, E The **double** argument is converted in the style **[–]***d.ddd*±*dd*, where there is one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is 0 and no '#' flag is present, no radix character appears. The value is rounded to the appropriate number of digits. The *E* conversion wide character shall produce a number with *E* instead of *e* introducing the exponent. The exponent always contains at least two digits. If the value is 0, the exponent is 0.

A **double** argument representing an infinity or NaN is converted in the style of an *f* or *F* conversion specifier.

g, G The **double** argument is converted in the style *f* or *e* (or in the style *E* in the case of a *G* conversion wide character), with the precision specifying the number of significant digits. If an explicit precision is 0, it is taken as 1. The style used depends on the value converted; style *e* (or *E*) shall be used only if the exponent resulting from such a conversion is less than –4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a radix character appears only if it is followed by a digit.

A **double** argument representing an infinity or NaN is converted in the style of an *f* or *F* conversion specifier.

a, A A **double** argument representing a floating-point number is converted in the style [–]0xh.hhhh p1d, where there is one hexadecimal digit (which is non-zero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character 235) and the number of hexadecimal digits after it is equal

15629 to the precision; if the precision is missing and FLT_RADIX is a power of 2, then the
15630 precision is sufficient for an exact representation of the value; if the precision is missing
15631 and FLT_RADIX is not a power of 2, then the precision is sufficient to distinguish 236)
15632 values of type **double**, except that trailing zeros may be omitted; if the precision is zero
15633 and the '#' flag is not specified, no decimal-point character appears. The letters
15634 "abcdef" are used for *a* conversion and the letters "ABCDEF" for *A* conversion. The *A*
15635 conversion specifier produces a number with 'X' and 'P' instead of 'x' and 'p'.
15636 The exponent always contains at least one digit, and only as many more digits as
15637 necessary to represent the decimal exponent of 2. If the value is zero, the exponent is
15638 zero.

15639 A **double** argument representing an infinity or NaN is converted in the style of an *f* or *F*
15640 conversion specifier.

15641 *c* If no *l* (ell) qualifier is present, the **int** argument is converted to a wide character as if by
15642 calling the *btowc*( ) function and the resulting wide character is written. Otherwise, the
15643 **wint_t** argument is converted to **wchar_t**, and written.

15644 *s* If no *l* (ell) qualifier is present, the application shall ensure that the argument is a
15645 pointer to a character array containing a character sequence beginning in the initial
15646 shift state. Characters from the array are converted as if by repeated calls to the
15647 *mbrtowc*( ) function, with the conversion state described by an **mbstate_t** object
15648 initialized to zero before the first character is converted, and written up to (but not
15649 including) the terminating null wide character. If the precision is specified, no more
15650 than that many wide characters are written. If the precision is not specified, or is
15651 greater than the size of the array, the application shall ensure that the array contains a
15652 null wide character.

15653 If an *l* (ell) qualifier is present, the application shall ensure that the argument is a
15654 pointer to an array of type **wchar_t**. Wide characters from the array are written up to
15655 (but not including) a terminating null wide character. If no precision is specified, or is
15656 greater than the size of the array, the application shall ensure that the array contains a
15657 null wide character. If a precision is specified, no more than that many wide characters
15658 are written.

15659 *p* The application shall ensure that the argument is a pointer to **void**. The value of the
15660 pointer is converted to a sequence of printable wide characters, in an implementation-
15661 defined manner.

15662 *n* The application shall ensure that the argument is a pointer to an integer into which is
15663 written the number of wide characters written to the output so far by this call to one of
15664 the *fwprintf*( ) functions. No argument is converted, but one is consumed. If the
15665 conversion specification includes any flags, a field width, or a precision, the behavior is
15666 undefined.

15667 XSI *C* Same as *lc*.

15668 XSI *S* Same as *ls*.

15669 % Output a '%' wide character; no argument is converted. The entire conversion
15670 specification shall be "%%".

15671 If a conversion specification does not match one of the above forms, the behavior is undefined.

15672 In no case does a nonexistent or small field width cause truncation of a field; if the result of a
15673 conversion is wider than the field width, the field is simply expanded to contain the conversion
15674 result. Characters generated by *fwprintf*( ) and *wprintf*( ) are printed as if *fputwc*( ) had been
15675 called.

For *a* and *A* conversions, if FLT_RADIX is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

If FLT_RADIX is not a power of 2, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

For *e*, *E*, *f*, *F*, *g*, and *G* conversions, if the number of significant decimal digits is at most DECIMAL_DIG, then the result should be correctly rounded. If the number of significant decimal digits is more than DECIMAL_DIG but the source value is exactly representable with DECIMAL_DIG digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings "L < U", both having DECIMAL_DIG significant digits; the value of the resultant decimal string "D" should satisfy "L <= D <= U", with the extra stipulation that the error should have a correct sign for the current rounding direction.

CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the call to a successful execution of *fwprintf*( ) or *wprintf*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same stream, or a call to *exit*( ) or *abort*( ).

RETURN VALUE

Upon successful completion, these functions shall return the number of wide characters transmitted, excluding the terminating null wide character in the case of *swprintf*( ), or a negative CX value if an output error was encountered, and set *errno* to indicate the error.

If *n* or more wide characters were requested to be written, *swprintf*( ) shall return a negative CX value, and set *errno* to indicate the error.

ERRORS

For the conditions under which *fwprintf*( ) and *wprintf*( ) fail and may fail, refer to *fputwc*( ).

In addition, all forms of *fwprintf*( ) may fail if:

XSI [EILSEQ] A wide-character code that does not correspond to a valid character has been detected.

XSI [EINVAL] There are insufficient arguments.

In addition, *wprintf*( ) and *fwprintf*( ) may fail if:

XSI [ENOMEM] Insufficient storage space is available.

EXAMPLES

To print the language-independent date and time format, the following statement could be used:

```
wprintf(format, weekday, month, day, hour, min);
```

For American usage, *format* could be a pointer to the wide-character string:

```
L"%s, %s %d, %d:%.2d\n"
```

producing the message:

```
Sunday, July 3, 10:02
```

whereas for German usage, *format* could be a pointer to the wide-character string:

```
L"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

producing the message:

```
Sonntag, 3. Juli, 10:02
```

15717 **APPLICATION USAGE**

15718 None.

15719 **RATIONALE**

15720 None.

15721 **FUTURE DIRECTIONS**

15722 None.

15723 **SEE ALSO**

15724 *btowc*( ), *fputwc*( ), *fwscanf*( ), *mbrtowc*( ), *setlocale*( ), the Base Definitions volume of |
15725 IEEE Std. 1003.1-200x, **<stdio.h>**, **<wchar.h>**, the Base Definitions volume of |
15726 IEEE Std. 1003.1-200x, Chapter 7, Locale |

15727 **CHANGE HISTORY**

15728 First released in Issue 5. Included for alignment with ISO / IEC 9899: 1990 / Amendment 1: 1995
15729 (E).

15730 **Issue 6**

15731 The Open Group corrigenda item U040 / 1 has been applied to the RETURN VALUE section,
15732 describing the case if *n* or more wide characters are requested to be written using *swprintf*( ).

15733 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements. |

15734 The following changes are made for alignment with the ISO / IEC 9899: 1999 standard: |

15735 • The prototypes for *fwprintf*( ), *swprintf*( ), and *wprintf*( ) are updated. |

15736 • The DESCRIPTION is updated. |

NAME

fwrite — binary output

SYNOPSIS

```
#include <stdio.h>

size_t fwrite(const void *restrict ptr, size_t size, size_t nitems,
    FILE *restrict 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 *fwrite*( ) function shall write, from the array pointed to by *ptr*, up to *nitems* members whose size is specified by *size*, to the stream pointed to by *stream*. For each object, size calls are made to the *fputc*( ) function, taking the values (in order) from an array of **unsigned char** exactly overlaying the object. The file-position indicator for the stream (if defined) shall be advanced by the number of bytes successfully written. If an error occurs, the resulting value of the file-position indicator for the stream is indeterminate.

CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful execution of *fwrite*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same stream, or a call to *exit*( ) or *abort*( ).

RETURN VALUE

The *fwrite*( ) function shall return the number of members successfully written, which may be less than *nitems* if a write error is encountered. If *size* or *nitems* is 0, *fwrite*( ) shall return 0 and the state of the stream remains unchanged. Otherwise, if a write error occurs, the error indicator for the stream shall be set, CX and *errno* shall be set to indicate the error.

ERRORS

Refer to *fputc*( ).

EXAMPLES

None.

APPLICATION USAGE

Because of possible differences in member 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

ferror( ), *fopen*( ), *printf*( ), *putc*( ), *puts*( ), *write*( ), 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 4

In the DESCRIPTION, the text is changed to make it clear that the function advances the file-position indicator by the number of bytes successfully written rather than the number of characters, which could include multi-byte sequences.

The following change is incorporated for alignment with the ISO C standard:

- The type of argument *ptr* is changed from **void*** to **const void***.

Issue 6

Extensions beyond the ISO C standard are now 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*( ).

15789 **NAME**

15790 fwscanf, swscanf, wscanf — convert formatted wide-character input

15791 **SYNOPSIS**

15792 ```
#include <stdio.h>
```

15793 ```
#include <wchar.h>
```

15794 ```
int fwscanf(FILE *restrict stream, const wchar_t *restrict format, ...);
```

15795 ```
int swscanf(const wchar_t *restrict ws,
```

15796 ```
    const wchar_t *restrict format, ...);
```

15797 ```
int wscanf(const wchar_t *format, ... );
```

15798 **DESCRIPTION**

15799 CX The functionality described on this reference page is aligned with the ISO C standard. Any
15800 conflict between the requirements described here and the ISO C standard is unintentional. This
15801 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

15802 The *fwscanf*( ) function reads from the named input *stream*. The *wscanf*( ) function reads from the
15803 standard input stream *stdin*. The *swscanf*( ) function reads from the wide-character string *ws*.
15804 Each function reads wide characters, interprets them according to a format, and stores the
15805 results in its arguments. Each expects, as arguments, a control wide-character string *format*
15806 described below, and a set of *pointer* arguments indicating where the converted input should be
15807 stored. The result is undefined if there are insufficient arguments for the format. If the format is
15808 exhausted while arguments remain, the excess arguments are evaluated but are otherwise
15809 ignored.

15810 XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
15811 to the next unused argument. In this case, the conversion wide character '%' (see below) is
15812 replaced by the sequence "%n$", where *n* is a decimal integer in the range [1,{NL_ARGMAX}].
15813 This feature provides for the definition of format wide-character strings that select arguments in
15814 an order appropriate to specific languages. In format wide-character strings containing the
15815 "%n$" form of conversion specifications, it is unspecified whether numbered arguments in the
15816 argument list can be referenced from the format wide-character string more than once.

15817 The *format* can contain either form of a conversion specification—that is, '%' or "%n$"—but the
15818 two forms cannot normally be mixed within a single *format* wide-character string. The only
15819 exception to this is that "%%" or "%*" can be mixed with the "%n$" form.

15820 The *fwscanf*( ) function in all its forms allows for detection of a language-dependent radix
15821 character in the input string, encoded as a wide-character value. The radix character is defined in
15822 the program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the
15823 radix character is not defined, the radix character defaults to a period ('.').

15824 The *format* is a wide-character string composed of zero or more directives. Each directive is
15825 composed of one of the following: one or more white-space wide characters (<space>, <tab>,
15826 <newline>, <vertical-tab>, or <form-feed> characters); an ordinary wide character (neither '%'
15827 nor a white-space character); or a conversion specification. Each conversion specification is
15828 XSI introduced by a '%' or the sequence "%n$" after which the following appear in sequence:

15829 - An optional assignment-suppressing character '*'.
15830 - An optional non-zero decimal integer that specifies the maximum field width.
15831 - An optional length modifier that specifies the size of the receiving object.
15832 - A conversion wide character that specifies the type of conversion to be applied. The valid
15833 conversion wide characters are described below.

The *fwscanf*( ) functions execute each directive of the format in turn. If a directive fails, as detailed below, the function shall return. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space wide characters is executed by reading input until no more valid input can be read, or up to the first wide character which is not a white-space wide character, which remains unread.

A directive that is an ordinary wide character is executed as follows. The next wide character is read from the input and compared with the wide character that comprises the directive; if the comparison shows that they are not equivalent, the directive fails, and the differing and subsequent wide characters remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a wide character from being read, the directive fails.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion wide character. A conversion specification is executed in the following steps.

Input white-space wide characters (as specified by *iswspace*( )) are skipped, unless the conversion specification includes a '[', *c*, or *n* conversion character.

An item is read from the input, unless the conversion specification includes an *n* conversion wide character. An input item is defined as the longest sequence of input wide characters, not exceeding any specified field width, which is an initial subsequence of a matching sequence. The first wide character, if any, after the input item remains unread. If the length of the input item is 0, the execution of the conversion specification fails; this condition is a matching failure, unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a '%' conversion wide character, the input item (or, in the case of a %*n* conversion specification, the count of input wide characters) is converted to a type appropriate to the conversion wide character. If the input item is not a matching sequence, the execution of the conversion specification fails; this condition is a matching failure. Unless assignment suppression was indicated by a '*', the result of the conversion is placed in the object pointed to by the first argument following the *format* argument that has not already received a
XSI conversion result if the conversion specification is introduced by '%', or in the *n*th argument if introduced by the wide-character sequence "%n$". If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The length modifiers and their meanings are:

hh Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an argument with type pointer to **signed char** or **unsigned char**.

h Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an argument with type pointer to **short** or **unsigned short**.

l (ell) Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an argument with type pointer to **long** or **unsigned long**; that a following *a*, *A*, *e*, *E*, *f*, *F*, *g*, or *G* conversion specifier applies to an argument with type pointer to **double**; or that a following *c*, *s*, or '[' conversion specifier applies to an argument with type pointer to **wchar_t**.

ll (ell-ell) Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an argument with type pointer to **long long** or **unsigned long long**.

15879 *j* Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an argument
15880 with type pointer to **intmax_t** or **uintmax_t**.

15881 *z* Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an argument
15882 with type pointer to **size_t** or the corresponding signed integer type.

15883 *t* Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an argument
15884 with type pointer to **ptrdiff_t** or the corresponding **unsigned** type.

15885 *L* Specifies that a following *a*, *A*, *e*, *E*, *f*, *F*, *g*, or *G* conversion specifier applies to an
15886 argument with type pointer to **long double**.

15887 If a length modifier appears with any conversion specifier other than as specified above, the
15888 behavior is undefined.

15889 The following conversion wide characters are valid:

15890 *d* Matches an optionally signed decimal integer, whose format is the same as expected for
15891 the subject sequence of *wcstol*( ) with the value 10 for the *base* argument. In the absence
15892 of a size modifier, the application shall ensure that the corresponding argument is a
15893 pointer to **int**.

15894 *i* Matches an optionally signed integer, whose format is the same as expected for the
15895 subject sequence of *wcstol*( ) with 0 for the *base* argument. In the absence of a size
15896 modifier, the application shall ensure that the corresponding argument is a pointer to
15897 **int**.

15898 *o* Matches an optionally signed octal integer, whose format is the same as expected for
15899 the subject sequence of *wcstoul*( ) with the value 8 for the *base* argument. In the absence
15900 of a size modifier, the application shall ensure that the corresponding argument is a
15901 pointer to **unsigned**.

15902 *u* Matches an optionally signed decimal integer, whose format is the same as expected for
15903 the subject sequence of *wcstoul*( ) with the value 10 for the *base* argument. In the absence
15904 of a size modifier, the application shall ensure that the corresponding argument is a
15905 pointer to **unsigned**.

15906 *x* Matches an optionally signed hexadecimal integer, whose format is the same as
15907 expected for the subject sequence of *wcstoul*( ) with the value 16 for the *base* argument.
15908 In the absence of a size modifier, the application shall ensure that the corresponding
15909 argument is a pointer to **unsigned**.

15910 *a*, *e*, *f*, *g* Matches an optionally signed floating-point number, infinity, or NaN whose format is
15911 the same as expected for the subject sequence of *wcstod*( ). In the absence of a size
15912 modifier, the application shall ensure that the corresponding argument is a pointer to
15913 **float**.

15914 If the *fwprintf*( ) family of functions generates character string representations for
15915 infinity and NaN (a symbolic entity encoded in floating-point format) to support
15916 IEEE Std. 754-1985, the *fwscanf*( ) family of functions shall recognize them as input.

15917 *s* Matches a sequence of non white-space wide characters. If no *l* (ell) qualifier is present,
15918 characters from the input field are converted as if by repeated calls to the *wcrtomb*( )
15919 function, with the conversion state described by an **mbstate_t** object initialized to zero
15920 before the first wide character is converted. The application shall ensure that the
15921 corresponding argument is a pointer to a character array large enough to accept the
15922 sequence and the terminating null character, which shall be added automatically.

Otherwise, the application shall ensure that the corresponding argument is a pointer to an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which shall be added automatically.

[Matches a non-empty sequence of wide characters from a set of expected wide characters (the *scanset*). If no *l* (ell) qualifier is present, wide characters from the input field are converted as if by repeated calls to the *wcrtomb*( ) function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted. The application shall ensure that the corresponding argument is a pointer to a character array large enough to accept the sequence and the terminating null character, which shall be added automatically.

If an *l* (ell) qualifier is present, the application shall ensure that the corresponding argument is a pointer to an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which shall be added automatically.

The conversion specification includes all subsequent wide characters in the *format* string up to and including the matching right square bracket (']'). The wide characters between the square brackets (the *scanlist*) comprise the scanset, unless the wide character after the left square bracket is a circumflex ('^'), in which case the scanset contains all wide characters that do not appear in the scanlist between the circumflex and the right square bracket. If the conversion specification begins with "[]" or "[^]", the right square bracket is included in the scanlist and the next right square bracket is the matching right square bracket that ends the conversion specification; otherwise, the first right square bracket is the one that ends the conversion specification. If a '-' is in the scanlist and is not the first wide character, nor the second where the first wide character is a '^', nor the last wide character, the behavior is implementation-defined.

c Matches a sequence of wide characters of exactly the number specified by the field width (1 if no field width is present in the conversion specification).

If no *l* (ell) length modifier is present, characters from the input field are converted as if by repeated calls to the *wcrtomb*( ) function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.

If an *l* (ell) length modifier is present, the corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence. No null wide character is added.

Otherwise, the application shall ensure that the corresponding argument is a pointer to an array of **wchar_t** large enough to accept the sequence. No null wide character is added.

p Matches an implementation-defined set of sequences, which shall be the same as the set of sequences that is produced by the %*p* conversion of the corresponding *fwprintf*( ) functions. The application shall ensure that the corresponding argument is a pointer to a pointer to **void**. The interpretation of the input item is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise, the behavior of the %*p* conversion is undefined.

n No input is consumed. The application shall ensure that the corresponding argument is a pointer to the integer into which is to be written the number of wide characters read from the input so far by this call to the *fwscanf*( ) functions. Execution of a %*n*

15971 conversion specification does not increment the assignment count returned at the
15972 completion of execution of the function. No argument is converted, but one is
15973 consumed. If the conversion specification includes an assignment-suppressing wide
15974 character or a field width, the behavior is undefined.

15975 XSI *C* Same as *lc*.

15976 *S* Same as *ls*.

15977 % Matches a single '%'; no conversion or assignment occurs. The complete conversion
15978 specification shall be "%%".

15979 If a conversion specification is invalid, the behavior is undefined.

15980 The conversion characters *A*, *E*, *F*, *G*, and *X* are also valid and behave the same as, respectively, *a*,
15981 *e*, *f*, *g*, and *x*.

15982 If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before
15983 any wide characters matching the current conversion specification (except for %*n*) have been
15984 read (other than leading white-space, where permitted), execution of the current conversion
15985 specification terminates with an input failure. Otherwise, unless execution of the current
15986 conversion specification is terminated with a matching failure, execution of the following
15987 conversion specification (if any) is terminated with an input failure.

15988 Reaching the end of the string in *swscanf*() is equivalent to encountering end-of-file for *fwscanf*().

15989 If conversion terminates on a conflicting input, the offending input is left unread in the input.
15990 Any trailing white space (including <newline>) is left unread unless matched by a conversion
15991 specification. The success of literal matches and suppressed assignments is only directly
15992 determinable via the %*n* conversion specification.

15993 The *fwscanf*() and *wscanf*() functions may mark the *st_atime* field of the file associated with
15994 *stream* for update. The *st_atime* field shall be marked for update by the first successful execution
15995 of *fgetc*(), *fgetwc*(), *fgets*(), *fgetws*(), *fread*(), *getc*(), *getwc*(), *getchar*(), *getwchar*(), *gets*(), *fscanf*(),
15996 or *fwscanf*() using *stream* that returns data not supplied by a prior call to *ungetc*().

15997 **RETURN VALUE**

15998 Upon successful completion, these functions shall return the number of successfully matched
15999 and assigned input items; this number can be 0 in the event of an early matching failure. If the
16000 input ends before the first matching failure or conversion, EOF shall be returned. If a read error
16001 CX occurs the error indicator for the stream is set, EOF shall be returned, and *errno* shall be set to
16002 indicate the error.

16003 **ERRORS**

16004 For the conditions under which the *fwscanf*() functions shall fail and may fail, refer to *fgetwc*().

16005 In addition, *fwscanf*() may fail if:

16006 XSI [EILSEQ] Input byte sequence does not form a valid character.

16007 XSI [EINVAL] There are insufficient arguments.

16008 **EXAMPLES**

16009 The call:

16010 ```
int i, n; float x; char name[50];
16011 n = wscanf(L"%d%f%s", &i, &x, name);
```

16012 with the input line:

16013 ```
25 54.32E-1 Hamster
```

16014 assigns to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* contains the string
16015 `"Hamster"`.

16016 The call:

16017 ```
int i; float x; char name[50];
16018 (void) wscanf(L"%2d%f%*d %[0123456789]", &i, &x, name);
```

16019 with input:

16020 ```
56789 0123 56a72
```

16021 assigns 56 to *i*, 789.0 to *x*, skip 0123, and place the string `"56\0"` in *name*. The next call to
16022 *getchar*( ) shall return the character `'a'`.

16023 **APPLICATION USAGE**

16024 In format strings containing the `'%'` form of conversion specifications, each argument in the
16025 argument list is used exactly once.

16026 **RATIONALE**

16027 None.

16028 **FUTURE DIRECTIONS**

16029 None.

16030 **SEE ALSO**

16031 *getwc*( ), *fwprintf*( ), *setlocale*( ), *wcstod*( ), *wcstol*( ), *wcstoul*( ), *wcrtomb*( ), the Base Definitions
16032 volume of IEEE Std. 1003.1-200x, **<langinfo.h>**, **<stdio.h>**, **<wchar.h>**, the Base Definitions
16033 volume of IEEE Std. 1003.1-200x, Chapter 7, Locale

16034 **CHANGE HISTORY**

16035 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
16036 (E).

16037 **Issue 6**

16038 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

16039 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

16040 • The prototypes for *fwscanf*( ) and *swscanf*( ) are updated.

16041 • The DESCRIPTION is updated.

16042 **NAME**

16043 gai_strerror — address and name information error description

16044 **SYNOPSIS**

16045 `#include <netdb.h>`

16046 `char *gai_strerror(int ecode);`

16047 **DESCRIPTION**

16048 The *gai_strerror*( ) function shall return a text string describing an error that is listed in the
16049 **<netdb.h>** header.

16050 When the *ecode* argument is one of the values listed in the **<netdb.h>** header, the function return
16051 value points to a string describing the error. If the argument is not one of those values, the
16052 function shall return a pointer to a string whose contents indicate an unknown error.

16053 **RETURN VALUE**

16054 Upon successful completion, *gai_strerror*( ) shall return a pointer to an implementation-defined |
16055 string. |

16056 **ERRORS**

16057 No errors are defined.

16058 **EXAMPLES**

16059 None.

16060 **APPLICATION USAGE**

16061 None.

16062 **RATIONALE**

16063 None.

16064 **FUTURE DIRECTIONS**

16065 None.

16066 **SEE ALSO**

16067 *getaddrinfo*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<netdb.h>** |

16068 **CHANGE HISTORY**

16069 First released in Issue 6. Derived from the XNS, Issue 5.2 specification. |

16070 **NAME**

16071 gcvt — convert a floating-point number to a string (**LEGACY**)

16072 **SYNOPSIS**

16073 XSI
```
#include <stdlib.h>
```

16074
```
char *gcvt(double value, int ndigit, char *buf);
```

16075

16076 **DESCRIPTION**

16077 Refer to *ecvt*( ).

16078 **NAME**

16079 getaddrinfo — get address information

16080 **SYNOPSIS**

16081 `#include <sys/socket.h>`

16082 `#include <netdb.h>`

16083 `int getaddrinfo(const char *restrict` *`nodename`*`,` |

16084 `    const char *restrict` *`servname`*`,` |

16085 `    const struct addrinfo *restrict` *`hints`*`,` |

16086 `    struct addrinfo **restrict` *`res`*`);` |

16087 **DESCRIPTION** |

16088 Refer to *freeaddrinfo*( ).

16089 **NAME**

16090 getc — get a byte from a stream

16091 **SYNOPSIS**

16092 `#include <stdio.h>`

16093 `int getc(FILE *stream);`

16094 **DESCRIPTION**

16095 CX The functionality described on this reference page is aligned with the ISO C standard. Any
16096 conflict between the requirements described here and the ISO C standard is unintentional. This
16097 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

16098 The *getc*( ) function shall be equivalent to *fgetc*( ), except that if it is implemented as a macro it
16099 may evaluate *stream* more than once, so the argument should never be an expression with side
16100 effects.

16101 **RETURN VALUE**

16102 Refer to *fgetc*( ).

16103 **ERRORS**

16104 Refer to *fgetc*( ).

16105 **EXAMPLES**

16106 None.

16107 **APPLICATION USAGE**

16108 If the integer value returned by *getc*( ) is stored into a variable of type **char** and then compared
16109 against the integer constant EOF, the comparison may never succeed, because sign-extension of
16110 a variable of type **char** on widening to integer is implementation-defined. |

16111 Because it may be implemented as a macro, *getc*( ) may treat incorrectly a *stream* argument with
16112 side effects. In particular, *getc*(*f*++) does not necessarily work as expected. Therefore, use of this
16113 function should be preceded by `"#undef getc"` in such situations; *fgetc*( ) could also be used.

16114 **RATIONALE**

16115 None.

16116 **FUTURE DIRECTIONS**

16117 None.

16118 **SEE ALSO**

16119 *fgetc*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>** |

16120 **CHANGE HISTORY**

16121 First released in Issue 1. Derived from Issue 1 of the SVID. |

16122 **Issue 4**

16123 The words ‘‘a character variable’’ are replaced by ‘‘a variable of type **char**’’, to emphasize the fact
16124 that this function deals with byte values.

16125 The APPLICATION USAGE section now states that the use of this function is not recommended.

16126 **NAME**

16127 getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked — stdio with explicit client
16128 locking

16129 **SYNOPSIS**

16130 TSF
```
#include <stdio.h>

int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
```
16135

16136 **DESCRIPTION**

16137 Versions of the functions *getc*(), *getchar*(), *putc*(), and *putchar*() respectively named
16138 *getc_unlocked*(), *getchar_unlocked*(), *putc_unlocked*(), and *putchar_unlocked*() shall be provided
16139 which are functionally identical to the original versions, with the exception that they are not
16140 required to be implemented in a thread-safe manner. They may only safely be used within a
16141 scope protected by *flockfile*() (or *ftrylockfile*()) and *funlockfile*(). These functions may safely be
16142 used in a multi-threaded program if and only if they are called while the invoking thread owns
16143 the (**FILE***) object, as is the case after a successful call of the *flockfile*() or *ftrylockfile*() functions.

16144 **RETURN VALUE**

16145 See *getc*(), *getchar*(), *putc*(), and *putchar*().

16146 **ERRORS**

16147 No errors are defined.

16148 **EXAMPLES**

16149 None.

16150 **APPLICATION USAGE**

16151 Because they may be implemented as macros, *getc_unlocked*() and *putc_unlocked*() may treat
16152 incorrectly a stream argument with side effects. In particular, *getc_unlocked*(*f++) and
16153 *putc_unlocked*(*f++) do not necessarily work as expected. Therefore, use of these functions in
16154 such situations should be preceded by the following statement as appropriate:

```
16155 #undef getc_unlocked
16156 #undef putc_unlocked
```

16157 **RATIONALE**

16158 Some I/O functions are typically implemented as macros for performance reasons (for example,
16159 *putc*() and *getc*()). For safety, they need to be synchronized, but it is often too expensive to
16160 synchronize on every character. Nevertheless, it was felt that the safety concerns were more
16161 important; consequently, the *getc*(), *getchar*(), *putc*(), and *putchar*() functions are required to be
16162 thread-safe. However, unlocked versions are also provided with names that clearly indicate the
16163 unsafe nature of their operation but can be used to exploit their higher performance. These
16164 unlocked versions can be safely used only within explicitly locked program regions, using
16165 exported locking primitives. In particular, a sequence such as:

```
16166 flockfile(fileptr);
16167 putc_unlocked('1', fileptr);
16168 putc_unlocked('\n', fileptr);
16169 fprintf(fileptr, "Line 2\n");
16170 funlockfile(fileptr);
```

16171 is permissible, and results in the text sequence:

```
1
Line 2
```

being printed without being interspersed with output from other threads.

It would be wrong to have the standard names such as *getc*( ), *putc*( ), and so on, map to the ''faster, but unsafe'' rather than the ''slower, but safe'' versions. In either case, you would still want to inspect all uses of *getc*( ), *putc*( ), and so on, by hand when converting existing code. Choosing the safe bindings as the default, at least, results in correct code and maintains the ''atomicity at the function'' invariant. To do otherwise would introduce gratuitous synchronization errors into converted code. Other routines that modify the *stdio* (**FILE***) structures or buffers are also safely synchronized.

Note that there is no need for functions of the form *getc_locked*( ), *putc_locked*( ), and so on, since this is the functionality of *getc*( ), *putc*( ), *et al*. It would be inappropriate to use a feature test macro to switch a macro definition of *getc*( ) between *getc_locked*( ) and *getc_unlocked*( ), since the ISO C standard requires an actual function to exist, a function whose behavior could not be changed by the feature test macro. Also, providing both the *xxx_locked*( ) and *xxx_unlocked*( ) forms leads to the confusion of whether the suffix describes the behavior of the function or the circumstances under which it should be used.

Three additional routines, *flockfile*( ), *ftrylockfile*( ), and *funlockfile*( ) (which may be macros), are provided to allow the user to delineate a sequence of I/O statements that are executed synchronously.

The *ungetc*( ) function is infrequently called relative to the other functions/macros so no unlocked variation is needed.

FUTURE DIRECTIONS

None.

SEE ALSO

getc( ), *getchar*( ), *putc*( ), *putchar*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

These functions are marked as part of the Thread-Safe Functions option.

The Open Group corrigenda item U030/2 has been applied adding APPLICATION USAGE describing how applications should be written to avoid the case when the functions are implemented as macros.

16206 **NAME**

16207 getchar — get a byte from a stdin stream

16208 **SYNOPSIS**

16209
```
#include <stdio.h>
```

16210
```
int getchar(void);
```

16211 **DESCRIPTION**

16212 CX The functionality described on this reference page is aligned with the ISO C standard. Any
16213 conflict between the requirements described here and the ISO C standard is unintentional. This
16214 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

16215 The *getchar*( ) function shall be equivalent to *getc*(*stdin*).

16216 **RETURN VALUE**

16217 Refer to *fgetc*( ).

16218 **ERRORS**

16219 Refer to *fgetc*( ).

16220 **EXAMPLES**

16221 None.

16222 **APPLICATION USAGE**

16223 If the integer value returned by *getchar*( ) is stored into a variable of type **char** and then
16224 compared against the integer constant EOF, the comparison may never succeed, because sign-
16225 extension of a variable of type **char** on widening to integer is implementation-defined. |

16226 **RATIONALE**

16227 None.

16228 **FUTURE DIRECTIONS**

16229 None.

16230 **SEE ALSO**

16231 *getc*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>** |

16232 **CHANGE HISTORY**

16233 First released in Issue 1. Derived from Issue 1 of the SVID. |

16234 **Issue 4**

16235 The words ‘‘a character variable’’ are replaced by ‘‘a variable of type **char**’’, to emphasize the fact
16236 that this function deals in byte values.

16237 The following change is incorporated for alignment with the ISO C standard:

16238 • The argument list is explicitly defined as **void**.

16239 **NAME**

16240 getcontext, setcontext — get and set current user context

16241 **SYNOPSIS**

16242 XSI
```
#include <ucontext.h>

int getcontext(ucontext_t *ucp);
int setcontext(const ucontext_t *ucp);
```
16243

16244

16245

16246 **DESCRIPTION**

16247 The *getcontext*( ) function shall initialize the structure pointed to by *ucp* to the current user
16248 context of the calling thread. The **ucontext_t** type that *ucp* points to defines the user context and
16249 includes the contents of the calling thread's machine registers, the signal mask, and the current
16250 execution stack.

16251 The *setcontext*( ) function shall restore the user context pointed to by *ucp*. A successful call to
16252 *setcontext*( ) shall not return; program execution resumes at the point specified by the *ucp*
16253 argument passed to *setcontext*( ). The *ucp* argument should be created either by a prior call to
16254 *getcontext*( ) or *makecontext*( ), or by being passed as an argument to a signal handler. If the *ucp*
16255 argument was created with *getcontext*( ), program execution continues as if the corresponding
16256 call of *getcontext*( ) had just returned. If the *ucp* argument was created with *makecontext*( ),
16257 program execution continues with the function passed to *makecontext*( ). When that function
16258 returns, the thread shall continue as if after a call to *setcontext*( ) with the *ucp* argument that was
16259 input to *makecontext*( ). If the *uc_link* member of the **ucontext_t** structure pointed to by the *ucp*
16260 argument is equal to 0, then this context is the main context, and the thread shall exit when this
16261 context returns. The effects of passing a *ucp* argument obtained from any other source are
16262 unspecified.

16263 **RETURN VALUE**

16264 Upon successful completion, *setcontext*( ) shall not return and *getcontext*( ) shall return 0;
16265 otherwise, a value of –1 shall be returned.

16266 **ERRORS**

16267 No errors are defined.

16268 **EXAMPLES**

16269 None.

16270 **APPLICATION USAGE**

16271 When a signal handler is executed, the current user context is saved and a new context is
16272 created. If the thread leaves the signal handler via *longjmp*( ), then it is unspecified whether the
16273 context at the time of the corresponding *setjmp*( ) call is restored and thus whether future calls to
16274 *getcontext*( ) provide an accurate representation of the current context, since the context restored
16275 by *longjmp*( ) does not necessarily contain all the information that *setcontext*( ) requires. Signal
16276 handlers should use *siglongjmp*( ) or *setcontext*( ) instead.

16277 Portable applications should not modify or access the *uc_mcontext* member of **ucontext_t**. A
16278 portable application cannot assume that context includes any process-wide static data, possibly
16279 including *errno*. Users manipulating contexts should take care to handle these explicitly when
16280 required.

16281 Use of contexts to create alternate stacks is not defined by this volume of IEEE Std. 1003.1-200x.

16282 **RATIONALE**

16283 None.

16284 **FUTURE DIRECTIONS**

16285 None.

16286 **SEE ALSO**

16287 *bsd_signal*( ), *makecontext*( ), *setjmp*( ), *sigaction*( ), *sigaltstack*( ), *sigprocmask*( ), *sigsetjmp*( ), the Base |
16288 Definitions volume of IEEE Std. 1003.1-200x, **<ucontext.h>** |

16289 **CHANGE HISTORY**

16290 First released in Issue 4, Version 2.

16291 **Issue 5**

16292 Moved from X/OPEN UNIX extension to BASE.

16293 The following sentence was removed from the DESCRIPTION: ‘‘If the *ucp* argument was passed
16294 to a signal handler, program execution continues with the program instruction following the
16295 instruction interrupted by the signal.’’

16296 **NAME**

16297 getcwd — get the path name of the current working directory

16298 **SYNOPSIS**

16299 ```
#include <unistd.h>
```

16300 ```
char *getcwd(char *buf, size_t size);
```

16301 **DESCRIPTION**

16302 The *getcwd*( ) function shall place an absolute path name of the current working directory in the
16303 array pointed to by *buf*, and return *buf*. The path name copied to the array shall contain no
16304 components that are symbolic links. The *size* argument is the size in bytes of the character array
16305 pointed to by the *buf* argument. If *buf* is a null pointer, the behavior of *getcwd*( ) is unspecified. |

16306 **RETURN VALUE**

16307 Upon successful completion, *getcwd*( ) shall return the *buf* argument. Otherwise, *getcwd*( ) shall
16308 return a null pointer and set *errno* to indicate the error. The contents of the array pointed to by
16309 *buf* are then undefined.

16310 **ERRORS**

16311 The *getcwd*( ) function shall fail if:

16312 [EINVAL] The *size* argument is 0. |

16313 [ERANGE] The size argument is greater than 0, but is smaller than the length of the path |
16314 name +1.

16315 The *getcwd*( ) function may fail if:

16316 [EACCES] Read or search permission was denied for a component of the path name. |

16317 [ENOMEM] Insufficient storage space is available. |

16318 **EXAMPLES**

16319 **Determining the Absolute Path Name of the Current Working Directory**

16320 The following example returns a pointer to an array that holds the absolute path name of the
16321 current working directory. The pointer is returned in the *ptr* variable, which points to the *buf*
16322 array where the path name is stored.

```
16323 #include <stdlib.h>
16324 #include <unistd.h>
16325 ...
16326 long size;
16327 char *buf;
16328 char *ptr;
16329 size = pathconf(".", _PC_PATH_MAX);
16330 if ((buf = (char *)malloc((size_t)size)) != NULL)
16331     ptr = getcwd(buf, (size_t)size);
16332 ...
```

16333 **APPLICATION USAGE**

16334 None. |

16335 **RATIONALE**

16336 Since the maximum path name length is arbitrary unless {PATH_MAX} is defined, an
16337 application generally cannot supply a *buf* with *size* {{PATH_MAX} +1}.

16338 Having *getcwd*( ) take no arguments and instead use the *malloc*( ) function to produce space for
16339 the returned argument was considered. The advantage is that *getcwd*( ) knows how big the
16340 working directory path name is and can allocate an appropriate amount of space. But the
16341 programmer would have to use the *free*( ) function to free the resulting object, or each use of
16342 *getcwd*( ) would further reduce the available memory. Also, *malloc*( ) and *free*( ) are used nowhere
16343 else in this volume of IEEE Std. 1003.1-200x. Finally, *getcwd*( ) is taken from the SVID where it
16344 has the two arguments used in this volume of IEEE Std. 1003.1-200x.

16345 The older function *getwd*( ) was rejected for use in this context because it had only a buffer
16346 argument and no *size* argument, and thus had no way to prevent overwriting the buffer, except
16347 to depend on the programmer to provide a large enough buffer.

16348 On some implementations, if *buf* is a null pointer, *getcwd*( ) may obtain *size* bytes of memory |
16349 using *malloc*( ). In this case, the pointer returned by *getcwd*( ) may be used as the argument in a |
16350 subsequent call to *free*( ). Invoking *getcwd*( ) with *buf* as a null pointer is not recommended in |
16351 portable applications. |

16352 If a program is operating in a directory where some (grand)parent directory does not permit
16353 reading, *getcwd*( ) may fail, as in most implementations it must read the directory to determine
16354 the name of the file. This can occur if search, but not read, permission is granted in an
16355 intermediate directory, or if the program is placed in that directory by some more privileged
16356 process (for example, login). Including the [EACCES] error condition makes the reporting of the |
16357 error consistent and warns the application writer that *getcwd*( ) can fail for reasons beyond the
16358 control of the application writer or user. Some implementations can avoid this occurrence (for
16359 example, by implementing *getcwd*( ) using *pwd*, where *pwd* is a set-user-root process), thus the
16360 error was made optional.

16361 Because this volume of IEEE Std. 1003.1-200x permits the addition of other errors, this would be
16362 a common addition and yet one that applications could not be expected to deal with without
16363 this addition.

16364 **FUTURE DIRECTIONS**

16365 None.

16366 **SEE ALSO**

16367 *malloc*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>** |

16368 **CHANGE HISTORY**

16369 First released in Issue 1. Derived from Issue 1 of the SVID. |

16370 **Issue 4**

16371 The **<unistd.h>** header is added to the SYNOPSIS section.

16372 The [ENOMEM] error is marked as an extension.

16373 The words ‘‘as this functionality may be subject to withdrawal’’ have been deleted from the end
16374 of the last sentence in the APPLICATION USAGE section.

16375 The following change is incorporated for alignment with the ISO POSIX-1 standard:

16376 • The DESCRIPTION is changed to indicate that the effects of passing a null pointer in *buf* are
16377 undefined.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [ENOMEM] optional error condition is added.

16382 **NAME**

16383 getdate — convert user format date and time

16384 **SYNOPSIS**

16385 XSI
```
#include <time.h>
```

16386
```
struct tm *getdate(const char *string);
```

16387

16388 **DESCRIPTION**

16389 The *getdate*( ) function shall convert a string representation of a date or time into a broken-down
16390 time.

16391 The external variable or macro *getdate_err* is used by *getdate*( ) to return error values.

16392 Templates are used to parse and interpret the input string. The templates are contained in a text
16393 file identified by the environment variable *DATEMSK*. The *DATEMSK* variable should be set to
16394 indicate the full path name of the file that contains the templates. The first line in the template
16395 that matches the input specification is used for interpretation and conversion into the internal
16396 time format.

16397 The following field descriptors are supported:

16398 %% Same as '%'.

16399 %*a* Abbreviated weekday name.

16400 %*A* Full weekday name.

16401 %*b* Abbreviated month name.

16402 %*B* Full month name.

16403 %*c* Locale's appropriate date and time representation.

16404 %*C* Century number (00-99; leading zeros are permitted but not required).

16405 %*d* Day of month (01-31; the leading 0 is optional).

16406 %*D* Date as %*m* / %*d* / %*y*.

16407 %*e* Same as %*d*.

16408 %*h* Abbreviated month name.

16409 %*H* Hour (00-23).

16410 %*I* Hour (01-12).

16411 %*m* Month number (01-12).

16412 %*M* Minute (00-59).

16413 %*n* Same as <newline>.

16414 %*p* Locale's equivalent of either AM or PM.

16415 %*r* The locale's appropriate representation of time in AM and PM notation. In the POSIX
16416 locale, this is equivalent to %*I*:%*M*:%*S* %*p*.

16417 %*R* Time as %*H*:%*M*.

16418 %*S* Seconds (00-61). Leap seconds are allowed but are not predictable through use of
16419 algorithms.

16420 %*t* Same as <tab>.

16421 %*T* Time as %*H*:%*M*:%*S*.

16422 %*w* Weekday number (Sunday = 0-6).

16423 %*x* Locale's appropriate date representation.

16424 %*X* Locale's appropriate time representation.

16425 %*y* Year within century. When a century is not otherwise specified, values in the range
16426 69-99 refer to years in the twentieth century (1969 to 1999 inclusive); values in the range
16427 00-68 refer to years in the twenty-first century (2000 to 2068 inclusive).

16428 %*Y* Year as *ccyy* (for example, 1994).

16429 %*Z* Timezone name or no characters if no timezone exists. If the timezone supplied by %*Z*
16430 is not the timezone that *getdate*( ) expects, an invalid input specification error shall
16431 result. The *getdate*( ) function calculates an expected timezone based on information
16432 supplied to the function (such as the hour, day, and month).

16433 The match between the template and input specification performed by *getdate*( ) is case-
16434 insensitive.

16435 The month and weekday names can consist of any combination of upper and lowercase letters.
16436 The process can request that the input date or time specification be in a specific language by
16437 setting the *LC_TIME* category (see *setlocale*( )).

16438 Leading 0s are not necessary for the descriptors that allow leading 0s. However, at most two
16439 digits are allowed for those descriptors, including leading 0s. Extra whitespace in either the
16440 template file or in *string* is ignored.

16441 The field descriptors %*c*, %*x*, and %*X* shall not be supported if they include unsupported field
16442 descriptors.

16443 The following rules apply for converting the input specification into the internal format:

16444 • If %*Z* is being scanned, then *getdate*( ) initializes the broken-down time to be the current time
16445 in the scanned timezone. Otherwise, it initializes the broken-down time based on the current
16446 local time as if *localtime*( ) had been called.

16447 • If only the weekday is given, today is assumed if the given day is equal to the current day
16448 and next week if it is less,

16449 • If only the month is given, the current month is assumed if the given month is equal to the
16450 current month and next year if it is less, and no year is given (the first day of month is
16451 assumed if no day is given),

16452 • If no hour, minute, and second are given the current hour, minute, and second are assumed,

16453 • If no date is given, today is assumed if the given hour is greater than the current hour and
16454 tomorrow is assumed if it is less.

16455 If a field descriptor specification in the DATEMSK file does not correspond to one of the field
16456 descriptors above, the behavior is unspecified.

16457 The *getdate*( ) function need not be reentrant. A function that is not required to be reentrant is not
16458 required to be thread-safe.

RETURN VALUE

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

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.

1. The *DATEMSK* environment variable is null or undefined.
2. The template file cannot be opened for reading.
3. Failed to get file status information.
4. The template file is not a regular file.
5. An I/O error is encountered while reading the template file.
6. Memory allocation failed (not enough memory available).
7. There is no line in the template that matches the input.
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).

EXAMPLES

1. The following example shows the possible contents of a template:

```
%m
%A %B %d, %Y, %H:%M:%S
%A
%B
%m/%d/%y %I %p
%d,%m,%Y %H:%M
at %A the %dst of %B in %Y
run job at %I %p,%B %dnd
%A den %d. %B %Y %H.%M Uhr
```

2. The following are examples of valid input specifications for the template in Example 1:

```
getdate("10/1/87 4 PM");
getdate("Friday");
getdate("Friday September 18, 1987, 10:30:30");
getdate("24,9,1986 10:30");
getdate("at monday the 1st of december in 1986");
getdate("run job at 3 PM, december 2nd");
```

   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:

```
getdate("freitag den 10. oktober 1986 10.30 Uhr");
```

3. The following example shows how local date and time specification can be defined in the template:

Invocation	Line in Template
`getdate("11/27/86")`	`%m/%d/%y`
`getdate("27.11.86")`	`%d.%m.%y`
`getdate("86-11-27")`	`%y-%m-%d`
`getdate("Friday 12:00:00")`	`%A %H:%M:%S`

4. The following examples help to illustrate the above rules assuming that the current date is Mon Sep 22 12:19:47 EDT 1986 and the *LC_TIME* category is set to the default C locale:

Input	Line in Template	Date
`Mon`	`%a`	Mon Sep 22 12:19:47 EDT 1986
`Sun`	`%a`	Sun Sep 28 12:19:47 EDT 1986
`Fri`	`%a`	Fri Sep 26 12:19:47 EDT 1986
`September`	`%B`	Mon Sep 1 12:19:47 EDT 1986
`January`	`%B`	Thu Jan 1 12:19:47 EST 1987
`December`	`%B`	Mon Dec 1 12:19:47 EST 1986
`Sep Mon`	`%b %a`	Mon Sep 1 12:19:47 EDT 1986
`Jan Fri`	`%b %a`	Fri Jan 2 12:19:47 EST 1987
`Dec Mon`	`%b %a`	Mon Dec 1 12:19:47 EST 1986
`Jan Wed 1989`	`%b %a %Y`	Wed Jan 4 12:19:47 EST 1989
`Fri 9`	`%a %H`	Fri Sep 26 09:00:00 EDT 1986
`Feb 10:30`	`%b %H:%S`	Sun Feb 1 10:00:30 EST 1987
`10:30`	`%H:%M`	Tue Sep 23 10:30:00 EDT 1986
`13:30`	`%H:%M`	Mon Sep 22 13:30:00 EDT 1986

APPLICATION USAGE

Although historical versions of *getdate*() did not require that **<time.h>** declare the external variable *getdate_err*, this volume of IEEE Std. 1003.1-200x does require it. The Open Group encourages applications to remove declarations of *getdate_err* and instead incorporate the declaration by including **<time.h>**.

Applications should use %*Y* (4-digit years) in preference to %*y* (2-digit years).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ctime(), *localtime*(), *setlocale*(), *strftime*(), *times*(), the Base Definitions volume of IEEE Std. 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The last paragraph of the DESCRIPTION is added.

The %*C* specifier is added, and the exact meaning of the %*y* specifier is clarified in the DESCRIPTION.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

The %*R* specifier is changed to follow historical practice.

Issue 6

The DESCRIPTION is updated to refer to ‘‘seconds since the Epoch’’ rather than ‘‘seconds since 00:00:00 UTC (Coordinated Universal Time), January 1 1970’’ for consistency with other *time* functions.

16547 **NAME**

16548 getegid — get the effective group ID

16549 **SYNOPSIS**

16550
```
#include <unistd.h>
```

16551
```
gid_t getegid(void);
```

16552 **DESCRIPTION**

16553 The *getegid*( ) function shall return the effective group ID of the calling process.

16554 **RETURN VALUE**

16555 The *getegid*( ) function is always successful and no return value is reserved to indicate an error.

16556 **ERRORS**

16557 No errors are defined.

16558 **EXAMPLES**

16559 None.

16560 **APPLICATION USAGE**

16561 None.

16562 **RATIONALE**

16563 None.

16564 **FUTURE DIRECTIONS**

16565 None.

16566 **SEE ALSO**

16567 *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base
16568 Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

16569 **CHANGE HISTORY**

16570 First released in Issue 1. Derived from Issue 1 of the SVID.

16571 **Issue 4**

16572 The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on
16573 XSI-conformant systems.

16574 The **<unistd.h>** header is added to the SYNOPSIS section.

16575 The following change is incorporated for alignment with the ISO POSIX-1 standard:

16576 • The argument list is explicitly defined as **void**.

16577 **Issue 6**

16578 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

16579 The following new requirements on POSIX implementations derive from alignment with the
16580 Single UNIX Specification:

16581 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
16582 required for conforming implementations of previous POSIX specifications, it was not
16583 required for UNIX applications.

16584 **NAME**

16585 getenv — get value of an environment variable

16586 **SYNOPSIS**

16587
```
#include <stdlib.h>
```

16588
```
char *getenv(const char *name);
```

16589 **DESCRIPTION**

16590 CX The functionality described on this reference page is aligned with the ISO C standard. Any
16591 conflict between the requirements described here and the ISO C standard is unintentional. This
16592 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

16593 The *getenv*( ) function shall search the environment of the calling process (see the Base
16594 Definitions volume of IEEE Std. 1003.1-200x, Chapter 8, Environment Variables) for the
16595 environment variable *name* if it exists and return a pointer to the value of the environment
16596 variable. If the specified environment variable cannot be found, a null pointer shall be returned.
16597 The application shall ensure that it does not modify the string pointed to by the *getenv*( )
16598 function. The string pointed to may be overwritten by a subsequent call to *getenv*( ), *setenv*( ),
16599 XSI *unsetenv*( ), or *putenv*( ) but shall not be overwritten by a call to any other function in this volume
16600 of IEEE Std. 1003.1-200x.

16601 If the application modifies *environ* or the pointers to which it points, the behavior of *getenv*( ) is
16602 undefined.

16603 CX The *getenv*( ) function need not be reentrant. A function that is not required to be reentrant is not
16604 required to be thread-safe.

16605 **RETURN VALUE**

16606 Upon successful completion, *getenv*( ) shall return a pointer to a string containing the *value* for
16607 the specified *name*. If the specified name cannot be found in the environment of the calling
16608 process, a null pointer shall be returned.

16609 The return value from *getenv*( ) may point to static data which may be overwritten by
16610 XSI subsequent calls to *getenv*( ), *setenv*( ), *unsetenv*( ), or *putenv*( ).

16611 **ERRORS**

16612 No errors are defined.

16613 **EXAMPLES**

16614 **Getting the Value of an Environment Variable**

16615 The following example gets the value of the *HOME* environment variable.

16616
```
#include <stdlib.h>
```
16617
```
...
```
16618
```
const char *name = "HOME";
char *value;
```

16620
```
value = getenv(name);
```

16621 **APPLICATION USAGE**

16622 None.

16623 **RATIONALE**

16624 The *clearenv*( ) function was considered but rejected. The *putenv*( ) function has now been
16625 included for alignment with the Single UNIX Specification.

The *getenv*( ) function is inherently not reentrant because it returns a value pointing to static data.

Conforming applications are required not to modify *environ* directly, but to use only the functions described here to manipulate the process environment as an abstract object. Thus, the implementation of the environment access functions has complete control over the data structure used to represent the environment (subject to the requirement that *environ* be maintained as a list of strings with embedded equal signs for applications that wish to scan the environment). This constraint allows the implementation to properly manage the memory it allocates, either by using allocated storage for all variables (copying them on the first invocation of *setenv*( ) or *unsetenv*( )), or keeping track of which strings are currently in allocated space and which are not, via a separate table or some other means. This enables the implementation to free any allocated space used by strings (and perhaps the pointers to them) stored in *environ* when *unsetenv*( ) is called. A C runtime start-up procedure (that which invokes *main*( ) and perhaps initializes *environ*) can also initialize a flag indicating that none of the environment has yet been copied to allocated storage, or that the separate table has not yet been initialized.

In fact, for higher performance of *getenv*( ), the implementation could also maintain a separate copy of the environment in a data structure that could be searched much more quickly (such as an indexed hash table, or a binary tree), and update both it and the linear list at *environ* when *setenv*( ) or *unsetenv*( ) is invoked.

Performance of *getenv*( ) can be important for applications which have large numbers of environment variables. Typically, applications like this use the environment as a resource database of user-configurable parameters. The fact that these variables are in the user's shell environment usually means that any other program that uses environment variables (such as *ls*, which attempts to use *COLUMNS*, or really almost any utility (*LANG*, *LC_ALL*, and so on) is similarly slowed down by the linear search through the variables.

An implementation that maintains separate data structures, or even one that manages the memory it consumes, is not currently required as it was thought it would reduce consensus among implementors who do not want to change their historical implementations.

The POSIX Threads Extension states that multi-threaded applications must not modify *environ* directly, and that IEEE Std. 1003.1-200x is providing functions which such applications can use in the future to manipulate the environment in a thread-safe manner. Thus, moving away from application use of *environ* is desirable from that standpoint as well.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *putenv*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 8, Environment Variables

CHANGE HISTORY

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

Issue 4

The DESCRIPTION is updated to indicate that the return string must not be modified by an application, may be overwritten by subsequent calls to *getenv*( ) or *putenv*( ), and is not overwritten by calls to other XSI system interfaces.

A reference to *putenv*( ) has also been added to the APPLICATION USAGE section.

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument *name* is changed from **char*** to **const char***.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

The following changes were made to align with the IEEE P1003.1a draft standard:

- References added to the new *setenv*( ) and *unsetenv*( ) functions.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

16680 **NAME**

16681 geteuid — get the effective user ID

16682 **SYNOPSIS**

16683
```
#include <unistd.h>
```

16684
```
uid_t geteuid(void);
```

16685 **DESCRIPTION**

16686 The *geteuid*( ) function shall return the effective user ID of the calling process.

16687 **RETURN VALUE**

16688 The *geteuid*( ) function is always successful and no return value is reserved to indicate an error.

16689 **ERRORS**

16690 No errors are defined.

16691 **EXAMPLES**

16692 None.

16693 **APPLICATION USAGE**

16694 None.

16695 **RATIONALE**

16696 None.

16697 **FUTURE DIRECTIONS**

16698 None.

16699 **SEE ALSO**

16700 *getegid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base
16701 Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

16702 **CHANGE HISTORY**

16703 First released in Issue 1. Derived from Issue 1 of the SVID.

16704 **Issue 4**

16705 The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on
16706 XSI-conformant systems.

16707 The **<unistd.h>** header is added to the SYNOPSIS section.

16708 The following change is incorporated for alignment with the ISO POSIX-1 standard:

16709 • The argument list is explicitly defined as **void**.

16710 **Issue 6**

16711 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

16712 The following new requirements on POSIX implementations derive from alignment with the
16713 Single UNIX Specification:

16714 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
16715 required for conforming implementations of previous POSIX specifications, it was not
16716 required for UNIX applications.

16717 **NAME**

16718 getgid — get the real group ID

16719 **SYNOPSIS**

16720
```
#include <unistd.h>
```

16721
```
gid_t getgid(void);
```

16722 **DESCRIPTION**

16723 The *getgid*( ) function shall return the real group ID of the calling process.

16724 **RETURN VALUE**

16725 The *getgid*( ) function is always successful and no return value is reserved to indicate an error.

16726 **ERRORS**

16727 No errors are defined.

16728 **EXAMPLES**

16729 None.

16730 **APPLICATION USAGE**

16731 None.

16732 **RATIONALE**

16733 None.

16734 **FUTURE DIRECTIONS**

16735 None.

16736 **SEE ALSO**

16737 *getegid*( ), *geteuid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base
16738 Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

16739 **CHANGE HISTORY**

16740 First released in Issue 1. Derived from Issue 1 of the SVID.

16741 **Issue 4**

16742 The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on
16743 XSI-conformant systems.

16744 The **<unistd.h>** header is added to the SYNOPSIS section.

16745 The following change is incorporated for alignment with the ISO POSIX-1 standard:

16746 - The argument list is explicitly defined as **void**.

16747 **Issue 6**

16748 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

16749 The following new requirements on POSIX implementations derive from alignment with the
16750 Single UNIX Specification:

16751 - The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
16752 required for conforming implementations of previous POSIX specifications, it was not
16753 required for UNIX applications.

16754 **NAME**

16755 getgrent — get the group database entry

16756 **SYNOPSIS**

16757 XSI
```
#include <grp.h>
```

16758
```
struct group *getgrent(void);
```

16759

16760 **DESCRIPTION**

16761 Refer to *endgrent*( ).

16762 **NAME**

16763 getgrgid, getgrgid_r — get group database entry for a group ID

16764 **SYNOPSIS**

16765 ```
#include <grp.h>
```

16766 ```
struct group *getgrgid(gid_t gid);
```

16767 TSF ```
int getgrgid_r(gid_t gid, struct group *grp, char *buffer,
16768   size_t bufsize, struct group **result);
```

16769

16770 **DESCRIPTION**

16771 The *getgrgid*( ) function shall search the group database for an entry with a matching *gid*.

16772 The *getgrgid*( ) function need not be reentrant. A function that is not required to be reentrant is
16773 not required to be thread-safe.

16774 TSF The *getgrgid_r*( ) function updates the **group** structure pointed to by *grp* and stores a pointer to
16775 that structure at the location pointed to by *result*. The structure contains an entry from the
16776 group database with a matching *gid*. Storage referenced by the group structure is allocated from
16777 the memory provided with the *buffer* parameter, which is *bufsize* characters in size. The
16778 maximum size needed for this buffer can be determined with the {_SC_GETGR_R_SIZE_MAX}
16779 *sysconf*( ) parameter. A NULL pointer is returned at the location pointed to by *result* on error or if
16780 the requested entry is not found.

16781 **RETURN VALUE**

16782 Upon successful completion, *getgrgid*( ) shall return a pointer to a **struct group** with the structure
16783 defined in **<grp.h>** with a matching entry if one is found. The *getgrgid*( ) function shall return a
16784 null pointer if either the requested entry was not found, or an error occurred. On error, *errno* |
16785 shall be set to indicate the error. |

16786 The return value may point to a static area which is overwritten by a subsequent call to
16787 *getgrent*( ), *getgrgid*( ), or *getgrnam*( ).

16788 TSF If successful, the *getgrgid_r*( ) function shall return zero; otherwise, an error number shall be
16789 returned to indicate the error.

16790 **ERRORS**

16791 The *getgrgid*( ) and *getgrgid_r*( ) functions may fail if: |

16792 [EIO] An I/O error has occurred. |

16793 [EINTR] A signal was caught during *getgrgid*( ). |

16794 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process. |

16795 [ENFILE] The maximum allowable number of files is currently open in the system. |

16796 TSF The *getgrgid_r*( ) function may fail if:

16797 TSF [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to |
16798 be 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.

```
#include <sys/types.h>
#include <grp.h>
#include <stdio.h>
...
struct stat statbuf;
struct group *grp;
...
if ((grp = getgrgid(statbuf.st_gid)) != NULL)
    printf(" %-8.8s", grp->gr_name);
else
    printf(" %-8d", statbuf.st_gid);
...
```

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 4

The DESCRIPTION is clarified.

In the RETURN VALUE section, the reference to the setting of *errno* is marked as an extension.

The errors [EIO], [EINTR], [EMFILE], and [ENFILE] are marked as extensions.

A note is added to the APPLICATION USAGE section advising how applications should check for errors.

The **<sys/types.h>** header is added as optional (OH); this header need not be included on XSI-conformant systems.

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 corrigenda item U028/3 has been 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 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 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.

16859 **NAME**

16860 getgrnam, getgrnam_r — search group database for a name

16861 **SYNOPSIS**

16862 ```
#include <grp.h>

struct group *getgrnam(const char *name);
TSF int getgrnam_r(const char *name, struct group *grp, char *buffer,
        size_t bufsize, struct group **result);
```

16866

16867 **DESCRIPTION**

16868 The *getgrnam*( ) function shall search the group database for an entry with a matching *name*.

16869 The *getgrnam*( ) function need not be reentrant. A function that is not required to be reentrant is
16870 not required to be thread-safe.

16871 TSF The *getgrnam_r*( ) function updates the **group** structure pointed to by *grp* and stores a pointer to
16872 that structure at the location pointed to by *result*. The structure contains an entry from the
16873 group database with a matching *gid* or *name*. Storage referenced by the group structure is
16874 allocated from the memory provided with the *buffer* parameter, which is *bufsize* characters in
16875 size. The maximum size needed for this buffer can be determined with the
16876 {_SC_GETGR_R_SIZE_MAX} *sysconf*( ) parameter. A NULL pointer is returned at the location
16877 pointed to by *result* on error or if the requested entry is not found.

16878 **RETURN VALUE**

16879 The *getgrnam*( ) function shall return a pointer to a **struct group** with the structure defined in
16880 **<grp.h>** with a matching entry if one is found. The *getgrnam*( ) function shall return a null
16881 pointer if either the requested entry was not found, or an error occurred. On error, *errno* shall be |
16882 set to indicate the error. |

16883 The return value may point to a static area which is overwritten by a subsequent call to
16884 *getgrent*( ), *getgrgid*( ), or *getgrnam*( ).

16885 TSF If successful, the *getgrnam_r*( ) function shall return zero; otherwise, an error number shall be
16886 returned to indicate the error.

16887 **ERRORS**

16888 The *getgrnam*( ) and *getgrnam_r*( ) functions may fail if: |

16889 [EIO] An I/O error has occurred. |

16890 [EINTR] A signal was caught during *getgrnam*( ). |

16891 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process. |

16892 [ENFILE] The maximum allowable number of files is currently open in the system. |

16893 The *getgrnam_r*( ) function may fail if:

16894 TSF [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to |
16895 be referenced by the resulting **group** structure.
```

16896 **EXAMPLES**

16897 None.

16898 **APPLICATION USAGE**

16899 Applications wishing to check for error situations should set *errno* to 0 before calling *getgrnam*().
16900 If *errno* is set on return, an error occurred.

16901 The *getgrnam_r*() function is thread-safe and shall return values in a user-supplied buffer instead
16902 of possibly using a static data area that may be overwritten by each call.

16903 **RATIONALE**

16904 None.

16905 **FUTURE DIRECTIONS**

16906 None.

16907 **SEE ALSO**

16908 *endgrent*(), *getgrgid*(), the Base Definitions volume of IEEE Std. 1003.1-200x, **<grp.h>**, **<limits.h>**, |
16909 **<sys/types.h>**

16910 **CHANGE HISTORY**

16911 First released in Issue 1. Derived from System V Release 2.0. |

16912 **Issue 4**

16913 The DESCRIPTION is clarified.

16914 The **<sys/types.h>** header is added as optional (OH); this header need not be included on XSI-
16915 conformant systems.

16916 In the RETURN VALUE section, reference to the setting of *errno* is marked as an extension.

16917 The errors [EIO], [EINTR], [EMFILE], and [ENFILE] are marked as extensions.

16918 A note is added to the APPLICATION USAGE section advising how applications should check
16919 for errors.

16920 The following change is incorporated for alignment with the ISO POSIX-1 standard:

16921 • The type of argument *name* is changed from **char*** to **const char***.

16922 **Issue 5**

16923 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
16924 VALUE section.

16925 The *getgrnam_r*() function is included for alignment with the POSIX Threads Extension.

16926 A note indicating that the *getgrnam*() function need not be reentrant is added to the
16927 DESCRIPTION.

16928 **Issue 6**

16929 The *getgrnam_r*() function is marked as part of the Thread-Safe Functions option. |

16930 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

16931 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

16932 The following new requirements on POSIX implementations derive from alignment with the
16933 Single UNIX Specification:

16934 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
16935 required for conforming implementations of previous POSIX specifications, it was not
16936 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.

NAME

getgroups — get supplementary group IDs

SYNOPSIS

```
#include <unistd.h>

int getgroups(int gidsetsize, gid_t grouplist[]);
```

DESCRIPTION

The *getgroups*( ) function fills in the array *grouplist* with the current supplementary group IDs of the calling process. It is implementation-defined whether *getgroups*( ) also returns the effective group ID in the *grouplist* array.

The *gidsetsize* argument specifies the number of elements in the array *grouplist*. The actual number of group IDs stored in the array is returned. The values of array entries with indices greater than or equal to the value returned are undefined.

If *gidsetsize* is 0, *getgroups*( ) shall return the number of group IDs that it would otherwise return without modifying the array pointed to by *grouplist*.

If the effective group ID of the process is returned with the supplementary group IDs, the value returned shall always be greater than or equal to one and less than or equal to the value of {NGROUPS_MAX}+1.

RETURN VALUE

Upon successful completion, the number of supplementary group IDs shall be returned. A return value of –1 indicates failure and *errno* shall be set to indicate the error.

ERRORS

The *getgroups*( ) function shall fail if:

[EINVAL] The *gidsetsize* argument is non-zero and less than the number of group IDs that would have been returned.

EXAMPLES

Getting the Supplementary Group IDs of the Calling Process

The following example places the current supplementary group IDs of the calling process into the *group* array.

```
#include <sys/types.h>
#include <unistd.h>
...
gid_t *group;
int nogroups;
long ngroups_max;

ngroups_max = sysconf(_SC_NGROUPS_MAX);
group = (gid_t *)malloc(ngroups_max *sizeof(gid_t));

ngroups = getgroups(ngroups_max, group);
```

APPLICATION USAGE

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 result.)

Applications 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.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

The **<unistd.h>** header is added to the SYNOPSIS section.

The following change is incorporated for alignment with the FIPS requirements:

- A return value of 0 is no longer permitted, because {NGROUPS_MAX} cannot be 0.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

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.
- 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:

- Explanation added that the effective group ID may be included in the supplementary group list.

17021 **NAME**

17022 gethostbyaddr (**LEGACY**), gethostbyname (**LEGACY**), getipnodebyaddr, getipnodebyname, —
17023 network host database functions

17024 **SYNOPSIS**

17025 ```
#include <netdb.h>

struct hostent *gethostbyaddr(const void *addr, socklen_t len,
    int type);
struct hostent *gethostbyname(const char *name);
struct hostent *getipnodebyaddr(const void *restrict addr, socklen_t len,
    int type, int *restrict error_num);
struct hostent *getipnodebyname(const char *name, int type, int flags,
    int *error_num);
```

17033 **DESCRIPTION**

17034 These functions enable applications to retrieve information about hosts. This information is
17035 considered to be stored in a database that can be accessed sequentially or randomly.
17036 Implementation of this database is unspecified.

17037 **Note:** In many cases it is implemented by the Domain Name System, as documented in
17038 RFC 1034, RFC 1035, and RFC 1886.

17039 Entries are returned in **hostent** structures.

17040 The *gethostbyaddr*( ) function shall return an entry containing addresses of address family *type* for
17041 the host with address *addr*. *len* contains the length of the address pointed to by *addr*. The
17042 *gethostbyaddr*( ) function need not be reentrant. A function that is not required to be reentrant is
17043 not required to be thread-safe.

17044 The *gethostbyname*( ) function shall return an entry containing addresses of address family
17045 AF_INET for the host with name *name*. The *gethostbyname*( ) function need not be reentrant. A
17046 function that is not required to be reentrant is not required to be thread-safe.

17047 The *getipnodebyaddr*( ) function shall return the entry containing addresses of address family *type*
17048 for the host with address *addr*, opening a connection to the database if necessary. The *len*
17049 argument contains the length of the address pointed to by *addr*. If an error occurs, the
17050 appropriate error code is returned in *error_num*. The *getipnodebyaddr*( ) function is thread-safe.

17051 The *getipnodebyname*( ) function shall return the entry containing addresses of address family
17052 *type* for the host with name *name*, opening a connection to the database if necessary. The *flags*
17053 argument affects what information is returned. If an error occurs, the appropriate error code is
17054 returned in *error_num*. The *getipnodebyname*( ) function is thread-safe.

17055 The *addr* argument of *gethostbyaddr*( ) or *getipnodebyaddr*( ) shall be an **in_addr** structure when
17056 IP6 *type* is AF_INET, and shall be an **in6_addr** structure when *type* is AF_INET6. It contains a binary
17057 format (that is, not null-terminated) address in network byte order. The *gethostbyaddr*( ) function
17058 is not guaranteed to return addresses of address families other than AF_INET, even when such
17059 addresses exist in the database.

17060 If *gethostbyaddr*( ) or *getipnodebyaddr*( ) returns a record, then its *h_addrtype* field is the same as the
17061 *type* argument that was passed to the function, and its *h_addr_list* field lists a single address that
17062 IP6 is a copy of the *addr* argument that was passed to the function. If *type* is AF_INET6 and *addr* is
17063 an IPv4-mapped IPv6 address or an IPv4-compatible IPv6 address, then the *h_name* and *h_aliases*
17064 fields are those that would have been returned for address family AF_INET and address equal to
17065 the last four bytes of *addr*.
```

If *gethostbyaddr*( ) or *getipnodebyaddr*( ) are called with *addr* containing the IPv6 unspecified address (all bytes zero), then no query is performed and the function fails with [HOST_NOT_FOUND].

The *name* argument of *getipnodebyname*( ) shall be either a node name or a numeric address string. For IPv4, a numeric address string shall be in the dotted-decimal notation described in *inet_addr*( ). IP6 For IPv6, a numeric address string shall be in one of the standard IPv6 text forms described in *inet_ntop*( ) The *name* argument of *gethostbyname*( ) shall be a node name; the behavior of *gethostbyname*( ) when passed a numeric address string is unspecified.

IP6 If *name* is a dotted-decimal IPv4 address and *af* equals AF_INET, or *name* is an IPv6 hex address and *af* equals AF_INET6, the members of the returned **hostent** structure are as follows:

h_name Points to a copy of the *name* argument.

h_aliases A NULL pointer.

h_addrtype A copy of the *type* argument.

IP6 *h_length* Either 4 (for AF_INET) or 16 (for AF_INET6).

IP6 *h_addr_list*[0] A pointer to the 4-byte or 16-byte binary address.

h_addr_list[1] A NULL pointer.

IP6 If *name* is a dotted-decimal IPv4 address and *af* equals AF_INET6 and AI_V4MAPPED is set in *flags*, an IPv4-mapped IPv6 address is returned, and the members are as follows:

h_name Points to an IPv6 hex address containing the IPv4-mapped IPv6 address.

h_aliases A NULL pointer.

h_addrtype AF_INET6.

h_length 16.

h_addr_list[0] A pointer to the 16-byte binary address.

h_addr_list[1] A NULL pointer.

If *name* is a dotted-decimal IPv4 address and *af* equals AF_INET6 and AI_V4MAPPED is not set, then NULL is returned with [HOST_NOT_FOUND].

It is an error when *name* is an IPv6 hex address and *af* equals AF_INET. The function's return value is a NULL pointer with the [HOST_NOT_FOUND] error.

If *name* is not a numeric address string and is an alias for a valid host name, then *gethostbyname*( ) or *getipnodebyname*( ) return information about the host name to which the alias refers, and *name* is included in the list of aliases returned.

If *name* is a node name, then operation of the *getipnodebyname*( ) function is modified by the value of the *flags* argument, as follows:

- If *flags* is 0 and *type* is AF_INET, then a query is made for IPv4 addresses. If it is successful, the IPv4 addresses are returned and the *h_length* member of the **hostent** structure shall have a value of 4. Otherwise, the function shall return a NULL pointer. IP6

- If *flags* is 0 and if *type* is AF_INET6, then a query is made for IPv6 addresses. If it is successful, the IPv6 addresses are returned and the *h_length* member of the **hostent** structure shall have a value of 16. If unsuccessful, the function shall return a NULL pointer.

- If the AI_V4MAPPED flag is set and *type* is AF_INET6, then a query is made for IPv6 addresses. If it is successful, the IPv6 addresses are returned, and no query is made for IPv4

addresses. If it is not successful, a query is made for IPv4 addresses and any found are returned as IPv4-mapped IPv6 addresses. *h_length* shall have a value of 16 in either case of addresses being returned. The AI_V4MAPPED flag is ignored unless *type* is AF_INET6.

- If the AI_ALL and AI_V4MAPPED flags are both set and *type* is AF_INET6, then a query is made for IPv6 addresses, and any found are returned. Another query is then made for IPv4 addresses, and any found are returned as IPv4-mapped IPv6 addresses, and *h_length* is 16. Only if both queries fail does the function return a NULL pointer. This flag is ignored unless type is AF_INET6.
- The AI_ADDRCONFIG flag specifies that a query for IPv6 addresses should be made only if the node has at least one IPv6 source address configured, and that a query for IPv4 addresses should be made only if the node has at least one IPv4 source address configured.
- If the AI_V4MAPPED and AI_ADDRCONFIG flags are both set and *type* is AF_INET6, then:

  — If the node has at least one IPv6 source address configured, a query is made for IPv6 addresses.

  — If it is successful, the IPv6 addresses are returned and no query is made for IPv4 addresses.

  — If the node has no IPv6 source address configured, or if the query for IPv6 addresses is not successful, then if the node has at least one IPv4 source address configured, a query is made for IPv4 addresses and any found are returned as IPv4-mapped IPv6 addresses.

  h_length shall have a value of 16 in either case of addresses being returned.
- Macro AI_DEFAULT is defined as the logical OR of AI_V4MAPPED and AI_ADDRCONFIG.

Note: It is intended that setting *flags* to AI_DEFAULT be appropriate for most applications.

RETURN VALUE

Upon successful completion, these functions shall return a pointer to a **hostent** structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found.

Upon unsuccessful completion, *getipnodebyaddr*( ) and *getipnodebyname*( ) shall set their *error_num* argument to indicate the error, while *gethostbyaddr*( ) and *gethostbyname*( ) shall set *h_errno* to indicate it.

ERRORS

These functions shall fail in the following cases. The *getipnodebyaddr*( ) and *getipnodebyname*( ) functions shall return the value shown in the list below in *error_num*; the *gethostbyaddr*( ) and *gethostbyname*( ) functions shall set *h_errno* to that value. Any changes to *errno* are unspecified.

[HOST_NOT_FOUND]
No such host is known.

[NO_DATA] The server recognized the request and the name, but no address is available. Another type of request to the name server for the domain might return an answer.

[NO_RECOVERY]
An unexpected server failure occurred which cannot be recovered.

[TRY_AGAIN] A temporary and possibly transient error occurred, such as a failure of a server to respond.

17150 **EXAMPLES**

17151 None.

17152 **APPLICATION USAGE**

17153 The **hostent** structure returned by *getipnodebyaddr*( ) and *getipnodebyname*( ), and any structures
17154 pointed to from those structures, are dynamically allocated. Applications should call
17155 *freehostent*( ) to free the memory used by these structures.

17156 The *gethostbyaddr*( ), and *gethostbyname*( ) functions may return pointers to static data, which may
17157 be overwritten by subsequent calls to any of these functions. Applications shall not call
17158 *freehostent*( ) for this area.

17159 The *getipnodebyaddr*( ) function is preferred over the *gethostbyaddr*( ) function.

17160 The *getipnodebyname*( ) function is preferred over the *gethostbyname*( ) function.

17161 **RATIONALE**

17162 None.

17163 **FUTURE DIRECTIONS**

17164 The *gethostbyaddr*( ) and *gethostbyname*( ) functions may be withdrawn in a future version.

17165 **SEE ALSO**

17166 *endhostent*( ), *freehostent*( ), *endservent*( ), *inet_addr*( ), the Base Definitions volume of |
17167 IEEE Std. 1003.1-200x, **<netdb.h>** |

17168 **CHANGE HISTORY**

17169 First released in Issue 6. Derived from the XNS, Issue 5.2 specification. |

17170 The **restrict** keyword is added to the *getipnodebyaddr*( ) prototype for alignment with the |
17171 ISO/IEC 9899:1999 standard. |

17172 **NAME**

17173 gethostbyname — network host database functions

17174 **SYNOPSIS**

17175 `#include <netdb.h>`

17176 `struct hostent *gethostbyname(const char *`*`name`*`);`

17177 **DESCRIPTION**

17178 Refer to *gethostbyaddr*( ).

17179 **NAME**

17180 gethostent — network host database functions

17181 **SYNOPSIS**

17182 `#include <netdb.h>`

17183 `struct hostent *gethostent(void);`

17184 **DESCRIPTION**

17185 Refer to *endhostent*( ).

17186 **NAME**

17187 gethostid — get an identifier for the current host

17188 **SYNOPSIS**

17189 XSI
```
#include <unistd.h>
```
17190
```
long gethostid(void);
```
17191

17192 **DESCRIPTION**

17193 The *gethostid*( ) function retrieves a 32-bit identifier for the current host.

17194 **RETURN VALUE**

17195 Upon successful completion, *gethostid*( ) shall return an identifier for the current host.

17196 **ERRORS**

17197 No errors are defined.

17198 **EXAMPLES**

17199 None.

17200 **APPLICATION USAGE**

17201 This volume of IEEE Std. 1003.1-200x does not define the domain in which the return value is
17202 unique.

17203 **RATIONALE**

17204 None.

17205 **FUTURE DIRECTIONS**

17206 None.

17207 **SEE ALSO**

17208 *random*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>**

17209 **CHANGE HISTORY**

17210 First released in Issue 4, Version 2.

17211 **Issue 5**

17212 Moved from X/OPEN UNIX extension to BASE.

17213 **NAME**

17214 gethostname — get name of current host

17215 **SYNOPSIS**

17216 `#include <unistd.h>`

17217 `int gethostname(char *`*`name`*`, socklen_t `*`namelen`*`);`

17218 **DESCRIPTION**

17219 The *gethostname*( ) function shall return the standard host name for the current machine. The
17220 *namelen* argument shall specify the size of the array pointed to by the *name* argument. The
17221 returned name shall be null-terminated, except that if *namelen* is an insufficient length to hold
17222 the host name, then the returned name shall be truncated and it is unspecified whether the
17223 returned name is null-terminated.

17224 Host names are limited to 255 bytes.

17225 **RETURN VALUE**

17226 Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned.

17227 **ERRORS**

17228 No errors are defined.

17229 **EXAMPLES**

17230 None.

17231 **APPLICATION USAGE**

17232 None.

17233 **RATIONALE**

17234 None.

17235 **FUTURE DIRECTIONS**

17236 None.

17237 **SEE ALSO**

17238 *gethostid*( ), *uname*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>** |

17239 **CHANGE HISTORY**

17240 First released in Issue 6. Derived from the XNS, Issue 5.2 specification. |

17241 **NAME**

17242 getipnodebyaddr — network host database functions

17243 **SYNOPSIS**

17244 `#include <netdb.h>`

17245 `struct hostent *getipnodebyaddr(const void *restrict addr, socklen_t len,`
17246 `int type, int *restrict error_num);`

17247 **DESCRIPTION**

17248 Refer to *gethostbyaddr*( ).

17249 **NAME**

17250 getipnodebyname — network host database functions

17251 **SYNOPSIS**

17252 `#include <netdb.h>`

17253 `struct hostent *getipnodebyname(const char *`*`name`*`, int `*`type`*`, int `*`flags`*`,`
17254 `int *`*`error_num`*`);`

17255 **DESCRIPTION**

17256 Refer to *gethostbyaddr*( ).

NAME

getitimer, setitimer — get or set value of interval timer

SYNOPSIS

XSI

```
#include <sys/time.h>

int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *restrict value,
    struct itimerval *restrict ovalue);
```

DESCRIPTION

The *getitimer*( ) function shall store the current value of the timer specified by *which* into the structure pointed to by *value*. The *setitimer*( ) function shall set the timer specified by *which* to the value specified in the structure pointed to by *value*, and if *ovalue* is not a null pointer, stores the previous value of the timer in the structure pointed to by *ovalue*.

A timer value is defined by the **itimerval** structure, specified in **<sys/time.h>**. If *it_value* is non-zero, it shall indicate the time to the next timer expiration. If *it_interval* is non-zero, it shall specify a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to 0 shall disable a timer, regardless of the value of *it_interval*. Setting *it_interval* to 0 shall disable a timer after its next expiration (assuming *it_value* is non-zero).

Implementations may place limitations on the granularity of timer values. For each interval timer, if the requested timer value requires a finer granularity than the implementation supports, the actual timer value shall be rounded up to the next supported value.

An XSI-conforming implementation provides each process with at least three interval timers, which are indicated by the *which* argument:

ITIMER_REAL — Decrements in real time. A SIGALRM signal is delivered when this timer expires.

ITIMER_VIRTUAL — Decrements in process virtual time. It runs only when the process is executing. A SIGVTALRM signal is delivered when it expires.

ITIMER_PROF — Decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by interpreters in statistically profiling the execution of interpreted programs. Each time the ITIMER_PROF timer expires, the SIGPROF signal is delivered.

The interaction between *setitimer*( ) and any of *alarm*( ), *sleep*( ), or *usleep*( ) is unspecified.

RETURN VALUE

Upon successful completion, *getitimer*( ) or *setitimer*( ) shall return 0; otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *setitimer*( ) function shall fail if:

[EINVAL] The *value* argument is not in canonical form. (In canonical form, the number of microseconds is a non-negative integer less than 1,000,000 and the number of seconds is a non-negative integer.)

The *getitimer*( ) and *setitimer*( ) functions may fail if:

[EINVAL] The *which* argument is not recognized.

17299 **EXAMPLES**

17300 None.

17301 **APPLICATION USAGE**

17302 None.

17303 **RATIONALE**

17304 None.

17305 **FUTURE DIRECTIONS**

17306 None.

17307 **SEE ALSO**

17308 *alarm*( ), *sleep*( ), *timer_getoverrun*( ), *ualarm*( ), *usleep*( ), the Base Definitions volume of |
17309 IEEE Std. 1003.1-200x, **<signal.h>**, **<sys/time.h>** |

17310 **CHANGE HISTORY**

17311 First released in Issue 4, Version 2.

17312 **Issue 5**

17313 Moved from X/OPEN UNIX extension to BASE. |

17314 **Issue 6** |

17315 The **restrict** keyword is added to the *setitimer*( ) prototype for alignment with the |
17316 ISO/IEC 9899:1999 standard. |

17317 **NAME**

17318 getlogin, getlogin_r — get login name

17319 **SYNOPSIS**

17320 `#include <unistd.h>`

17321 `char *getlogin(void);`

17322 TSF `int getlogin_r(char *`*`name`*`, size_t `*`namesize`*`);`

17323

17324 **DESCRIPTION**

17325 The *getlogin*( ) function shall return a pointer to a string containing the user name associated by
17326 the login activity with the controlling terminal of the current process. If *getlogin*( ) returns a non-
17327 null pointer, then that pointer points to the name that the user logged in under, even if there are
17328 several login names with the same user ID.

17329 The *getlogin*( ) function need not be reentrant. A function that is not required to be reentrant is
17330 not required to be thread-safe.

17331 TSF The *getlogin_r*( ) function puts the name associated by the login activity with the controlling
17332 terminal of the current process in the character array pointed to by *name*. The array is *namesize*
17333 characters long and should have space for the name and the terminating null character. The
17334 maximum size of the login name is {LOGIN_NAME_MAX}.

17335 If *getlogin_r*( ) is successful, *name* points to the name the user used at login, even if there are
17336 several login names with the same user ID.

17337 **RETURN VALUE**

17338 Upon successful completion, *getlogin*( ) shall return a pointer to the login name or a null pointer
17339 if the user's login name cannot be found. Otherwise, it shall return a null pointer and set *errno* to
17340 indicate the error.

17341 The return value from *getlogin*( ) may point to static data whose content is overwritten by each
17342 call.

17343 TSF If successful, the *getlogin_r*( ) function shall return zero; otherwise, an error number shall be
17344 returned to indicate the error.

17345 **ERRORS**

17346 The *getlogin*( ) and *getlogin_r*( ) functions may fail if:

17347 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

17348 [ENFILE] The maximum allowable number of files is currently open in the system.

17349 [ENXIO] The calling process has no controlling terminal.

17350 The *getlogin_r*( ) function may fail if:

17351 TSF [ERANGE] The value of *namesize* is smaller than the length of the string to be returned
17352 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.

```
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
...
char *lgn;
struct passwd *pw;
...
if ((lgn = getlogin()) == NULL || (pw = getpwnam(lgn)) == NULL) {
    fprintf(stderr, "Get of user information failed.\n"); exit(1);
    }
```

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 shall return 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>**

17395 **CHANGE HISTORY**

17396 First released in Issue 1. Derived from System V Release 2.0.

17397 **Issue 4**

17398 The **<unistd.h>** header is added to the SYNOPSIS section.

17399 In the RETURN VALUE section, reference to the setting of *errno* is marked as an extension.

17400 The behavior of the function when the login name cannot be found is included in the RETURN
17401 VALUE section instead of the DESCRIPTION.

17402 The errors [EMFILE], [ENFILE], and [ENXIO] are marked as extensions.

17403 The APPLICATION USAGE section is changed to refer to *getpwuid*( ) rather than *cuserid*( ), which
17404 may be withdrawn in a future version.

17405 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

17406 - The argument list is explicitly defined as **void**.

17407 - The DESCRIPTION is updated to state explicitly that the return value is a pointer to a string
17408 giving the user name, rather than simply a pointer to the user name as stated in previous
17409 issues.

17410 **Issue 5**

17411 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
17412 VALUE section.

17413 The *getlogin_r*( ) function is included for alignment with the POSIX Threads Extension.

17414 A note indicating that the *getlogin*( ) function need not be reentrant is added to the
17415 DESCRIPTION.

17416 **Issue 6**

17417 The *getlogin_r*( ) function is marked as part of the Thread-Safe Functions option.

17418 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

17419 The following new requirements on POSIX implementations derive from alignment with the
17420 Single UNIX Specification:

17421 - In the RETURN VALUE section, the requirement to set *errno* on error is added.

17422 - The [EMFILE], [ENFILE], and [ENXIO] optional error conditions are added.

17423 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
17424 its avoidance of possibly using a static data area.

17425 **NAME**

17426 getmsg, getpmsg — receive next message from a STREAMS file (**STREAMS**)

17427 **SYNOPSIS**

17428 XSR
```
#include <stropts.h>

int getmsg(int fildes, struct strbuf *restrict ctlptr,
    struct strbuf *restrict dataptr, int *restrict flagsp);
int getpmsg(int fildes, struct strbuf *restrict ctlptr,
    struct strbuf *restrict dataptr, int *restrict bandp,
    int *restrict flagsp);
```

17435 **DESCRIPTION**

17436 The *getmsg*( ) function shall retrieve the contents of a message located at the head of the
17437 STREAM head read queue associated with a STREAMS file and place the contents into one or
17438 more buffers. The message contains either a data part, a control part, or both. The data and
17439 control parts of the message are placed into separate buffers, as described below. The semantics
17440 of each part are defined by the originator of the message.

17441 The *getpmsg*( ) function does the same thing as *getmsg*( ), but provides finer control over the
17442 priority of the messages received. Except where noted, all requirements on *getmsg*( ) also pertain
17443 to *getpmsg*( ).

17444 The *fildes* argument specifies a file descriptor referencing a STREAMS-based file.

17445 The *ctlptr* and *dataptr* arguments each point to a **strbuf** structure, in which the *buf* member points
17446 to a buffer in which the data or control information is to be placed, and the *maxlen* member
17447 indicates the maximum number of bytes this buffer can hold. On return, the *len* member
17448 contains the number of bytes of data or control information actually received. The *len* member is
17449 set to 0 if there is a zero-length control or data part and *len* is set to –1 if no data or control
17450 information is present in the message.

17451 When *getmsg*( ) is called, *flagsp* should point to an integer that indicates the type of message the
17452 process is able to receive. This is described further below.

17453 The *ctlptr* argument is used to hold the control part of the message, and *dataptr* is used to hold
17454 the data part of the message. If *ctlptr* (or *dataptr*) is a null pointer or the *maxlen* member is –1, the
17455 control (or data) part of the message is not processed and is left on the STREAM head read
17456 queue, and if the *ctlptr* (or *dataptr*) is not a null pointer, *len* is set to –1. If the *maxlen* member is
17457 set to 0 and there is a zero-length control (or data) part, that zero-length part is removed from
17458 the read queue and *len* is set to 0. If the *maxlen* member is set to 0 and there are more than 0 bytes
17459 of control (or data) information, that information is left on the read queue and *len* is set to 0. If
17460 the *maxlen* member in *ctlptr* (or *dataptr*) is less than the control (or data) part of the message,
17461 *maxlen* bytes are retrieved. In this case, the remainder of the message is left on the STREAM head
17462 read queue and a non-zero return value is provided.

17463 By default, *getmsg*( ) processes the first available message on the STREAM head read queue.
17464 However, a process may choose to retrieve only high-priority messages by setting the integer
17465 pointed to by *flagsp* to RS_HIPRI. In this case, *getmsg*( ) shall only process the next message if it is
17466 a high-priority message. When the integer pointed to by *flagsp* is 0, any message shall be
17467 retrieved. In this case, on return, the integer pointed to by *flagsp* is set to RS_HIPRI if a high-
17468 priority message was retrieved, or 0 otherwise.

17469 For *getpmsg*( ), the flags are different. The *flagsp* argument points to a bitmask with the following
17470 mutually-exclusive flags defined: MSG_HIPRI, MSG_BAND, and MSG_ANY. Like *getmsg*( ),
17471 *getpmsg*( ) processes the first available message on the STREAM head read queue. A process may

choose to retrieve only high-priority messages by setting the integer pointed to by *flagsp* to MSG_HIPRI and the integer pointed to by *bandp* to 0. In this case, *getpmsg*( ) shall only process the next message if it is a high-priority message. In a similar manner, a process may choose to retrieve a message from a particular priority band by setting the integer pointed to by *flagsp* to MSG_BAND and the integer pointed to by *bandp* to the priority band of interest. In this case, *getpmsg*( ) shall only process the next message if it is in a priority band equal to, or greater than, the integer pointed to by *bandp*, or if it is a high-priority message. If a process just wants to get the first message off the queue, the integer pointed to by *flagsp* should be set to MSG_ANY and the integer pointed to by *bandp* should be set to 0. On return, if the message retrieved was a high-priority message, the integer pointed to by *flagsp* is set to MSG_HIPRI and the integer pointed to by *bandp* shall be set to 0. Otherwise, the integer pointed to by *flagsp* shall be set to MSG_BAND and the integer pointed to by *bandp* shall be set to the priority band of the message.

If O_NONBLOCK is not set, *getmsg*( ) and *getpmsg*( ) shall block until a message of the type specified by *flagsp* is available at the front of the STREAM head read queue. If O_NONBLOCK is set and a message of the specified type is not present at the front of the read queue, *getmsg*( ) and *getpmsg*( ) shall fail and set *errno* to [EAGAIN].

If a hangup occurs on the STREAM from which messages are retrieved, *getmsg*( ) and *getpmsg*( ) continue to operate normally, as described above, until the STREAM head read queue is empty. Thereafter, they return 0 in the *len* members of *ctlptr* and *dataptr*.

RETURN VALUE

Upon successful completion, *getmsg*( ) and *getpmsg*( ) shall return a non-negative value. A value of 0 indicates that a full message was read successfully. A return value of MORECTL indicates that more control information is waiting for retrieval. A return value of MOREDATA indicates that more data is waiting for retrieval. A return value of the bitwise-logical OR of MORECTL and MOREDATA indicates that both types of information remain. Subsequent *getmsg*( ) and *getpmsg*( ) calls shall retrieve the remainder of the message. However, if a message of higher priority has come in on the STREAM head read queue, the next call to *getmsg*( ) or *getpmsg*( ) shall retrieve that higher-priority message before retrieving the remainder of the previous message.

If the high priority control part of the message is consumed, the message shall be placed back on the queue as a normal message of band 0. Subsequent *getmsg*( ) and *getpmsg*( ) calls shall retrieve the remainder of the message. If, however, a priority message arrives or already exists on the STREAM head, the subsequent call to *getmsg*( ) or *getpmsg*( ) retrieves the higher-priority message before retrieving the remainder of the message that was put back.

Upon failure, *getmsg*( ) and *getpmsg*( ) shall return –1 and set *errno* to indicate the error.

ERRORS

The *getmsg*( ) and *getpmsg*( ) functions shall fail if:

[EAGAIN] The O_NONBLOCK flag is set and no messages are available.

[EBADF] The *fildes* argument is not a valid file descriptor open for reading.

[EBADMSG] The queued message to be read is not valid for *getmsg*( ) or *getpmsg*( ) or a pending file descriptor is at the STREAM head.

[EINTR] A signal was caught during *getmsg*( ) or *getpmsg*( ).

[EINVAL] An illegal value was specified by *flagsp*, or the STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

17517 [ENOSTR] A STREAM is not associated with *fildes*.

17518 In addition, *getmsg*( ) and *getpmsg*( ) shall fail if the STREAM head had processed an
17519 asynchronous error before the call. In this case, the value of *errno* does not reflect the result of
17520 *getmsg*( ) or *getpmsg*( ) but reflects the prior error.

17521 **EXAMPLES**

17522 **Getting Any Message**

17523 In the following example, the value of *fd* is assumed to refer to an open STREAMS file. The call
17524 to *getmsg*( ) retrieves any available message on the associated STREAM-head read queue,
17525 returning control and data information to the buffers pointed to by *ctrlbuf* and *databuf*,
17526 respectively.

```
17527 #include <stropts.h>
17528 ...
17529 int fd;
17530 char ctrlbuf[128];
17531 char databuf[512];
17532 struct strbuf ctrl;
17533 struct strbuf data;
17534 int flags = 0;
17535 int ret;

17536 ctrl.buf = ctrlbuf;
17537 ctrl.maxlen = sizeof(ctrlbuf);

17538 data.buf = databuf;
17539 data.maxlen = sizeof(databuf);

17540 ret = getmsg (fd, &ctrl, &data, &flags);
```

17541 **Getting the First Message off the Queue**

17542 In the following example, the call to *getpmsg*( ) retrieves the first available message on the
17543 associated STREAM-head read queue.

```
17544 #include <stropts.h>
17545 ...

17546 int fd;
17547 char ctrlbuf[128];
17548 char databuf[512];
17549 struct strbuf ctrl;
17550 struct strbuf data;
17551 int band = 0;
17552 int flags = MSG_ANY;
17553 int ret;

17554 ctrl.buf = ctrlbuf;
17555 ctrl.maxlen = sizeof(ctrlbuf);

17556 data.buf = databuf;
17557 data.maxlen = sizeof(databuf);

17558 ret = getpmsg (fd, &ctrl, &data, &band, &flags);
```

17559 **APPLICATION USAGE**

17560 None.

17561 **RATIONALE**

17562 None.

17563 **FUTURE DIRECTIONS**

17564 None.

17565 **SEE ALSO**

17566 *poll*( ), *putmsg*( ), *read*( ), *write*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
17567 **<stropts.h>**, Section 2.6 (on page 539)

17568 **CHANGE HISTORY**

17569 First released in Issue 4, Version 2.

17570 **Issue 5**

17571 Moved from X/OPEN UNIX extension to BASE.

17572 A paragraph regarding ‘‘high-priority control parts of messages’’ is added to the RETURN
17573 VALUE section.

17574 **Issue 6**

17575 This function is marked as part of the XSI STREAMS Option Group. |

17576 The **restrict** keyword is added to the *getmsg*( ) and *getpmsg*( ) prototypes for alignment with the |
17577 ISO/IEC 9899:1999 standard. |

17578 **NAME**

17579 getnameinfo — get name information

17580 ***Notes to Reviewers***

17581 *This section with side shading will not appear in the final copy. - Ed.*

17582 The IPv6 developers believe that *getnameinfo*( ) should not truncate the result if the user buffer is
17583 too small, but return an error status.

17584 **SYNOPSIS**

```
17585       #include <sys/socket.h>
17586       #include <netdb.h>

17587       int getnameinfo(const struct sockaddr *restrict sa, socklen_t salen,
17588           char *restrict node, socklen_t nodelen, char *restrict service,
17589           socklen_t servicelen, unsigned flags);
```

17590 **DESCRIPTION**

17591 The *getnameinfo*( ) function translates a socket address to a node name and service location, all of
17592 which are defined as in *getaddrinfo*( ).

17593 The *sa* argument points to a socket address structure to be translated.

17594 If the *node* argument is non-NULL and the *nodelen* argument is non-zero, then the *node* argument
17595 points to a buffer able to contain up to *nodelen* characters that receives the node name as a null-
17596 terminated string. If the *node* argument is NULL or the *nodelen* argument is zero, the node name
17597 shall not be returned. If the node's name cannot be located, the numeric form of the node's
17598 address is returned instead of its name.

17599 If the *service* argument is non-NULL and the *servicelen* argument is non-zero, then the *service*
17600 argument points to a buffer able to contain up to *servicelen* characters that receives the service
17601 name as a null-terminated string. If the *service* argument is NULL or the *servicelen* argument is
17602 zero, the service name shall not be returned. If the service's name cannot be located, the numeric
17603 form of the service address (for example, its port number) is returned instead of its name.

17604 The *node* and *service* arguments cannot both be NULL.

17605 The *flags* argument is a flag that changes the default actions of the function. By default the fully-
17606 qualified domain name (FQDN) for the host is returned, but:

17607 • If the flag bit NI_NOFQDN is set, only the node name portion of the FQDN shall be returned
17608 for local hosts.

17609 • If the flag bit NI_NUMERICHOST is set, the numeric form of the host's address shall be
17610 returned instead of its name, under all circumstances.

17611 • If the flag bit NI_NAMEREQD is set, an error shall be returned if the host's name cannot be
17612 located.

17613 • If the flag bit NI_NUMERICSERV is set, the numeric form of the service address shall be
17614 returned (for example, its port number) instead of its name, under all circumstances.

17615 • If the flag bit NI_DGRAM is set, this indicates that the service is a datagram service
17616 (SOCK_DGRAM). The default behavior shall assume that the service is a stream service
17617 (SOCK_STREAM).

17618 **Notes:**

17619 1. The two NI_NUMERICxxx flags are required to support the **–n** flag that many
17620 commands provide.

2. The NI_DGRAM flag is required for the few AF_INET and AF_INET6 port numbers (for example, 512-514) that represent different services for UDP and TCP.

The *getnameinfo*( ) function shall be thread-safe.

RETURN VALUE

A zero return value for *getnameinfo*( ) indicates successful completion; a non-zero return value indicates failure. The possible values for the failures are listed in the ERRORS section.

Upon successful completion, *getnameinfo*( ) shall return the *node* and *service* names, if requested, in the buffers provided. The returned names are always null-terminated strings, and may be truncated if the actual values are longer than can be stored in the buffers provided.

ERRORS

The *getnameinfo*( ) function shall fail and return the corresponding value if:

[EAI_AGAIN] The name could not be resolved at this time. Future attempts may succeed.

[EAI_BADFLAGS] The *flags* had an invalid value.

[EAI_FAIL] A non-recoverable error occurred.

[EAI_FAMILY] The address family was not recognized or the address length was invalid for the specified family.

[EAI_MEMORY] There was a memory allocation failure.

[EAI_NONAME] The name does not resolve for the supplied parameters.

NI_NAMEREQD is set and the host's name cannot be located, or both *nodename* and *servname* were null.

[EAI_SYSTEM] A system error occurred. The error code can be found in *errno*.

EXAMPLES

None.

APPLICATION USAGE

If the returned values are to be used as part of any further name resolution (for example, passed to *getaddrinfo*( )), applications shall either provide buffers large enough to store any result possible on the system or shall check for truncation and handle that case appropriately.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getaddrinfo( ), *getservbyname*( ), *getservbyport*( ), *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.

17661 **NAME**

17662 getnetbyaddr — network database functions

17663 **SYNOPSIS**

17664 `#include <netdb.h>`

17665 `struct netent *getnetbyaddr(uint32_t net, int type);`

17666 **DESCRIPTION**

17667 Refer to *endnetent*( ).

17668 **NAME**

17669 getnetbyname — network database functions

17670 **SYNOPSIS**

17671 `#include <netdb.h>`

17672 `struct netent *getnetbyname(const char *`*`name`*`);`

17673 **DESCRIPTION**

17674 Refer to *endnetent*( ).

17675 **NAME**

17676 getnetent — network database functions

17677 **SYNOPSIS**

17678 `#include <netdb.h>`

17679 `struct netent *getnetent(void);`

17680 **DESCRIPTION**

17681 Refer to *endnetent*( ).

17682 **NAME**

17683 getopt, optarg, opterr, optind, optopt — command option parsing

17684 **SYNOPSIS**

17685 ```
#include <unistd.h>
```

17686 ```
int getopt(int argc, char * const argv[], const char *optstring);
```
17687 ```
extern char *optarg;
```
17688 ```
extern int optind, opterr, optopt;
```

17689 **DESCRIPTION**

17690 The *getopt*( ) function is a command-line parser that can be used by applications that follow
17691 Utility Syntax Guidelines 3, 4, 5, 6, 7, 9, and 10 in the Base Definitions volume of
17692 IEEE Std. 1003.1-200x, Section 12.2, Utility Syntax Guidelines. The remaining guidelines are not
17693 addressed by *getopt*( ) and are the responsibility of the application.

17694 The parameters *argc* and *argv* are the argument count and argument array as passed to *main*( )
17695 (see *exec*). The argument *optstring* is a string of recognized option characters; if a character is
17696 followed by a colon, the option takes an argument. All option characters allowed by Utility
17697 Syntax Guideline 3 are allowed in *optstring*. The implementation may accept other characters as
17698 an extension.

17699 The variable *optind* is the index of the next element of the *argv*[ ] vector to be processed. It is
17700 initialized to 1 by the system, and *getopt*( ) updates it when it finishes with each element of
17701 *argv*[ ]. When an element of *argv*[ ] contains multiple option characters, it is unspecified how
17702 *getopt*( ) determines which options have already been processed.

17703 The *getopt*( ) function shall return the next option character (if one is found) from *argv* that
17704 matches a character in *optstring*, if there is one that matches. If the option takes an argument,
17705 *getopt*( ) shall set the variable *optarg* to point to the option-argument as follows:

17706 1. If the option was the last character in the string pointed to by an element of *argv*, then
17707 *optarg* contains the next element of *argv*, and *optind* is incremented by 2. If the resulting
17708 value of *optind* is greater than *argc*, this indicates a missing option-argument, and *getopt*( )
17709 shall return an error indication.

17710 2. Otherwise, *optarg* points to the string following the option character in that element of
17711 *argv*, and *optind* is incremented by 1.

17712 If, when *getopt*( ) is called:

17713 ```
argv[optind]
```  is a null pointer
17714 ```
*argv[optind]
```  is not the character –
17715 ```
argv[optind]
```  points to the string "–"

17716 *getopt*( ) shall return –1 without changing *optind*. If:

17717 ```
argv[optind]
```  points to the string "––"

17718 *getopt*( ) shall return –1 after incrementing *optind*.

17719 If *getopt*( ) encounters an option character that is not contained in *optstring*, it shall return the
17720 question-mark (`'?'`) character. If it detects a missing option-argument, it shall return the colon
17721 character (`':'`) if the first character of *optstring* was a colon, or a question-mark character (`'?'`)
17722 otherwise. In either case, *getopt*( ) shall set the variable *optopt* to the option character that caused
17723 the error. If the application has not set the variable *opterr* to 0 and the first character of *optstring*
17724 is not a colon, *getopt*( ) shall also print a diagnostic message to *stderr* in the format specified for
17725 the *getopts* utility.

The *getopt*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

## RETURN VALUE

The *getopt*( ) function shall return the next option character specified on the command line.

A colon (':') shall be returned if *getopt*( ) detects a missing argument and the first character of *optstring* was a colon (':').

A question mark ('?') shall be returned if *getopt*( ) encounters an option character not in *optstring* or detects a missing argument and the first character of *optstring* was not a colon (':').

Otherwise, *getopt*( ) shall return –1 when all command line options are parsed.

## ERRORS

No errors are defined.

## EXAMPLES

### Parsing Command Line Options

The following code fragment shows how you might process the arguments for a utility that can take the mutually-exclusive options *a* and *b* and the options *f* and *o*, both of which require arguments:

```
#include <unistd.h>

int
main(int argc, char *argv[ ])
{
    int c;
    int bflg, aflg, errflg;
    char *ifile;
    char *ofile;
    extern char *optarg;
    extern int optind, optopt;
    . . .
    while ((c = getopt(argc, argv, ":abf:o:")) != -1) {
        switch(c) {
        case 'a':
            if (bflg)
                errflg++;
            else
                aflg++;
            break;
        case 'b':
            if (aflg)
                errflg++;
            else {
                bflg++;
                bproc();
            }
            break;
        case 'f':
            ifile = optarg;
            break;
        case 'o':
```

```
                ofile = optarg;
                break;
                case ':':       /* -f or -o without operand */
                        fprintf(stderr,
                                "Option -%c requires an operand\n", optopt);
                        errflg++;
                        break;
            case '?':
                        fprintf(stderr,
                                "Unrecognized option: -%c\n", optopt);
                errflg++;
            }
        }
        if (errflg) {
            fprintf(stderr, "usage: . . . ");
            exit(2);
        }
        for ( ; optind < argc; optind++) {
            if (access(argv[optind], R_OK)) {
        . . .
    }
```

This code accepts any of the following as equivalent:

```
cmd –ao arg path path
cmd –a –o arg path path
cmd –o arg –a path path
cmd –a –o arg –– path path
cmd –a –oarg path path
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.

```
#include <unistd.h>
#include <stdio.h>
...
int c;
char *filename;
extern char *optarg;
extern int optind, optopt, opterr;
...
while ((c = getopt(argc, argv, ":abf:")) != -1) {
    switch(c) {
    case 'a':
        printf("a is set\n");
        break;
    case 'b':
        printf("b is set\n");
        break;
    case 'f':
        filename = optarg;
```

```
            printf("filename is %s\n", filename);
            break;
        case ':':
            printf("-%c without filename\n", optopt);
            break;
        case '?':
            printf("unknown arg %c\n", optopt);
            break;
        }
    }
```

Selecting Options from the Command Line

The following example selects the type of database routines the user wants to use based on the *Options* argument.

```
#include <unistd.h>
#include <string.h>
...
char *Options = "hdbtl";
...
int dbtype, i;
char c;
char *st;
...
dbtype = 0;
while ((c = getopt(argc, argv, Options)) != -1) {
    if ((st = strchr(Options, c)) != NULL) {
        dbtype = st - Options;
        break;
    }
}
```

APPLICATION USAGE

The *getopt*( ) function is only required to support option characters included in Utility Syntax Guideline 3. Many historical implementations of *getopt*( ) support other characters as options. This is an allowed extension, but applications that use extensions are not maximally portable. Note that support for multi-byte option characters is only possible when such characters can be represented as type **int**.

RATIONALE

The *optopt* variable represents historical practice and allows the application to obtain the identity of the invalid option.

The description has been written to make it clear that *getopt*( ), like the *getopts* utility, deals with option-arguments whether separated from the option by <blank> characters or not. Note that the requirements on *getopt*( ) and *getopts* are more stringent than the Utility Syntax Guidelines.

The *getopt*( ) function shall return –1, rather than EOF, so that **<stdio.h>** is not required.

The special significance of a colon as the first character of *optstring* makes *getopt*( ) consistent with the *getopts* utility. It allows an application to make a distinction between a missing argument and an incorrect option letter without having to examine the option letter. It is true that a missing argument can only be detected in one case, but that is a case that has to be considered.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>**, the Shell and Utilities volume of IEEE Std. 1003.1-200x

CHANGE HISTORY

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

Issue 4

The following changes are incorporated for alignment with the ISO POSIX-2 standard:

- The **<unistd.h>** header is added to the SYNOPSIS section and **<stdio.h>** is deleted.
- The type of argument *argv* is changed from **char**** to **char*const**[ ].
- The integer *optopt* is added to the list of external data items.
- The DESCRIPTION is largely rewritten, without functional change, for alignment with the ISO POSIX-2 standard, although the following differences should be noted:
  - If the function detects a missing option-argument, it returns a colon (':') and sets *optopt* to the option character.
  - The termination conditions under which *getopt*( ) returns –1 are extended. Also note that the termination condition is explicitly –1, rather than the value of EOF.
- The EXAMPLES section is changed to illustrate the new functionality.

Issue 5

A note indicating that the *getopt*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

IEEE PASC Interpretation 1003.2 #150 is applied.

17892 **NAME**

17893 getpeername — get the name of the peer socket

17894 **SYNOPSIS**

17895 `#include <sys/socket.h>`

17896 `int getpeername(int socket, struct sockaddr *restrict address,`
17897 `socklen_t *address_len);`

17898 **DESCRIPTION**

17899 The *getpeername*( ) function shall retrieve the peer address of the specified socket, store this
17900 address in the **sockaddr** structure pointed to by the *address* argument, and store the length of this
17901 address in the object pointed to by the *address_len* argument.

17902 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
17903 the stored address shall be truncated.

17904 If the protocol permits connections by unbound clients, and the peer is not bound, then the value
17905 stored in the object pointed to by *address* is unspecified.

17906 **RETURN VALUE**

17907 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
17908 indicate the error.

17909 **ERRORS**

17910 The *getpeername*( ) function shall fail if:

17911 [EBADF] The *socket* argument is not a valid file descriptor.

17912 [EINVAL] The socket has been shut down.

17913 [ENOTCONN] The socket is not connected or otherwise has not had the peer prespecified.

17914 [ENOTSOCK] The *socket* argument does not refer to a socket.

17915 [EOPNOTSUPP] The operation is not supported for the socket protocol.

17916 The *getpeername*( ) function may fail if:

17917 [ENOBUFS] Insufficient resources were available in the system to complete the call.

17918 **EXAMPLES**

17919 None.

17920 **APPLICATION USAGE**

17921 None.

17922 **RATIONALE**

17923 None.

17924 **FUTURE DIRECTIONS**

17925 None.

17926 **SEE ALSO**

17927 *accept*( ), *bind*( ), *getsockname*( ), *socket*( ), the Base Definitions volume of IEEE Std. 1003.1-200x,
17928 **<sys/socket.h>**

17929 **CHANGE HISTORY**

17930 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

17931 The **restrict** keyword is added to the *getpeername*( ) prototype for alignment with the
17932 ISO / IEC 9899: 1999 standard.

NAME

getpgid — get the process group ID for a process

SYNOPSIS

XSI
```
#include <unistd.h>

pid_t getpgid(pid_t pid);
```

DESCRIPTION

The *getpgid*( ) function shall return the process group ID of the process whose process ID is equal to *pid*. If *pid* is equal to 0, *getpgid*( ) shall return the process group ID of the calling process.

RETURN VALUE

Upon successful completion, *getpgid*( ) shall return a process group ID. Otherwise, it shall return (**pid_t**)–1 and set *errno* to indicate the error.

ERRORS

The *getpgid*( ) function shall fail if:

[EPERM] The process whose process ID is equal to *pid* is not in the same session as the calling process, and the implementation does not allow access to the process group ID of that process from the calling process.

[ESRCH] There is no process with a process ID equal to *pid*.

The *getpgid*( ) function may fail if:

[EINVAL] The value of the *pid* argument is invalid.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fork*( ), *getpgrp*( ), *getpid*( ), *getsid*( ), *setpgid*( ), *setsid*( ), 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.

17968 **NAME**

17969 getpgrp — get the process group ID of the calling process

17970 **SYNOPSIS**

17971 `#include <unistd.h>`

17972 `pid_t getpgrp(void);`

17973 **DESCRIPTION**

17974 The *getpgrp*( ) function shall return the process group ID of the calling process.

17975 **RETURN VALUE**

17976 The *getpgrp*( ) function is always successful and no return value is reserved to indicate an error.

17977 **ERRORS**

17978 No errors are defined.

17979 **EXAMPLES**

17980 None.

17981 **APPLICATION USAGE**

17982 None.

17983 **RATIONALE**

17984 4.3 BSD provides a *getpgrp*( ) function that returns the process group ID for a specified process.
17985 Although this function is used to support job control, all known job control shells always specify
17986 the calling process with this function. Thus, the simpler System V *getpgrp*( ) suffices, and the
17987 added complexity of the 4.3 BSD *getpgrp*( ) has been omitted from this volume of
17988 IEEE Std. 1003.1-200x.

17989 **FUTURE DIRECTIONS**

17990 None.

17991 **SEE ALSO**

17992 *exec*, *fork*( ), *getpgid*( ), *getpid*( ), *getppid*( ), *kill*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of
17993 IEEE Std. 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

17994 **CHANGE HISTORY**

17995 First released in Issue 1. Derived from Issue 1 of the SVID.

17996 **Issue 4**

17997 The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on
17998 XSI-conformant systems.

17999 The **<unistd.h>** header is added to the SYNOPSIS section.

18000 The following change is incorporated for alignment with the ISO POSIX-1 standard:

18001 - The argument list is explicitly defined as **void**.

18002 **Issue 6**

18003 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

18004 The following new requirements on POSIX implementations derive from alignment with the
18005 Single UNIX Specification:

18006 - The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
18007 required for conforming implementations of previous POSIX specifications, it was not
18008 required for UNIX applications.

18009 **NAME**

18010 getpid — get the process ID

18011 **SYNOPSIS**

18012
```
#include <unistd.h>
```

18013
```
pid_t getpid(void);
```

18014 **DESCRIPTION**

18015 The *getpid*( ) function shall return the process ID of the calling process.

18016 **RETURN VALUE**

18017 The *getpid*( ) function is always successful and no return value is reserved to indicate an error.

18018 **ERRORS**

18019 No errors are defined.

18020 **EXAMPLES**

18021 None.

18022 **APPLICATION USAGE**

18023 None.

18024 **RATIONALE**

18025 None.

18026 **FUTURE DIRECTIONS**

18027 None.

18028 **SEE ALSO**

18029 *exec*, *fork*( ), *getpgrp*( ), *getppid*( ), *kill*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of |
18030 IEEE Std. 1003.1-200x, **<sys/types.h>**, **<unistd.h>** |

18031 **CHANGE HISTORY**

18032 First released in Issue 1. Derived from Issue 1 of the SVID. |

18033 **Issue 4**

18034 The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on
18035 XSI-conformant systems.

18036 The **<unistd.h>** header is added to the SYNOPSIS section.

18037 The following change is incorporated for alignment with the ISO POSIX-1 standard:

18038 • The argument list is explicitly defined as **void**.

18039 **Issue 6**

18040 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

18041 The following new requirements on POSIX implementations derive from alignment with the
18042 Single UNIX Specification:

18043 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
18044 required for conforming implementations of previous POSIX specifications, it was not
18045 required for UNIX applications.

18046 **NAME**

18047 getpmsg — receive next message from a STREAMS file

18048 **SYNOPSIS**

18049 XSI
```
#include <stropts.h>

int getpmsg(int fildes, struct strbuf *restrict ctlptr,
    struct strbuf *restrict dataptr, int *restrict bandp,
    int *restrict flagsp);
```

18053

18054 **DESCRIPTION**

18055 Refer to *getmsg*( ).

18056 **NAME**

18057 getppid — get the parent process ID

18058 **SYNOPSIS**

18059
```
#include <unistd.h>
```

18060
```
pid_t getppid(void);
```

18061 **DESCRIPTION**

18062 The *getppid*( ) function shall return the parent process ID of the calling process.

18063 **RETURN VALUE**

18064 The *getppid*( ) function is always successful and no return value is reserved to indicate an error.

18065 **ERRORS**

18066 No errors are defined.

18067 **EXAMPLES**

18068 None.

18069 **APPLICATION USAGE**

18070 None.

18071 **RATIONALE**

18072 None.

18073 **FUTURE DIRECTIONS**

18074 None.

18075 **SEE ALSO**

18076 *exec*, *fork*( ), *getpgid*( ), *getpgrp*( ), *getpid*( ), *kill*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of |
18077 IEEE Std. 1003.1-200x, **<sys/types.h>**, **<unistd.h>** |

18078 **CHANGE HISTORY**

18079 First released in Issue 1. Derived from Issue 1 of the SVID. |

18080 **Issue 4**

18081 The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on
18082 XSI-conformant systems.

18083 The **<unistd.h>** header is added to the SYNOPSIS section.

18084 The following change is incorporated for alignment with the ISO POSIX-1 standard:

18085 • The argument list is explicitly defined as **void**.

18086 **Issue 6**

18087 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

18088 The following new requirements on POSIX implementations derive from alignment with the
18089 Single UNIX Specification:

18090 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
18091 required for conforming implementations of previous POSIX specifications, it was not
18092 required for UNIX applications.

18093 **NAME**

18094 getpriority, setpriority — get or set the nice value

18095 **SYNOPSIS**

18096 XSI
```
#include <sys/resource.h>

int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int value);
```
18099

18100 **DESCRIPTION**

18101 The *getpriority*( ) function obtains the nice value of a process, process group, or user. The
18102 *setpriority*( ) function sets the nice value of a process, process group, or user to *value*+ {NZERO}.

18103 Target processes are specified by the values of the *which* and *who* arguments. The *which*
18104 argument may be one of the following values: PRIO_PROCESS, PRIO_PGRP, or PRIO_USER,
18105 indicating that the *who* argument is to be interpreted as a process ID, a process group ID, or an
18106 effective user ID, respectively. A 0 value for the *who* argument specifies the current process,
18107 process group, or user.

18108 The nice value set with *setpriority*( ) shall be applied to the process. If the process is multi-
18109 threaded, the nice value shall affect all system scope threads in the process.

18110 If more than one process is specified, *getpriority*( ) shall return value {NZERO} less than the
18111 lowest nice value pertaining to any of the specified processes, and *setpriority*( ) sets the nice
18112 values of all of the specified processes to *value*+ {NZERO}.

18113 The default nice value is {NZERO}; lower nice values cause more favorable scheduling. While
18114 the range of valid nice values is [0,{NZERO}*2 –1], implementations may enforce more
18115 restrictive limits. If *value*+ {NZERO} is less than the system's lowest supported nice value,
18116 *setpriority*( ) sets the nice value to the lowest supported value; if *value*+ {NZERO} is greater than
18117 the system's highest supported nice value, *setpriority*( ) sets the nice value to the highest
18118 supported value.

18119 Only a process with appropriate privileges can lower its nice value.

18120 PS | TPS Any processes or threads using SCHED_FIFO or SCHED_RR are unaffected by a call to
18121 *setpriority*( ). This is not considered an error.

18122 The effect of changing the nice value may vary depending on the process-scheduling algorithm
18123 in effect.

18124 Because *getpriority*( ) can return the value –1 on successful completion, it is necessary to set *errno*
18125 to 0 prior to a call to *getpriority*( ). If *getpriority*( ) returns the value –1, then *errno* can be checked
18126 to see if an error occurred or if the value is a legitimate nice value.

18127 **RETURN VALUE**

18128 Upon successful completion, *getpriority*( ) shall return an integer in the range from –{NZERO} to
18129 {NZERO}–1. Otherwise, –1 shall be returned and *errno* set to indicate the error.

18130 Upon successful completion, *setpriority*( ) shall return 0; otherwise, –1 shall be returned and *errno*
18131 set to indicate the error.

18132 **ERRORS**

18133 The *getpriority*( ) and *setpriority*( ) functions shall fail if:

18134 [ESRCH] No process could be located using the *which* and *who* argument values |
18135 specified.

[EINVAL] The value of the *which* argument was not recognized, or the value of the *who* argument is not a valid process ID, process group ID, or user ID.

In addition, *setpriority*( ) may fail if:

[EPERM] A process was located, but neither the real nor effective user ID of the executing process match the effective user ID of the process whose nice value is being changed.

[EACCES] A request was made to change the nice value to a lower numeric value and the current process does not have appropriate privileges.

EXAMPLES

Using getpriority( )

The following example returns the current scheduling priority for the process ID returned by the call to *getpid*( ).

```
#include <sys/resource.h>
...
int which = PRIO_PROCESS;
id_t pid;
int ret;

pid = getpid();
ret = getpriority(which, pid);
```

Using setpriority( )

The following example sets the priority for the current process ID to –20.

```
#include <sys/resource.h>
...
int which = PRIO_PROCESS;
id_t pid;
int priority = -20;
int ret;

pid = getpid();
ret = setpriority(which, pid, priority);
```

APPLICATION USAGE

The *getpriority*( ) and *setpriority*( ) functions work with an offset nice value (nice value –{NZERO}). The nice value is in the range [0,2*{NZERO} –1], while the return value for *getpriority*( ) and the third parameter for *setpriority*( ) are in the range [–{NZERO},{NZERO} –1].

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

nice( ), *sched_get_priority_max*( ), *sched_setscheduler*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/resource.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is reworded in terms of the nice value rather than *priority* to avoid confusion with functionality in the POSIX Realtime Extension.

18182 **NAME**

18183 getprotobyname — network protocol database functions

18184 **SYNOPSIS**

18185 `#include <netdb.h>`

18186 `struct protoent *getprotobyname(const char *`*`name`*`);`

18187 **DESCRIPTION**

18188 Refer to *endprotoent*( ).

18189 **NAME**

18190 getprotobynumber — network protocol database functions

18191 **SYNOPSIS**

18192 `#include <netdb.h>`

18193 `struct protoent *getprotobynumber(int proto);`

18194 **DESCRIPTION**

18195 Refer to *endprotoent*( ).

18196 **NAME**

18197 getprotoent — network protocol database functions

18198 **SYNOPSIS**

18199 `#include <netdb.h>`

18200 `struct protoent *getprotoent(void);`

18201 **DESCRIPTION**

18202 Refer to *endprotoent*( ).

18203 **NAME**

18204 getpwent — get user database entry

18205 **SYNOPSIS**

18206 XSI
```
#include <pwd.h>
```

18207
```
struct passwd *getpwent(void);
```

18208

18209 **DESCRIPTION**

18210 Refer to *endpwent*( ).

18211 **NAME**

18212 getpwnam, getpwnam_r — search user database for a name

18213 **SYNOPSIS**

18214 ```
#include <pwd.h>
```

18215 ```
struct passwd *getpwnam(const char *name);
```

18216 TSF ```
int getpwnam_r(const char *name, struct passwd *pwd, char *buffer,
```

18217 ```
    size_t bufsize, struct passwd **result);
```

18218

18219 **DESCRIPTION**

18220 The *getpwnam*( ) function shall search the user database for an entry with a matching *name*.

18221 The *getpwnam*( ) function need not be reentrant. A function that is not required to be reentrant is
18222 not required to be thread-safe.

18223 Applications wishing to check for error situations should set *errno* to 0 before calling
18224 *getpwnam*( ). If *getpwnam*( ) returns a null pointer and *errno* is non-zero, an error occurred.

18225 TSF The *getpwnam_r*( ) function updates the **passwd** structure pointed to by *pwd* and stores a pointer
18226 to that structure at the location pointed to by *result*. The structure shall contain an entry from
18227 the user database with a matching *name*. Storage referenced by the structure is allocated from
18228 the memory provided with the *buffer* parameter, which is *bufsize* characters in size.

18229 ***Notes to Reviewers***

18230 *This section with side shading will not appear in the final copy. - Ed.*

18231 D3, XSH, ERN 301 says that the size above is in bytes, not characters, and proposes changing
18232 "characters" to "bytes".

18233 The maximum size needed for this buffer can be determined with the
18234 {_SC_GETPW_R_SIZE_MAX} *sysconf*( ) parameter. A NULL pointer shall be returned at the
18235 location pointed to by *result* on error or if the requested entry is not found.

18236 **RETURN VALUE**

18237 The *getpwnam*( ) function shall return a pointer to a **struct passwd** with the structure as defined
18238 in **<pwd.h>** with a matching entry if found. A null pointer shall be returned if the requested
18239 entry is not found, or an error occurs. On error, *errno* shall be set to indicate the error.

18240 The return value may point to a static area which is overwritten by a subsequent call to
18241 *getpwent*( ), *getpwnam*( ), or *getpwuid*( ).

18242 TSF If successful, the *getpwnam_r*( ) function shall return zero; otherwise, an error number shall be
18243 returned to indicate the error.

18244 **ERRORS**

18245 The *getpwnam*( ) and *getpwnam_r*( ) functions may fail if:

18246 [EIO] An I/O error has occurred.

18247 [EINTR] A signal was caught during *getpwnam*( ).

18248 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

18249 [ENFILE] The maximum allowable number of files is currently open in the system.

18250 TSF The *getpwnam_r*( ) function may fail if:

18251 TSF [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to
18252 be 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.

```
#include <sys/types.h>
#include <pwd.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
...
char *lgn;
struct passwd *pw;
...
if ((lgn = getlogin()) == NULL || (pw = getpwnam(lgn)) == NULL) {
    fprintf(stderr, "Get of user information failed.\n"); exit(1);
}
...
```

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 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

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 4

The DESCRIPTION is clarified.

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

The last sentence in the RETURN VALUE section, indicating that *errno* is set on error, is marked as an extension.

The errors [EIO], [EINTR], [EMFILE], and [ENFILE] are marked as extensions.

The APPLICATION USAGE section is expanded to warn about possible reuses of the area used to pass the return value, and to indicate how applications should check for errors.

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument *name* is changed from **char*** to **const char***.

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 corrigenda item U028/3 has been applied correcting text in the DESCRIPTION describing matching the *name*.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

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 [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.

18318 **NAME**

18319 getpwuid, getpwuid_r — search user database for a user ID

18320 **SYNOPSIS**

18321 `#include <pwd.h>`

18322 `struct passwd *getpwuid(uid_t uid);`

18323 TSF `int getpwuid_r(uid_t uid, struct passwd *pwd, char *buffer,`

18324 `size_t bufsize, struct passwd **result);`

18325

18326 **DESCRIPTION**

18327 The *getpwuid*( ) function shall search the user database for an entry with a matching *uid*.

18328 The *getpwuid*( ) function need not be reentrant. A function that is not required to be reentrant is
18329 not required to be thread-safe.

18330 Applications wishing to check for error situations should set *errno* to 0 before calling *getpwuid*( ). |
18331 If *getpwuid*( ) returns a null pointer and *errno* is set to non-zero, an error occurred. |

18332 TSF The *getpwuid_r*( ) function updates the **passwd** structure pointed to by *pwd* and stores a pointer |
18333 to that structure at the location pointed to by *result*. The structure shall contain an entry from
18334 the user database with a matching *uid*. Storage referenced by the structure is allocated from the
18335 memory provided with the *buffer* parameter, which is *bufsize* characters in size. The maximum
18336 size needed for this buffer can be determined with the {_SC_GETPW_R_SIZE_MAX} *sysconf*( )
18337 parameter. A NULL pointer shall be returned at the location pointed to by *result* on error or if the
18338 requested entry is not found. |

18339 **RETURN VALUE**

18340 The *getpwuid*( ) function shall return a pointer to a **struct passwd** with the structure as defined in
18341 **<pwd.h>** with a matching entry if found. A null pointer shall be returned if the requested entry |
18342 is not found, or an error occurs. On error, *errno* shall be set to indicate the error. |

18343 The return value may point to a static area which is overwritten by a subsequent call to
18344 *getpwent*( ), *getpwnam*( ), or *getpwuid*( ).

18345 TSF If successful, the *getpwuid_r*( ) function shall return zero; otherwise, an error number shall be
18346 returned to indicate the error.

18347 **ERRORS**

18348 The *getpwuid*( ) and *getpwuid_r*( ) functions may fail if: |

18349 [EIO] An I/O error has occurred. |

18350 [EINTR] A signal was caught during *getpwuid*( ). |

18351 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process. |

18352 [ENFILE] The maximum allowable number of files is currently open in the system. |

18353 TSF The *getpwuid_r*( ) function may fail if:

18354 TSF [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to |
18355 be 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).

```
#include <sys/types.h>
#include <pwd.h>
...
uid_t id = 0;
struct passwd *pwd;

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.

```
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
...
struct passwd *pws;
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.

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
...
struct stat statbuf;
struct passwd *pwd;
...
if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
    printf(" %-8.8s", pwd->pw_name);
else
    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 shall return values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

18399 **RATIONALE**

18400 None.

18401 **FUTURE DIRECTIONS**

18402 None.

18403 **SEE ALSO**

18404 *getpwnam*( ), *geteuid*( ), *getuid*( ), *getlogin*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
18405 **<limits.h>**, **<pwd.h>**, **<sys/types.h>**

18406 **CHANGE HISTORY**

18407 First released in Issue 1. Derived from System V Release 2.0. |

18408 **Issue 4**

18409 The DESCRIPTION is clarified.

18410 The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on
18411 XSI-conformant systems.

18412 The last sentence in the RETURN VALUE section, indicating that *errno* is set on error, is marked
18413 as an extension.

18414 The errors [EIO], [EINTR], [EMFILE], and [ENFILE] are marked as extensions.

18415 A note is added to the APPLICATION USAGE section indicating how an application should
18416 check for errors.

18417 **Issue 5**

18418 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
18419 VALUE section.

18420 The *getpwuid_r*( ) function is included for alignment with the POSIX Threads Extension.

18421 A note indicating that the *getpwuid*( ) function need not be reentrant is added to the
18422 DESCRIPTION.

18423 **Issue 6**

18424 The *getpwuid_r*( ) function is marked as part of the Thread-Safe Functions option. |

18425 The Open Group corrigenda item U028/3 has been applied correcting text in the DESCRIPTION
18426 describing matching the *uid*.

18427 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

18428 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

18429 The following new requirements on POSIX implementations derive from alignment with the
18430 Single UNIX Specification:

18431 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
18432 required for conforming implementations of previous POSIX specifications, it was not
18433 required for UNIX applications.

18434 • In the RETURN VALUE section, the requirement to set *errno* on error is added.

18435 • The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.

18436 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
18437 its avoidance of possibly using a static data area.

18438 **NAME**

18439 getrlimit, setrlimit — control maximum resource consumption

18440 **SYNOPSIS**

18441 XSI
```
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlp);
int setrlimit(int resource, const struct rlimit *rlp);
```
18444

18445 **DESCRIPTION**

18446 Limits on the consumption of a variety of resources by the calling process may be obtained with
18447 *getrlimit*( ) and set with *setrlimit*( ).

18448 Each call to either *getrlimit*( ) or *setrlimit*( ) identifies a specific resource to be operated upon as
18449 well as a resource limit. A resource limit is represented by an **rlimit** structure. The *rlim_cur*
18450 member specifies the current or soft limit and the *rlim_max* member specifies the maximum or
18451 hard limit. Soft limits may be changed by a process to any value that is less than or equal to the
18452 hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or
18453 equal to the soft limit. Only a process with appropriate privileges can raise a hard limit. Both
18454 hard and soft limits can be changed in a single call to *setrlimit*( ) subject to the constraints
18455 described above.

18456 The value RLIM_INFINITY, defined in **<sys/resource.h>**, shall be considered to be larger than
18457 any other limit value. If a call to *getrlimit*( ) returns RLIM_INFINITY for a resource, it means the
18458 implementation shall not enforce limits on that resource. Specifying RLIM_INFINITY as any
18459 resource limit value on a successful call to *setrlimit*( ) inhibits enforcement of that resource limit.

18460 The following resources are defined:

18461 RLIMIT_CORE — This is the maximum size of a core file in bytes that may be created by a
18462 process. A limit of 0 shall prevent the creation of a core file. If this limit is
18463 exceeded, the writing of a core file shall terminate at this size.

18464 RLIMIT_CPU — This is the maximum amount of CPU time in seconds used by a process.
18465 If this limit is exceeded, SIGXCPU shall be generated for the process. If
18466 the process is catching or ignoring SIGXCPU, or all threads belonging to
18467 that process are blocking SIGXCPU, the behavior is unspecified.

18468 RLIMIT_DATA — This is the maximum size of a process' data segment in bytes. If this limit
18469 is exceeded, the *malloc*( ) function shall fail with *errno* set to [ENOMEM].

18470 RLIMIT_FSIZE — This is the maximum size of a file in bytes that may be created by a
18471 process. If a write or truncate operation would cause this limit to be
18472 exceeded, SIGXFSZ shall be generated for the thread. If the thread is
18473 blocking, or the process is catching or ignoring SIGXFSZ, continued
18474 attempts to increase the size of a file from end-of-file to beyond the limit
18475 shall fail with *errno* set to [EFBIG].

18476 RLIMIT_NOFILE — This is a number one greater than the maximum value that the system
18477 may assign to a newly-created descriptor. If this limit is exceeded,
18478 functions that allocate new file descriptors may fail with *errno* set to
18479 [EMFILE]. This limit constrains the number of file descriptors that a
18480 process may allocate.

18481 RLIMIT_STACK — This is the maximum size of a process' stack in bytes. The
18482 implementation does not automatically grow the stack beyond this limit.
18483 If this limit is exceeded, SIGSEGV shall be generated for the thread. If the

thread is blocking SIGSEGV, or the process is ignoring or catching SIGSEGV and has not made arrangements to use an alternate stack, the disposition of SIGSEGV shall be set to SIG_DFL before it is generated.

RLIMIT_AS — This is the maximum size of a process' total available memory, in bytes. If this limit is exceeded, the *malloc*( ) and *mmap*( ) functions shall fail with *errno* set to [ENOMEM]. In addition, the automatic stack growth fails with the effects outlined above.

When using the *getrlimit*( ) function, if a resource limit can be represented correctly in an object of type **rlim_t**, then its representation is returned; otherwise, if the value of the resource limit is equal to that of the corresponding saved hard limit, the value returned shall be RLIM_SAVED_MAX; otherwise, the value returned shall be RLIM_SAVED_CUR.

When using the *setrlimit*( ) function, if the requested new limit is RLIM_INFINITY, the new limit shall be ''no limit''; otherwise, if the requested new limit is RLIM_SAVED_MAX, the new limit shall be the corresponding saved hard limit; otherwise, if the requested new limit is RLIM_SAVED_CUR, the new limit shall be the corresponding saved soft limit; otherwise, the new limit shall be the requested value. In addition, if the corresponding saved limit can be represented correctly in an object of type **rlim_t** then it shall be overwritten with the new limit.

The result of setting a limit to RLIM_SAVED_MAX or RLIM_SAVED_CUR is unspecified unless a previous call to *getrlimit*( ) returned that value as the soft or hard limit for the corresponding resource limit.

The determination of whether a limit can be correctly represented in an object of type **rlim_t** is implementation-defined. For example, some implementations permit a limit whose value is greater than RLIM_INFINITY and others do not.

The *exec* family of functions also cause resource limits to be saved.

RETURN VALUE

Upon successful completion, *getrlimit*( ) and *setrlimit*( ) shall return 0. Otherwise, these functions shall return −1 and set *errno* to indicate the error.

ERRORS

The *getrlimit*( ) and *setrlimit*( ) functions shall fail if:

[EINVAL] — An invalid *resource* was specified; or in a *setrlimit*( ) call, the new *rlim_cur* exceeds the new *rlim_max*.

[EPERM] — The limit specified to *setrlimit*( ) would have raised the maximum limit value, and the calling process does not have appropriate privileges.

The *setrlimit*( ) function may fail if:

[EINVAL] — The limit specified cannot be lowered because current usage is already higher 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.

18525 **RATIONALE**

18526 None.

18527 **FUTURE DIRECTIONS**

18528 None.

18529 **SEE ALSO**

18530 *exec*, *fork*( ), *malloc*( ), *open*( ), *sigaltstack*( ), *sysconf*( ), *ulimit*( ), the Base Definitions volume of |
18531 IEEE Std. 1003.1-200x, **<stropts.h>**, **<sys/resource.h>** |

18532 **CHANGE HISTORY**

18533 First released in Issue 4, Version 2.

18534 **Issue 5**

18535 Moved from X/OPEN UNIX extension to BASE and an APPLICATION USAGE section is added.

18536 Large File Summit extensions are added.

18537 **NAME**

18538 getrusage — get information about resource utilization

18539 **SYNOPSIS**

18540 XSI
```
#include <sys/resource.h>

int getrusage(int who, struct rusage *r_usage);
```
18541

18542

18543 **DESCRIPTION**

18544 The *getrusage*( ) function provides measures of the resources used by the current process or its
18545 terminated and waited-for child processes. If the value of the *who* argument is RUSAGE_SELF,
18546 information shall be returned about resources used by the current process. If the value of the *who*
18547 argument is RUSAGE_CHILDREN, information shall be returned about resources used by the
18548 terminated and waited-for children of the current process. If the child is never waited for (for
18549 example, if the parent has SA_NOCLDWAIT set or sets SIGCHLD to SIG_IGN), the resource
18550 information for the child process is discarded and not included in the resource information
18551 provided by *getrusage*( ).

18552 The *r_usage* argument is a pointer to an object of type **struct rusage** in which the returned
18553 information is stored.

18554 **RETURN VALUE**

18555 Upon successful completion, *getrusage*( ) shall return 0; otherwise, –1 shall be returned and *errno*
18556 set to indicate the error.

18557 **ERRORS**

18558 The *getrusage*( ) function shall fail if:

18559 [EINVAL] The value of the *who* argument is not valid. |

18560 **EXAMPLES**

18561 **Using getrusage( )**

18562 The following example returns information about the resources used by the current process.

```
18563 #include <sys/resource.h>
18564 ...
18565 int who = RUSAGE_SELF;
18566 struct rusage usage;
18567 int ret;

18568 ret = getrusage(who, &usage);
```

18569 **APPLICATION USAGE**

18570 None.

18571 **RATIONALE**

18572 None.

18573 **FUTURE DIRECTIONS**

18574 None.

18575 **SEE ALSO**

18576 *exit*( ), *sigaction*( ), *time*( ), *times*( ), *wait*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
18577 **<sys/resource.h>**

18578 **CHANGE HISTORY**

18579 First released in Issue 4, Version 2.

18580 **Issue 5**

18581 Moved from X/OPEN UNIX extension to BASE.

18582 **NAME**

18583 gets — get a string from a stdin stream

18584 **SYNOPSIS**

18585 ```
#include <stdio.h>
```

18586 ```
char *gets(char *s);
```

18587 **DESCRIPTION**

18588 CX The functionality described on this reference page is aligned with the ISO C standard. Any
18589 conflict between the requirements described here and the ISO C standard is unintentional. This
18590 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

18591 The *gets*( ) function shall read bytes from the standard input stream, *stdin*, into the array pointed
18592 to by *s*, until a newline is read or an end-of-file condition is encountered. Any newline is
18593 discarded and a null byte is placed immediately after the last byte read into the array.

18594 CX The *gets*( ) function may mark the *st_atime* field of the file associated with *stream* for update. The
18595 *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ),
18596 *fread*( ), *getc*( ), *getchar*( ), *gets*( ), *fscanf*( ), or *scanf*( ) using *stream* that returns data not supplied by
18597 a prior call to *ungetc*( ).

18598 **RETURN VALUE**

18599 Upon successful completion, *gets*( ) shall return *s*. If the stream is at end-of-file, the end-of-file
18600 indicator for the stream shall be set and *gets*( ) shall return a null pointer. If a read error occurs,
18601 CX the error indicator for the stream shall be set, *gets*( ) shall return a null pointer and set *errno* to
18602 indicate the error.

18603 **ERRORS**

18604 Refer to *fgetc*( ).

18605 **EXAMPLES**

18606 None.

18607 **APPLICATION USAGE**

18608 Reading a line that overflows the array pointed to by *s* results in undefined behavior. The use of
18609 *fgets*( ) is recommended.

18610 Since the user cannot specify the length of the buffer passed to *gets*( ), use of this function is
18611 discouraged. The length of the string read is unlimited. It is possible to overflow this buffer in
18612 such a way as to cause applications to fail, or possible system security violations.

18613 It is recommended that the *fgets*( ) function should be used to read input lines.

18614 **RATIONALE**

18615 None.

18616 **FUTURE DIRECTIONS**

18617 None.

18618 **SEE ALSO**

18619 *feof*( ), *ferror*( ), *fgets*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**

18620 **CHANGE HISTORY**

18621 First released in Issue 1. Derived from Issue 1 of the SVID.

18622 **Issue 4**

18623 In the DESCRIPTION:

18624 • The text is changed to make it clear that the function reads bytes rather than (possibly multi-
18625 byte) characters.

- The list of functions that may cause the *st_atime* field to be updated is revised.

Issue 6

Extensions beyond the ISO C standard are now marked.

18629 **NAME**

18630 getservbyname — network services database functions

18631 **SYNOPSIS**

18632 `#include <netdb.h>`

18633 `struct servent *getservbyname(const char *`*`name`*`, const char *`*`proto`*`);`

18634 **DESCRIPTION**

18635 Refer to *endservent*( ).

18636 **NAME**

18637 getservbyport — network services database functions

18638 **SYNOPSIS**

18639 `#include <netdb.h>`

18640 `struct servent *getservbyport(int` *`port`*`, const char *`*`proto`*`);`

18641 **DESCRIPTION**

18642 Refer to *endservent*( ).

18643 **NAME**

18644 getservent — network services database functions

18645 **SYNOPSIS**

18646 `#include <netdb.h>`

18647 `struct servent *getservent(void);`

18648 **DESCRIPTION**

18649 Refer to *endservent*( ).

18650 **NAME**

18651 getsid — get the process group ID of a session leader

18652 **SYNOPSIS**

18653 XSI
```
#include <unistd.h>
```

18654
```
pid_t getsid(pid_t pid);
```

18655

18656 **DESCRIPTION**

18657 The *getsid*( ) function obtains the process group ID of the process that is the session leader of the
18658 process specified by *pid*. If *pid* is (**pid_t**)0, it specifies the calling process.

18659 **RETURN VALUE**

18660 Upon successful completion, *getsid*( ) shall return the process group ID of the session leader of
18661 the specified process. Otherwise, it shall return (**pid_t**)–1 and set *errno* to indicate the error.

18662 **ERRORS**

18663 The *getsid*( ) function shall fail if:

18664 [EPERM] The process specified by *pid* is not in the same session as the calling process, |
18665 and the implementation does not allow access to the process group ID of the
18666 session leader of that process from the calling process.

18667 [ESRCH] There is no process with a process ID equal to *pid*. |

18668 **EXAMPLES**

18669 None.

18670 **APPLICATION USAGE**

18671 None.

18672 **RATIONALE**

18673 None.

18674 **FUTURE DIRECTIONS**

18675 None.

18676 **SEE ALSO**

18677 *exec*, *fork*( ), *getpid*( ), *getpgid*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of |
18678 IEEE Std. 1003.1-200x, **<unistd.h>** |

18679 **CHANGE HISTORY**

18680 First released in Issue 4, Version 2.

18681 **Issue 5**

18682 Moved from X/OPEN UNIX extension to BASE.

18683 **NAME**

18684 getsockname — get the socket name

18685 **SYNOPSIS**

18686 `#include <sys/socket.h>`

18687 `int getsockname(int socket, struct sockaddr *restrict address,`
18688 `socklen_t *address_len);`

18689 **DESCRIPTION**

18690 The *getsockname*( ) function shall retrieve the locally-bound name of the specified socket, store
18691 this address in the **sockaddr** structure pointed to by the *address* argument, and store the length of
18692 this address in the object pointed to by the *address_len* argument.

18693 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
18694 the stored address shall be truncated.

18695 If the socket has not been bound to a local name, the value stored in the object pointed to by
18696 *address* is unspecified.

18697 **RETURN VALUE**

18698 Upon successful completion, 0 shall be returned, the *address* argument shall point to the address
18699 of the socket, and the *address_len* argument shall point to the length of the address. Otherwise, –1
18700 shall be returned and *errno* set to indicate the error.

18701 **ERRORS**

18702 The *getsockname*( ) function shall fail if:

18703 [EBADF] The *socket* argument is not a valid file descriptor.

18704 [ENOTSOCK] The *socket* argument does not refer to a socket.

18705 [EOPNOTSUPP] The operation is not supported for this socket's protocol.

18706 The *getsockname*( ) function may fail if:

18707 [EINVAL] The socket has been shut down.

18708 [ENOBUFS] Insufficient resources were available in the system to complete the function.

18709 **EXAMPLES**

18710 None.

18711 **APPLICATION USAGE**

18712 None.

18713 **RATIONALE**

18714 None.

18715 **FUTURE DIRECTIONS**

18716 None.

18717 **SEE ALSO**

18718 *accept*( ), *bind*( ), *getpeername*( ), *socket*( ), the Base Definitions volume of IEEE Std. 1003.1-200x,
18719 **<sys/socket.h>**

18720 **CHANGE HISTORY**

18721 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

18722 The **restrict** keyword is added to the *getsockname*( ) prototype for alignment with the
18723 ISO / IEC 9899: 1999 standard.

18724 **NAME**

18725 getsockopt — get the socket options

18726 **SYNOPSIS**

18727 `#include <sys/socket.h>`

18728 `int getsockopt(int socket, int level, int option_name,`
18729 `    void *restrict option_value, socklen_t *restrict option_len);`

18730 **DESCRIPTION**

18731 The *getsockopt*( ) function manipulates options associated with a socket.

18732 The *getsockopt*( ) function shall retrieve the value for the option specified by the *option_name*
18733 argument for the socket specified by the *socket* argument. If the size of the option value is greater
18734 than *option_len*, the value stored in the object pointed to by the *option_value* argument shall be
18735 silently truncated. Otherwise, the object pointed to by the *option_len* argument shall be modified
18736 to indicate the actual length of the value.

18737 The *level* argument specifies the protocol level at which the option resides. To retrieve options at
18738 the socket level, specify the *level* argument as SOL_SOCKET. To retrieve options at other levels,
18739 supply the appropriate level identifier for the protocol controlling the option. For example, to
18740 indicate that an option is interpreted by the TCP (Transmission Control Protocol), set *level* to
18741 IPPROTO_TCP as defined in the **<netinet/in.h>** header.

18742 The socket in use may require the process to have appropriate privileges to use the *getsockopt*( )
18743 function.

18744 The *option_name* argument specifies a single option to be retrieved. It can be one of the following
18745 values defined in **<sys/socket.h>**:

18746 SO_DEBUG — Reports whether debugging information is being recorded. This option
18747 stores an **int** value. This is a Boolean option.

18748 SO_ACCEPTCONN — Reports whether socket listening is enabled. This option stores an **int**
18749 value. This is a Boolean option.

18750 SO_BROADCAST — Reports whether transmission of broadcast messages is supported, if this
18751 is supported by the protocol. This option stores an **int** value. This is a
18752 Boolean option.

18753 SO_REUSEADDR — Reports whether the rules used in validating addresses supplied to *bind*( )
18754 should allow reuse of local addresses, if this is supported by the protocol.
18755 This option stores an **int** value. This is a Boolean option.

18756 SO_KEEPALIVE — Reports whether connections are kept active with periodic transmission
18757 of messages, if this is supported by the protocol.

18758 If the connected socket fails to respond to these messages, the connection
18759 is broken and threads writing to that socket are notified with a SIGPIPE
18760 signal. This option stores an **int** value.

18761 This is a Boolean option.

18762 SO_LINGER — Reports whether the socket lingers on *close*( ) if data is present. If
18763 SO_LINGER is set, the system blocks the process during *close*( ) until it
18764 can transmit the data or until the end of the interval indicated by the
18765 *l_linger* member, whichever comes first. If SO_LINGER is not specified,
18766 and *close*( ) is issued, the system handles the call in a way that allows the
18767 process to continue as quickly as possible. This option stores a **linger**
18768 structure.

SO_OOBINLINE Reports whether the socket leaves received out-of-band data (data marked urgent) inline. This option stores an **int** value. This is a Boolean option.

SO_SNDBUF Reports send buffer size information. This option stores an **int** value.

SO_RCVBUF Reports receive buffer size information. This option stores an **int** value.

SO_ERROR Reports information about error status and clears it. This option stores an **int** value.

SO_TYPE Reports the socket type. This option stores an **int** value.

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 stores an **int** value. This is a Boolean option.

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 than that returned; for example, out-of-band data.) This option stores an **int** value. Note that not all implementations allow this option to be retrieved.

SO_RCVTIMEO Reports the timeout value for input operations. This option stores 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 *errno* 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.

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 stores an **int** value. Note that not all implementations allow this option to be retrieved.

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 *errno* set to [EAGAIN] or [EWOULDBLOCK] if no data were sent. The default for this option is zero, which indicates that a send operation shall not time out. The option stores a **timeval** structure. Note that not all implementations allow this option to be retrieved.

For Boolean options, a zero value indicates that the option is disabled and a non-zero value indicates that the option is enabled.

Options at other protocol levels vary in format and name.

The socket in use may require the process to have appropriate privileges to use the *getsockopt*( ) function.

RETURN VALUE

Upon successful completion, *getsockopt*( ) shall return 0; otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *getsockopt*( ) function shall fail if:

[EBADF] The *socket* argument is not a valid file descriptor.

[EINVAL] The specified option is invalid at the specified socket level.

[ENOPROTOOPT]
The option is not supported by the protocol.

[ENOTSOCK] The *socket* argument does not refer to a socket.

The *getsockopt*( ) function may fail if:

[EACCES] The calling process does not have the appropriate privileges.

[EINVAL] The socket has been shut down.

[ENOBUFS] Insufficient resources are available in the system to complete the function.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

bind( ), *close*( ), *endprotoent*( ), *setsockopt*( ), *socket*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/socket.h>**, **<netinet/in.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The **restrict** keyword is added to the *getsockopt*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

18846 **NAME**

18847 getsubopt — parse suboption arguments from a string

18848 **SYNOPSIS**

18849 XSI
```
#include <stdlib.h>

int getsubopt(char **optionp, char * const *tokens, char **valuep);
```
18850

18851

18852 **DESCRIPTION**

18853 The *getsubopt*( ) function parses suboption arguments in a flag argument that was initially parsed
18854 by *getopt*( ). The application shall ensure that these suboption arguments are separated by
18855 commas and may consist of either a single token, or a token-value pair separated by an equal
18856 sign. Because commas delimit suboption arguments in the option string, they are not allowed to
18857 be part of the suboption arguments or the value of a suboption argument. Similarly, because the
18858 equal sign separates a token from its value, undefined behavior will result if the application
18859 passes a token that contains an equal sign.

18860 The *getsubopt*( ) function takes the address of a pointer to the option argument string, a vector of
18861 possible tokens, and the address of a value string pointer. If the option argument string at
18862 **optionp* contains only one suboption argument, *getsubopt*( ) updates **optionp* to point to the null
18863 at the end of the string. Otherwise, it isolates the suboption argument by replacing the comma
18864 separator with a null, and updates **optionp* to point to the start of the next suboption argument.
18865 If the suboption argument has an associated value, *getsubopt*( ) updates **valuep* to point to the
18866 value's first character. Otherwise, it sets **valuep* to a null pointer.

18867 The token vector is organized as a series of pointers to strings. The end of the token vector is
18868 identified by a null pointer.

18869 When *getsubopt*( ) returns, if **valuep* is not a null pointer, then the suboption argument processed
18870 included a value. The calling program may use this information to determine whether the
18871 presence or lack of a value for this suboption is an error.

18872 Additionally, when *getsubopt*( ) fails to match the suboption argument with the tokens in the
18873 *tokens* array, the calling program should decide if this is an error, or if the unrecognized option
18874 should be passed on to another program.

18875 **RETURN VALUE**

18876 The *getsubopt*( ) function shall return the index of the matched token string, or –1 if no token
18877 strings were matched.

18878 **ERRORS**

18879 No errors are defined.

18880 **EXAMPLES**

```
18881 #include <stdio.h>
18882 #include <stdlib.h>

18883 int do_all;
18884 const char *type;
18885 int read_size;
18886 int write_size;
18887 int read_only;

18888 enum
18889 {
18890     RO_OPTION = 0,
18891     RW_OPTION,
```

```
18892           READ_SIZE_OPTION,
18893           WRITE_SIZE_OPTION
18894       };
18895       const char *mount_opts[] =
18896       {
18897           [RO_OPTION] = "ro",
18898           [RW_OPTION] = "rw",
18899           [READ_SIZE_OPTION] = "rsize",
18900           [WRITE_SIZE_OPTION] = "wsize",
18901           NULL
18902       };
18903       int
18904       main(int argc, char *argv[])
18905       {
18906           char *subopts, *value;
18907           int opt;
18908           while ((opt = getopt(argc, argv, "at:o:")) != -1)
18909               switch(opt)
18910                   {
18911                   case 'a':
18912                       do_all = 1;
18913                       break;
18914                   case 't':
18915                       type = optarg;
18916                       break;
18917                   case 'o':
18918                       subopts = optarg;
18919                       while (*subopts != ' ')
18920                           switch(getsubopt(&subopts, mount_opts, &value))
18921                       {
18922                       case RO_OPTION:
18923                           read_only = 1;
18924                           break;
18925                       case RW_OPTION:
18926                           read_only = 0;
18927                           break;
18928                       case READ_SIZE_OPTION:
18929                           if (value == NULL)
18930                               abort();
18931                           read_size = atoi(value);
18932                           break;
18933                       case WRITE_SIZE_OPTION:
18934                           if (value == NULL)
18935                               abort();
18936                           write_size = atoi(value);
18937                           break;
18938                       default:
18939                           /* Unknown suboption. */
18940                           printf("Unknown suboption '%s'0, value);
18941                           break;
18942                       }
```

```
                break;
            default:
                abort();
            }
        /* Do the real work. */
        return 0;
    }
```

Parsing Suboptions

The following example uses the *getsubopt*( ) function to parse a value argument in the *optarg* external variable returned by a call to *getopt*( ).

```
    #include <stdlib.h>
    ...
    char *tokens[] = {"HOME", "PATH", "LOGNAME", (char *) NULL };
    char *value;
    int opt, index;

    while ((opt = getopt(argc, argv, "e:")) != -1) {
        switch(opt)  {
        case 'e' :
            while ((index = getsubopt(&optarg, tokens, &value)) != -1) {
                switch(index) {
    ...
            }
            break;
    ...
        }
    }
    ...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getopt( ), 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.

18982 **NAME**

18983 gettimeofday — get the date and time

18984 **SYNOPSIS**

18985 XSI
```
#include <sys/time.h>

int gettimeofday(struct timeval *restrict tp, void *tzp);
```
18986 |

18987 |

18988 **DESCRIPTION**

18989 The *gettimeofday*( ) function obtains the current time, expressed as seconds and microseconds
18990 since the Epoch, and stores it in the **timeval** structure pointed to by *tp*. The resolution of the
18991 system clock is unspecified.

18992 If *tzp* is not a null pointer, the behavior is unspecified.

18993 **RETURN VALUE**

18994 The *gettimeofday*( ) function shall return 0 and no value shall be reserved to indicate an error.

18995 **ERRORS**

18996 No errors are defined.

18997 **EXAMPLES**

18998 None.

18999 **APPLICATION USAGE**

19000 None.

19001 **RATIONALE**

19002 None.

19003 **FUTURE DIRECTIONS**

19004 None.

19005 **SEE ALSO**

19006 *ctime*( ), *ftime*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/time.h>** |

19007 **CHANGE HISTORY**

19008 First released in Issue 4, Version 2.

19009 **Issue 5**

19010 Moved from X/OPEN UNIX extension to BASE.

19011 **Issue 6**

19012 The DESCRIPTION is updated to refer to ‘‘seconds since the Epoch’’ rather than ‘‘seconds since
19013 00:00:00 UTC (Coordinated Universal Time), January 1 1970’’ for consistency with other *time* |
19014 functions.

19015 **NAME**

19016 getuid — get a real user ID

19017 **SYNOPSIS**

19018 `#include <unistd.h>`

19019 `uid_t getuid(void);`

19020 **DESCRIPTION**

19021 The *getuid*( ) function shall return the real user ID of the calling process.

19022 **RETURN VALUE**

19023 The *getuid*( ) function is always successful and no return value is reserved to indicate the error.

19024 **ERRORS**

19025 No errors are defined.

19026 **EXAMPLES**

19027 **Setting the Effective User ID to the Real User ID**

19028 The following example sets the effective user ID and the real user ID of the current process to the
19029 real user ID of the caller.

```
19030 #include <unistd.h>
19031 #include <sys/types.h>
19032 ...
19033 setreuid(getuid(), getuid());
19034 ...
```

19035 **APPLICATION USAGE**

19036 None.

19037 **RATIONALE**

19038 None.

19039 **FUTURE DIRECTIONS**

19040 None.

19041 **SEE ALSO**

19042 *getegid*( ), *geteuid*( ), *getgid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base
19043 Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

19044 **CHANGE HISTORY**

19045 First released in Issue 1. Derived from Issue 1 of the SVID.

19046 **Issue 4**

19047 The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on
19048 XSI-conformant systems.

19049 The **<unistd.h>** header is added to the SYNOPSIS section.

19050 The following change is incorporated for alignment with the ISO POSIX-1 standard:

19051 - The argument list is explicitly defined as **void**.

19052 **Issue 6**

19053 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

19054 The following new requirements on POSIX implementations derive from alignment with the
19055 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.

19059 **NAME**

19060 getutxent, getutxid, getutxline — get user accounting database entries

19061 **SYNOPSIS**

19062 XSI
```
#include <utmpx.h>

struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *id);
struct utmpx *getutxline(const struct utmpx *line);
```
19063

19064

19065

19066

19067 **DESCRIPTION**

19068 Refer to *endutxent*( ).

NAME

getwc — get a wide character from a stream

SYNOPSIS

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

wint_t getwc(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 *getwc*( ) function is equivalent to *fgetwc*( ), except that if it is implemented as a macro it may evaluate *stream* more than once, so the argument should never be an expression with side effects.

RETURN VALUE

Refer to *fgetwc*( ).

ERRORS

Refer to *fgetwc*( ).

EXAMPLES

None.

APPLICATION USAGE

Because it may be implemented as a macro, *getwc*( ) may treat incorrectly a *stream* argument with side effects. In particular, *getwc*(*f*++) does not necessarily work as expected. Therefore, use of this function is not recommended; *fgetwc*( ) should be used instead.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fgetwc( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**, **<wchar.h>** |

CHANGE HISTORY

First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working draft. |

Issue 5

The Optional Header (OH) marking is removed from **<stdio.h>**.

19102 **NAME**

19103 getwchar — get a wide character from a stdin stream

19104 **SYNOPSIS**

19105 `#include <wchar.h>`

19106 `wint_t getwchar(void);`

19107 **DESCRIPTION**

19108 CX The functionality described on this reference page is aligned with the ISO C standard. Any
19109 conflict between the requirements described here and the ISO C standard is unintentional. This
19110 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

19111 The *getwchar*( ) function is equivalent to *getwc*(*stdin*).

19112 **RETURN VALUE**

19113 Refer to *fgetwc*( ).

19114 **ERRORS**

19115 Refer to *fgetwc*( ).

19116 **EXAMPLES**

19117 None.

19118 **APPLICATION USAGE**

19119 If the **wint_t** value returned by *getwchar*( ) is stored into a variable of type **wchar_t** and then
19120 compared against the **wint_t** macro WEOF, the result may be incorrect. Only the **wint_t** type is
19121 guaranteed to be able to represent any wide character and WEOF.

19122 **RATIONALE**

19123 None.

19124 **FUTURE DIRECTIONS**

19125 None.

19126 **SEE ALSO**

19127 *fgetwc*( ), *getwc*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>**

19128 **CHANGE HISTORY**

19129 First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working
19130 draft.

19131 **NAME**

19132 getwd — get the current working directory path name (**LEGACY**)

19133 **SYNOPSIS**

19134 XSI
```
#include <unistd.h>
```

19135
```
char *getwd(char *path_name);
```

19136

19137 **DESCRIPTION**

19138 The *getwd*( ) function determines an absolute path name of the current working directory of the
19139 calling process, and copies that path name into the array pointed to by the *path_name* argument.

19140 If the length of the path name of the current working directory is greater than ({PATH_MAX}+1)
19141 including the null byte, *getwd*( ) shall fail and return a null pointer.

19142 **RETURN VALUE**

19143 Upon successful completion, a pointer to the string containing the absolute path name of the
19144 current working directory shall be returned. Otherwise, *getwd*( ) shall return a null pointer and
19145 the contents of the array pointed to by *path_name* are undefined.

19146 **ERRORS**

19147 No errors are defined.

19148 **EXAMPLES**

19149 None.

19150 **APPLICATION USAGE**

19151 For applications portability, the *getcwd*( ) function should be used to determine the current
19152 working directory instead of *getwd*( ).

19153 **RATIONALE**

19154 Since the user cannot specifyy the length of the buffer passed to *getwd*( ), use of this function is |
19155 discouraged. The length of a path name described in {PATH_MAX} is file system-dependent and |
19156 may vary from one mount point to another, or might even be unlimited. It is possible to |
19157 overflow this buffer in such a way as to cause applications to fail, or possible system security |
19158 violations. |

19159 It is recommended that the *getcwd*( ) function should be used to determine the current working |
19160 directory. |

19161 **FUTURE DIRECTIONS**

19162 This function may be withdrawn in a future version.

19163 **SEE ALSO**

19164 *getcwd*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>** |

19165 **CHANGE HISTORY**

19166 First released in Issue 4, Version 2.

19167 **Issue 5**

19168 Moved from X/OPEN UNIX extension to BASE.

19169 **Issue 6**

19170 This function is marked LEGACY.

19171 **NAME**

19172 glob, globfree — generate path names matching a pattern

19173 **SYNOPSIS**

19174 ```
#include <glob.h>
```

19175 ```
int glob(const char *restrict pattern, int flags,
    int(*retrict errfunc)(const char *retrict epath, int eerrno),
    glob_t *restrict pglob);
void globfree(glob_t *pglob);
```

19179 **DESCRIPTION**

19180 The *glob*( ) function is a path name generator that implements the rules defined in the Shell and
19181 Utilities volume of IEEE Std. 1003.1-200x, Section 2.14, Pattern Matching Notation, with optional
19182 support for rule 3 in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.14.3,
19183 Patterns Used for File Name Expansion.

19184 The structure type **glob_t** is defined in **<glob.h>** and includes at least the following members:

19185

19186

Member Type	Member Name	Description
size_t	*gl_pathc*	Count of paths matched by *pattern*.
char **	*gl_pathv*	Pointer to a list of matched path names.
size_t	*gl_offs*	Slots to reserve at the beginning of *gl_pathv*.

19190 The argument *pattern* is a pointer to a path name pattern to be expanded. The *glob*( ) function
19191 matches all accessible path names against this pattern and develops a list of all path names that
19192 match. In order to have access to a path name, *glob*( ) requires search permission on every
19193 component of a path except the last, and read permission on each directory of any file name
19194 component of *pattern* that contains any of the following special characters: `'*'`, `'?'`, and `'['`.

19195 The *glob*( ) function stores the number of matched path names into *pglob->gl_pathc* and a pointer
19196 to a list of pointers to path names into *pglob->gl_pathv*. The path names are in sort order as
19197 defined by the current setting of the *LC_COLLATE* category; see the Base Definitions volume of
19198 IEEE Std. 1003.1-200x, Section 7.3.2, LC_COLLATE . The first pointer after the last path name is a
19199 null pointer. If the pattern does not match any path names, the returned number of matched
19200 paths is set to 0, and the contents of *pglob->gl_pathv* are implementation-defined.

19201 It is the caller's responsibility to create the structure pointed to by *pglob*. The *glob*( ) function
19202 allocates other space as needed, including the memory pointed to by *gl_pathv*. The *globfree*( )
19203 function frees any space associated with *pglob* from a previous call to *glob*( ).

19204 The *flags* argument is used to control the behavior of *glob*( ). The value of *flags* is a bitwise-
19205 inclusive OR of zero or more of the following constants, which are defined in **<glob.h>**:

19206 GLOB_APPEND — Append path names generated to the ones from a previous call to *glob*( ).

19207 GLOB_DOOFFS — Make use of *pglob->gl_offs*. If this flag is set, *pglob->gl_offs* is used to
19208 specify how many null pointers to add to the beginning of *pglob->gl_pathv*. In other words, *pglob->gl_pathv* shall point to *pglob->gl_offs* null
19210 pointers, followed by *pglob->gl_pathc* path name pointers, followed by a
19211 null pointer.

19212 GLOB_ERR — Causes *glob*( ) to return when it encounters a directory that it cannot open
19213 or read. Ordinarily, *glob*( ) continues to find matches.

19214 GLOB_MARK — Each path name that is a directory that matches *pattern* has a slash
19215 appended.

GLOB_NOCHECK — Supports rule 3 in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.14.3, Patterns Used for File Name Expansion. If *pattern* does not match any path name, then *glob*( ) shall return a list consisting of only *pattern*, and the number of matched path names is 1.

GLOB_NOESCAPE — Disable backslash escaping.

GLOB_NOSORT — Ordinarily, *glob*( ) sorts the matching path names according to 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 . When this flag is used, the order of path names returned is unspecified.

The GLOB_APPEND flag can be used to append a new set of path names 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 path names that were in the *pglob->gl_pathv* list before the call, in the same order as before.
   c. Pointers to the new path names generated by the second call, in the specified order.
4. The count returned in *pglob->gl_pathc* shall be the total number of path names 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.)

The following constants are defined as error return values for *glob*( ):

GLOB_ABORTED — The scan was stopped because GLOB_ERR was set or (**errfunc*( )) returned non-zero.

GLOB_NOMATCH — The pattern does not match any existing path name, and GLOB_NOCHECK was not set in flags.

GLOB_NOSPACE — An attempt to allocate memory failed.

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.

RETURN VALUE

Upon successful completion, *glob*( ) shall return 0. The argument *pglob->gl_pathc* shall return the number of matched path names and the argument *pglob->gl_pathv* shall contain a pointer to a null-terminated list of matched and sorted path names. However, if *pglob->gl_pathc* is 0, the content of *pglob->gl_pathv* is undefined.

The *globfree*( ) function shall return no value.

If *glob*( ) terminates due to an error, it shall return one of the non-zero constants defined in **<glob.h>**. The arguments *pglob->gl_pathc* and *pglob->gl_pathv* are still set as defined above.

ERRORS

No errors are defined.

EXAMPLES

One use of the GLOB_DOOFFS flag is by applications that build an argument list for use with *execv*( ), *execve*( ), or *execvp*( ). Suppose, for example, that an application wants to do the equivalent of:

```
ls -l *.c
```

but for some reason:

```
system("ls -l *.c")
```

is not acceptable. The application could obtain approximately the same result using the sequence:

```
globbuf.gl_offs = 2;
glob("*.c", GLOB_DOOFFS, NULL, &globbuf);
globbuf.gl_pathv[0] = "ls";
globbuf.gl_pathv[1] = "-l";
execvp("ls", &globbuf.gl_pathv[0]);
```

Using the same example:

```
ls -l *.c *.h
```

could be approximately simulated using GLOB_APPEND as follows:

```
globbuf.gl_offs = 2;
glob("*.c", GLOB_DOOFFS, NULL, &globbuf);
glob("*.h", GLOB_DOOFFS|GLOB_APPEND, NULL, &globbuf);
...
```

APPLICATION USAGE

This function is not provided for the purpose of enabling utilities to perform path name expansion on their arguments, as this operation is performed by the shell, and utilities are explicitly not expected to redo this. Instead, it is provided for applications that need to do path name expansion on strings obtained from other sources, such as a pattern typed by a user or read from a file.

If a utility needs to see if a path name matches a given pattern, it can use *fnmatch*( ).

Note that *gl_pathc* and *gl_pathv* have meaning even if *glob*( ) fails. This allows *glob*( ) to report partial results in the event of an error. However, if *gl_pathc* is 0, *gl_pathv* is unspecified even if *glob*( ) did not return an error.

The GLOB_NOCHECK option could be used when an application wants to expand a path name if wildcards are specified, but wants to treat the pattern as just a string otherwise. The *sh* utility might use this for option-arguments, for example.

The new path names generated by a subsequent call with GLOB_APPEND are not sorted together with the previous path names. This mirrors the way that the shell handles path name 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 been obtained from *malloc*() in a large chunk and then carved up within *glob*(), or it might have been created using a different memory allocator. It is not the intent of the standard developers to specify or imply how the memory used by *glob*() is managed.

The GLOB_APPEND flag would be used when an application wants to expand several different patterns into a single list.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fnmatch*(), *opendir*(), *readdir*(), *stat*(), *wordexp*(), the Base Definitions volume of IEEE Std. 1003.1-200x, **<glob.h>**, the Shell and Utilities volume of IEEE Std. 1003.1-200x

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 5

Moved from POSIX2 C-language Binding to BASE.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *glob*() prototype for alignment with the ISO/IEC 9899: 1999 standard.

19334 **NAME**

19335 gmtime, gmtime_r — convert a time value to a broken-down UTC time

19336 **SYNOPSIS**

19337 `#include <time.h>`

19338 `struct tm *gmtime(const time_t *timer);`

19339 TSF `struct tm *gmtime_r(const time_t *restrict timer, struct tm *restrict result);`|

19340 |

19341 **DESCRIPTION**

19342 CX For *gmtime*( ): The functionality described on this reference page is aligned with the ISO C
19343 standard. Any conflict between the requirements described here and the ISO C standard is
19344 unintentional. This volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

19345 The *gmtime*( ) function shall convert the time in seconds since the Epoch pointed to by *timer* into
19346 a broken-down time, expressed as Coordinated Universal Time (UTC).

19347 CX The *gmtime*( ) funtion need not be reentrant. A function that is not required to be reentrant is not
19348 required to be thread-safe.

19349 TSF The *gmtime_r*( ) function shall convert the time in seconds since the Epoch pointed to by *timer* |
19350 into a broken-down time expressed as Coordinated Universal Time (UTC). The broken-down
19351 time is stored in the structure referred to by *result*. The *gmtime_r*( ) function shall also return the
19352 address of the same structure.

19353 **RETURN VALUE**

19354 The *gmtime*( ) function shall return a pointer to a **struct tm**.

19355 TSF Upon successful completion, *gmtime_r*( ) shall return the address of the structure pointed to by
19356 the argument *result*. If an error is detected, or UTC is not available, *gmtime_r*( ) shall return a null
19357 pointer.

19358 **ERRORS**

19359 No errors are defined.

19360 **EXAMPLES**

19361 None.

19362 **APPLICATION USAGE**

19363 The *asctime*( ), *ctime*( ), *gmtime*( ), and *localtime*( ) functions return values in one of two static
19364 objects: a broken-down time structure and an array of **char**. Execution of any of the functions
19365 may overwrite the information returned in either of these objects by any of the other functions.

19366 The *gmtime_r*( ) function is thread-safe and shall return values in a user-supplied buffer instead
19367 of possibly using a static data area that may be overwritten by each call.

19368 **RATIONALE**

19369 None.

19370 **FUTURE DIRECTIONS**

19371 None.

19372 **SEE ALSO**

19373 *asctime*( ), *clock*( ), *ctime*( ), *difftime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ), |
19374 the Base Definitions volume of IEEE Std. 1003.1-200x, **<time.h>** |

19375 **CHANGE HISTORY**

19376 First released in Issue 1. Derived from Issue 1 of the SVID.

19377 **Issue 4**

19378 In the APPLICATION USAGE section, the list of functions with which this function may interact
19379 is revised and the wording clarified.

19380 The following change is incorporated for alignment with the ISO C standard:

19381 • The type of argument *timer* is changed from **time_t*** to **const time_t***.

19382 **Issue 5**

19383 A note indicating that the *gmtime*() function need not be reentrant is added to the
19384 DESCRIPTION.

19385 The *gmtime_r*() function is included for alignment with the POSIX Threads Extension.

19386 **Issue 6**

19387 The *gmtime_r*() function is marked as part of the Thread-Safe Functions option.

19388 Extensions beyond the ISO C standard are now marked.

19389 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
19390 its avoidance of possibly using a static data area.

19391 The **restrict** keyword is added to the *gmtime_r*() prototype for alignment with the
19392 ISO/IEC 9899:1999 standard.

19393 **NAME**

19394 grantpt — grant access to the slave pseudo-terminal device

19395 **SYNOPSIS**

19396 XSI
```
#include <stdlib.h>

int grantpt(int fildes);
```
19397

19398

19399 **DESCRIPTION**

19400 The *grantpt*( ) function shall change the mode and ownership of the slave pseudo-terminal
19401 device associated with its master pseudo-terminal counterpart. The *fildes* argument is a file
19402 descriptor that refers to a master pseudo-terminal device. The user ID of the slave shall be set to
19403 the real UID of the calling process and the group ID shall be set to an unspecified group ID. The
19404 permission mode of the slave pseudo-terminal shall be set to readable and writable by the
19405 owner, and writable by the group.

19406 The behavior of the *grantpt*( ) function is unspecified if the application has installed a signal
19407 handler to catch SIGCHLD signals.

19408 **RETURN VALUE**

19409 Upon successful completion, *grantpt*( ) shall return 0; otherwise, it shall return –1 and set *errno* to
19410 indicate the error.

19411 **ERRORS**

19412 The *grantpt*( ) function may fail if:

19413 [EBADF] The *fildes* argument is not a valid open file descriptor. |

19414 [EINVAL] The *fildes* argument is not associated with a master pseudo-terminal device. |

19415 [EACCES] The corresponding slave pseudo-terminal device could not be accessed. |

19416 **EXAMPLES**

19417 None.

19418 **APPLICATION USAGE**

19419 None.

19420 **RATIONALE**

19421 None.

19422 **FUTURE DIRECTIONS**

19423 None.

19424 **SEE ALSO**

19425 *open*( ), *ptsname*( ), *unlockpt*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.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 The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section in
19431 previous issues.

19432 **NAME**

19433 h_errno — error return value for network database operations (**LEGACY**)

19434 **SYNOPSIS**

19435 `#include <netdb.h>`

19436 **DESCRIPTION**

19437 Note that this method of returning errors is used only in connection with legacy functions.

19438 The **<netdb.h>** header provides a declaration of *h_errno* as a modifiable *l*-value of type **int**.

19439 It is unspecified whether *h_errno* is a macro or an identifier declared with external linkage. If a
19440 macro definition is suppressed in order to access an actual object, or a program defines an
19441 identifier with the name *h_errno*, the behavior is undefined.

19442 **RETURN VALUE**

19443 None.

19444 **ERRORS**

19445 No errors are defined.

19446 **EXAMPLES**

19447 None.

19448 **APPLICATION USAGE**

19449 Applications should obtain the definition of *h_errno* by the inclusion of the **<netdb.h>** header.
19450 The practice of defining *h_errno* in a program as an **extern int** *h_errno* is obsolescent.

19451 **RATIONALE**

19452 None.

19453 **FUTURE DIRECTIONS**

19454 *h_errno* may be withdrawn in a future version.

19455 **SEE ALSO**

19456 *endhostent*( ), *errno*, the Base Definitions volume of IEEE Std. 1003.1-200x, **<netdb.h>**

19457 **CHANGE HISTORY**

19458 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19459 **NAME**

19460 hcreate, hdestroy, hsearch — manage hash search table

19461 **SYNOPSIS**

19462 XSI
```
#include <search.h>

int hcreate(size_t nel);
void hdestroy(void);
ENTRY *hsearch(ENTRY item, ACTION action);
```

19463
19464
19465
19466

19467 **DESCRIPTION**

19468 The *hcreate*( ), *hdestroy*( ), and *hsearch*( ) functions manage hash search tables.

19469 The *hcreate*( ) function allocates sufficient space for the table, and the application shall ensure it is
19470 called before *hsearch*( ) is used. The *nel* argument is an estimate of the maximum number of
19471 entries that the table shall contain. This number may be adjusted upward by the algorithm in
19472 order to obtain certain mathematically favorable circumstances.

19473 The *hdestroy*( ) function disposes of the search table, and may be followed by another call to
19474 *hcreate*( ). After the call to *hdestroy*( ), the data can no longer be considered accessible.

19475 The *hsearch*( ) function is a hash-table search routine. It shall return a pointer into a hash table
19476 indicating the location at which an entry can be found. The *item* argument is a structure of type
19477 **ENTRY** (defined in the **<search.h>** header) containing two pointers: *item.key* points to the
19478 comparison key (a **char***), and *item.data* (a **void***) points to any other data to be associated with
19479 that key. The comparison function used by *hsearch*( ) is *strcmp*( ). The *action* argument is a
19480 member of an enumeration type **ACTION** indicating the disposition of the entry if it cannot be
19481 found in the table. ENTER indicates that the item should be inserted in the table at an
19482 appropriate point. FIND indicates that no entry should be made. Unsuccessful resolution is
19483 indicated by the return of a null pointer.

19484 These functions need not be reentrant. A function that is not required to be reentrant is not
19485 required to be thread-safe.

19486 **RETURN VALUE**

19487 The *hcreate*( ) function shall return 0 if it cannot allocate sufficient space for the table; otherwise,
19488 it shall return non-zero.

19489 The *hdestroy*( ) function shall return no value.

19490 The *hsearch*( ) function shall return a null pointer if either the action is FIND and the item could
19491 not be found or the action is ENTER and the table is full.

19492 **ERRORS**

19493 The *hcreate*( ) and *hsearch*( ) functions may fail if:

19494 [ENOMEM] Insufficient storage space is available.

19495 **EXAMPLES**

19496 The following example reads in strings followed by two numbers and stores them in a hash
19497 table, discarding duplicates. It then reads in strings and finds the matching entry in the hash
19498 table and prints it out.

```
19499 #include <stdio.h>
19500 #include <search.h>
19501 #include <string.h>

19502 struct info {        /* This is the info stored in the table */
19503     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;
}
```

APPLICATION USAGE

The *hcreate*( ) and *hsearch*( ) functions may use *malloc*( ) to allocate space.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

bsearch( ), *lsearch*( ), *malloc*( ), *strcmp*( ), *tsearch*( ), 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 4

In the SYNOPSIS section, the type of argument *nel* in the declaration of *hcreate*( ) is changed from **unsigned** to **size_t**, and the argument list is explicitly defined as **void** in the declaration of *hdestroy*( ).

In the DESCRIPTION, the type of the comparison key is explicitly defined as **char***, the type of *item.data* is explicitly defined as **void***, and a statement is added indicating that *hsearch*( ) uses *strcmp*( ) as the comparison function.

In the EXAMPLES section, the sample code is updated to use ISO C standard syntax.

An ERRORS section is added and [ENOMEM] is defined as an error that may be returned by *hsearch*( ) and *hcreate*( ).

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

htonl, htons, ntohl, ntohs — convert values between host and network byte order

SYNOPSIS

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

DESCRIPTION

These functions shall convert 16-bit and 32-bit quantities between network byte order and host byte order.

On some implementations, these functions are defined as macros.

The **uint32_t** and **uint16_t** types shall be defined as described in **<inttypes.h>**.

RETURN VALUE

The *htonl*( ) and *htons*( ) functions shall return the argument value converted from host to network byte order.

The *ntohl*( ) and *ntohs*( ) functions shall return the argument value converted from network to host byte order.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

These functions are most often used in conjunction with IPv4 addresses and ports as returned by *gethostent*( ) and *getservent*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endhostent( ), *endservent*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<inttypes.h>**, **<arpa/inet.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19600 **NAME**

19601 htons — convert values between host and network byte order

19602 **SYNOPSIS**

19603 `#include <arpa/inet.h>`

19604 `uint16_t htons(uint16_t` *`hostshort`*`);`

19605 **DESCRIPTION**

19606 Refer to *htonl*( ).

19607 **NAME**

19608 hypot, hypotf, hypotl — Euclidean distance function

19609 **SYNOPSIS**

19610 XSI
```
#include <math.h>

double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
```

19614

19615 **DESCRIPTION**

19616 These functions shall compute the value of the square root of x^2+y^2.

19617 An application wishing to check for error situations should set *errno* to 0 before calling *hypot*().
19618 If *errno* is non-zero on return, or the return value is HUGE_VAL or NaN, an error has occurred.

19619 **RETURN VALUE**

19620 Upon successful completion, these functions shall return the length of the hypotenuse of a
19621 right-angled triangle with sides of length *x* and *y*.

19622 If the result would cause overflow, HUGE_VAL shall be returned and *errno* may be set to
19623 [ERANGE].

19624 If *x* or *y* is NaN, NaN shall be returned. and *errno* may be set to [EDOM].

19625 If the correct result would cause underflow, 0 shall be returned and *errno* may be set to
19626 [ERANGE].

19627 **ERRORS**

19628 These functions may fail if:

19629 [EDOM] The value of *x* or *y* is NaN.

19630 [ERANGE] The result overflows or underflows.

19631 No other errors shall occur.

19632 **EXAMPLES**

19633 None.

19634 **APPLICATION USAGE**

19635 The *hypot*() function takes precautions against overflow during intermediate steps of the
19636 computation. If the calculated result would still overflow a double, then *hypot*() shall return
19637 HUGE_VAL.

19638 **RATIONALE**

19639 None.

19640 **FUTURE DIRECTIONS**

19641 None.

19642 **SEE ALSO**

19643 *isnan*(), *sqrt*(), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

19644 **CHANGE HISTORY**

19645 First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 4

References to *matherr*( ) are removed.

The RETURN VALUE and ERRORS sections are substantially rewritten to rationalize error handling in the mathematics functions.

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 *hypotf*( ) and *hypotl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

19656 **NAME**

19657 iconv — codeset conversion function

19658 **SYNOPSIS**

19659 XSI
```
#include <iconv.h>

size_t iconv(iconv_t cd, char **restrict inbuf,
    size_t *restrict inbytesleft, char **restrict outbuf,
    size_t *restrict outbytesleft);
```

19663

19664 **DESCRIPTION**

19665 The *iconv*( ) function shall convert the sequence of characters from one codeset, in the array
19666 specified by *inbuf*, into a sequence of corresponding characters in another codeset, in the array
19667 specified by *outbuf*. The codesets are those specified in the *iconv_open*( ) call that returned the
19668 conversion descriptor, *cd*. The *inbuf* argument points to a variable that points to the first
19669 character in the input buffer and *inbytesleft* indicates the number of bytes to the end of the buffer
19670 to be converted. The *outbuf* argument points to a variable that points to the first available byte in
19671 the output buffer and *outbytesleft* indicates the number of the available bytes to the end of the
19672 buffer.

19673 For state-dependent encodings, the conversion descriptor *cd* is placed into its initial shift state by
19674 a call for which *inbuf* is a null pointer, or for which *inbuf* points to a null pointer. When *iconv*( ) is
19675 called in this way, and if *outbuf* is not a null pointer or a pointer to a null pointer, and *outbytesleft*
19676 points to a positive value, *iconv*( ) shall place, into the output buffer, the byte sequence to change
19677 the output buffer to its initial shift state. If the output buffer is not large enough to hold the
19678 entire reset sequence, *iconv*( ) shall fail and set *errno* to [E2BIG]. Subsequent calls with *inbuf* as
19679 other than a null pointer or a pointer to a null pointer cause the conversion to take place from
19680 the current state of the conversion descriptor.

19681 If a sequence of input bytes does not form a valid character in the specified codeset, conversion
19682 stops after the previous successfully converted character. If the input buffer ends with an
19683 incomplete character or shift sequence, conversion stops after the previous successfully
19684 converted bytes. If the output buffer is not large enough to hold the entire converted input,
19685 conversion stops just prior to the input bytes that would cause the output buffer to overflow.
19686 The variable pointed to by *inbuf* is updated to point to the byte following the last byte
19687 successfully used in the conversion. The value pointed to by *inbytesleft* is decremented to reflect
19688 the number of bytes still not converted in the input buffer. The variable pointed to by *outbuf* is
19689 updated to point to the byte following the last byte of converted output data. The value pointed
19690 to by *outbytesleft* is decremented to reflect the number of bytes still available in the output buffer.
19691 For state-dependent encodings, the conversion descriptor is updated to reflect the shift state in
19692 effect at the end of the last successfully converted byte sequence.

19693 If *iconv*( ) encounters a character in the input buffer that is valid, but for which an identical
19694 character does not exist in the target codeset, *iconv*( ) performs an implementation-defined
19695 conversion on this character.

19696 **RETURN VALUE**

19697 The *iconv*( ) function shall update the variables pointed to by the arguments to reflect the extent
19698 of the conversion and return the number of non-identical conversions performed. If the entire
19699 string in the input buffer is converted, the value pointed to by *inbytesleft* shall be 0. If the input
19700 conversion is stopped due to any conditions mentioned above, the value pointed to by *inbytesleft*
19701 shall be non-zero and *errno* shall be set to indicate the condition. If an error occurs *iconv*( ) shall
19702 return (**size_t**)–1 and set *errno* to indicate the error.

19703 **ERRORS**

19704 The *iconv*( ) function shall fail if:

19705 [EILSEQ] Input conversion stopped due to an input byte that does not belong to the input codeset.

19707 [E2BIG] Input conversion stopped due to lack of space in the output buffer.

19708 [EINVAL] Input conversion stopped due to an incomplete character or shift sequence at the end of the input buffer.

19710 The *iconv*( ) function may fail if:

19711 [EBADF] The *cd* argument is not a valid open conversion descriptor.

19712 **EXAMPLES**

19713 None.

19714 **APPLICATION USAGE**

19715 The *inbuf* argument indirectly points to the memory area which contains the conversion input data. The *outbuf* argument indirectly points to the memory area which is to contain the result of the conversion. The objects indirectly pointed to by *inbuf* and *outbuf* are not restricted to containing data that is directly representable in the ISO C standard language **char** data type. The type of *inbuf* and *outbuf*, **char****, does not imply that the objects pointed to are interpreted as null-terminated C strings or arrays of characters. Any interpretation of a byte sequence that represents a character in a given character set encoding scheme is done internally within the codeset converters. For example, the area pointed to indirectly by *inbuf* and/or *outbuf* can contain all zero octets that are not interpreted as string terminators but as coded character data according to the respective codeset encoding scheme. The type of the data (**char**, **short**, **long**, and so on) read or stored in the objects is not specified, but may be inferred for both the input and output data by the converters determined by the *fromcode* and *tocode* arguments of *iconv_open*( ).

19727 Regardless of the data type inferred by the converter, the size of the remaining space in both input and output objects (the *intbytesleft* and *outbytesleft* arguments) is always measured in bytes.

19729 For implementations that support the conversion of state-dependent encodings, the conversion descriptor must be able to accurately reflect the shift-state in effect at the end of the last successful conversion. It is not required that the conversion descriptor itself be updated, which would require it to be a pointer type. Thus, implementations are free to implement the descriptor as a handle (other than a pointer type) by which the conversion information can be accessed and updated.

19735 **RATIONALE**

19736 None.

19737 **FUTURE DIRECTIONS**

19738 None.

19739 **SEE ALSO**

19740 *iconv_open*( ), *iconv_close*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<iconv.h>**

19741 **CHANGE HISTORY**

19742 First released in Issue 4. Derived from the HP-UX Manual.

19743 **Issue 6**

19744 The SYNOPSIS has been corrected to align with the **<iconv.h>** reference page.

19745 The **restrict** keyword is added to the *iconv*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

19747 **NAME**

19748 iconv_close — codeset conversion deallocation function

19749 **SYNOPSIS**

19750 XSI
```
#include <iconv.h>
```

19751
```
int iconv_close(iconv_t cd);
```

19752

19753 **DESCRIPTION**

19754 The *iconv_close*( ) function deallocates the conversion descriptor *cd* and all other associated
19755 resources allocated by *iconv_open*( ).

19756 If a file descriptor is used to implement the type **iconv_t**, that file descriptor is closed.

19757 **RETURN VALUE**

19758 Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to
19759 indicate the error.

19760 **ERRORS**

19761 The *iconv_close*( ) function may fail if:

19762 [EBADF] The conversion descriptor is invalid. |

19763 **EXAMPLES**

19764 None.

19765 **APPLICATION USAGE**

19766 None.

19767 **RATIONALE**

19768 None.

19769 **FUTURE DIRECTIONS**

19770 None.

19771 **SEE ALSO**

19772 *iconv*( ), *iconv_open*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<iconv.h>** |

19773 **CHANGE HISTORY**

19774 First released in Issue 4. Derived from the HP-UX Manual. |

19775 **NAME**

19776 iconv_open — codeset conversion allocation function

19777 **SYNOPSIS**

19778 XSI
```
#include <iconv.h>

iconv_t iconv_open(const char *tocode, const char *fromcode);
```
19780

19781 **DESCRIPTION**

19782 The *iconv_open*( ) function shall return a conversion descriptor that describes a conversion from
19783 the codeset specified by the string pointed to by the *fromcode* argument to the codeset specified
19784 by the string pointed to by the *tocode* argument. For state-dependent encodings, the conversion
19785 descriptor is in a codeset-dependent initial shift state, ready for immediate use with *iconv*( ).

19786 Settings of *fromcode* and *tocode* and their permitted combinations are implementation-defined.

19787 A conversion descriptor remains valid until it is closed by *iconv_close*( ) or an implicit close.

19788 If a file descriptor is used to implement conversion descriptors, the FD_CLOEXEC flag shall be
19789 set; see **<fcntl.h>**.

19790 **RETURN VALUE**

19791 Upon successful completion, *iconv_open*( ) shall return a conversion descriptor for use on
19792 subsequent calls to *iconv*( ). Otherwise, *iconv_open*( ) shall return (**iconv_t**)–1 and set *errno* to
19793 indicate the error.

19794 **ERRORS**

19795 The *iconv_open*( ) function may fail if:

19796 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

19797 [ENFILE] Too many files are currently open in the system.

19798 [ENOMEM] Insufficient storage space is available.

19799 [EINVAL] The conversion specified by *fromcode* and *tocode* is not supported by the
19800 implementation.

19801 **EXAMPLES**

19802 None.

19803 **APPLICATION USAGE**

19804 Some implementations of *iconv_open*( ) use *malloc*( ) to allocate space for internal buffer areas.
19805 The *iconv_open*( ) function may fail if there is insufficient storage space to accommodate these
19806 buffers.

19807 Portable applications must assume that conversion descriptors are not valid after a call to one of
19808 the *exec* functions.

19809 **RATIONALE**

19810 None.

19811 **FUTURE DIRECTIONS**

19812 None.

19813 **SEE ALSO**

19814 *iconv*( ), *iconv_close*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<fcntl.h>**, **<iconv.h>**

19815 **CHANGE HISTORY**

19816 First released in Issue 4. Derived from the HP-UX Manual.

19817 **NAME**

19818 if_freenameindex — free memory allocated by *ifnameindex*( )

19819 **SYNOPSIS**

19820 `#include <net/if.h>`

19821 `void if_freenameindex(struct if_nameindex *ptr);`

19822 **DESCRIPTION**

19823 The *if_freenameindex*( ) function shall free the memory allocated by *if_nameindex*. The *ptr* |
19824 argument shall be a pointer that was returned by *if_nameindex*. After *if_freenameindex*( ) has been |
19825 called, the application should not use the array of which *ptr* is the address.

19826 **RETURN VALUE**

19827 None.

19828 **ERRORS**

19829 No errors are defined. |

19830 **EXAMPLES**

19831 None.

19832 **APPLICATION USAGE**

19833 None.

19834 **RATIONALE**

19835 None.

19836 **FUTURE DIRECTIONS**

19837 None.

19838 **SEE ALSO**

19839 *getsockopt*( ), *if_indextoname*( ), *if_nameindex*( ), *if_nametoindex*( ), *setsockopt*( ), the Base Definitions |
19840 volume of IEEE Std. 1003.1-200x, **<net/if.h>** |

19841 **CHANGE HISTORY**

19842 First released in Issue 6. Derived from the XNS, Issue 5.2 specification. |

19843 **NAME**

19844 if_indextoname — map a network interface index to its corresponding name

19845 **SYNOPSIS**

19846 `#include <net/if.h>`

19847 `char *if_indextoname(unsigned ifindex, char *ifname);`

19848 **DESCRIPTION**

19849 The *if_indextoname*( ) function shall map an interface index to its corresponding name.

19850 When this function is called, *ifname* shall point to a buffer of at least {IFNAMSIZ} bytes. The
19851 function shall place in this buffer the name of the interface with index *ifindex*.

19852 **RETURN VALUE**

19853 If *ifindex* is an interface index, then the function shall return the value supplied in *ifname*, which
19854 points to a buffer now containing the interface name. Otherwise, the function shall return a
19855 NULL pointer and set *errno* to indicate the error.

19856 **ERRORS**

19857 The *if_indextoname*( ) function shall fail if:

19858 [ENXIO] The interface does not exist.

19859 **EXAMPLES**

19860 None.

19861 **APPLICATION USAGE**

19862 None.

19863 **RATIONALE**

19864 None.

19865 **FUTURE DIRECTIONS**

19866 None.

19867 **SEE ALSO**

19868 *getsockopt*( ), *if_freenameindex*( ), *if_nameindex*( ), *if_nametoindex*( ), *setsockopt*( ), the Base
19869 Definitions volume of IEEE Std. 1003.1-200x, **<net/if.h>**

19870 **CHANGE HISTORY**

19871 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19872 **NAME**

19873 if_nameindex — return all network interface names and indexes

19874 **SYNOPSIS**

19875 `#include <net/if.h>`

19876 `struct if_nameindex *if_nameindex(void);`

19877 **DESCRIPTION**

19878 The *if_nameindex*( ) function shall return an array of *if_nameindex* structures, one structure per
19879 interface. The end of the array is indicated by a structure with an *if_index* field of zero and an
19880 *if_name* field of NULL.

19881 Applications should call *if_freenameindex*( ) to release the memory that may be dynamically
19882 allocated by this function, after they have finished using it.

19883 **RETURN VALUE**

19884 Array of structures identifying local interfaces. A NULL pointer is returned upon an error, with
19885 *errno* set to indicate the error.

19886 **ERRORS**

19887 The *if_nameindex*( ) function may fail if:

19888 [ENOBUFS] Insufficient resources are available to complete the function.

19889 **EXAMPLES**

19890 None.

19891 **APPLICATION USAGE**

19892 None.

19893 **RATIONALE**

19894 None.

19895 **FUTURE DIRECTIONS**

19896 None.

19897 **SEE ALSO**

19898 *getsockopt*( ), *if_freenameindex*( ), *if_indextoname*( ), *if_nametoindex*( ), *setsockopt*( ), the Base
19899 Definitions volume of IEEE Std. 1003.1-200x, **<net/if.h>**

19900 **CHANGE HISTORY**

19901 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19902 **NAME**

19903 if_nametoindex — map a network interface name to its corresponding index

19904 **SYNOPSIS**

19905 `#include <net/if.h>`

19906 `unsigned if_nametoindex(const char *`*`ifname`*`);`

19907 **DESCRIPTION**

19908 The *if_nametoindex*( ) function shall return the interface index corresponding to name *ifname*.

19909 **RETURN VALUE**

19910 The corresponding index if *ifname* is the name of an interface; otherwise, zero.

19911 **ERRORS**

19912 No errors are defined.

19913 **EXAMPLES**

19914 None.

19915 **APPLICATION USAGE**

19916 None.

19917 **RATIONALE**

19918 None.

19919 **FUTURE DIRECTIONS**

19920 None.

19921 **SEE ALSO**

19922 *getsockopt*( ), *if_freenameindex*( ), *if_indextoname*( ), *if_nameindex*( ), *setsockopt*( ), the Base
19923 Definitions volume of IEEE Std. 1003.1-200x, **<net/if.h>**

19924 **CHANGE HISTORY**

19925 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19926 **NAME**

19927 ilogb, ilogbf, ilogbl — return an unbiased exponent

19928 **SYNOPSIS**

19929 `#include <math.h>`

19930 `int ilogb(double x);`
19931 `int ilogbf(float x);`
19932 `int ilogbl(long double x);`

19933 **DESCRIPTION**

19934 These functions shall return the exponent part of *x*. Formally, the return value is the integral
19935 part of $\log_r |x|$ as a signed integral value, for non-zero *x*, where *r* is the radix of the machine's
19936 floating point arithmetic, which is the value of FLT_RADIX defined in **<float.h>**.

19937 **RETURN VALUE**

19938 Upon successful completion, these functions shall return the exponent part of *x* as a signed
19939 integer value.

19940 If *x* is zero, these functions shall return the value FP_ILOGB0; if *x* is infinite, they shall return the
19941 value {INT_MAX}; if *x* is a NaN, they shall return the value FP_ILOGBNAN; otherwise, they
19942 shall be equivalent to calling the corresponding *logb*( ) function and casting the returned value to
19943 type **int**.

19944 **ERRORS**

19945 These functions may fail if:

19946 [ERANGE] The value of *x* is 0.

19947 **EXAMPLES**

19948 None.

19949 **APPLICATION USAGE**

19950 None.

19951 **RATIONALE**

19952 None.

19953 **FUTURE DIRECTIONS**

19954 None.

19955 **SEE ALSO**

19956 *logb*( ), *scalb*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<float.h>**, **<math.h>**

19957 **CHANGE HISTORY**

19958 First released in Issue 4, Version 2.

19959 **Issue 5**

19960 Moved from X/OPEN UNIX extension to BASE.

19961 **Issue 6**

19962 The *ilogbf*( ) and *ilogbl*( ) functions are added for alignment with the ISO/IEC 9899:1999
19963 standard.

19964 **NAME**

19965 imaxabs — return absolute value

19966 **SYNOPSIS**

19967 `#include <inttypes.h>`

19968 `intmax_t imaxabs(intmax_t j);`

19969 **DESCRIPTION**

19970 CX The functionality described on this reference page is aligned with the ISO C standard. Any
19971 conflict between the requirements described here and the ISO C standard is unintentional. This
19972 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

19973 The *imaxabs*( ) function shall compute the absolute value of an integer *j*. If the result cannot be
19974 represented, the behavior is undefined.

19975 **RETURN VALUE**

19976 The *imaxabs*( ) function shall return the absolute value.

19977 **ERRORS**

19978 No errors are defined.

19979 **EXAMPLES**

19980 None.

19981 **APPLICATION USAGE**

19982 The absolute value of the most negative number cannot be represented in two's complement.

19983 **RATIONALE**

19984 None.

19985 **FUTURE DIRECTIONS**

19986 None.

19987 **SEE ALSO**

19988 *imaxdiv*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<inttypes.h>**

19989 **CHANGE HISTORY**

19990 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

19991 **NAME**

19992 imaxdiv — return quotient and remainder

19993 **SYNOPSIS**

19994 `#include <inttypes.h>`

19995 `imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);`

19996 **DESCRIPTION**

19997 CX The functionality described on this reference page is aligned with the ISO C standard. Any
19998 conflict between the requirements described here and the ISO C standard is unintentional. This
19999 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

20000 The *imaxdiv*( ) function shall compute *numer* / *denom* and *numer* % *denom* in a single operation.

20001 **RETURN VALUE**

20002 The *imaxdiv*( ) function shall return a structure of type **imaxdiv_t**, comprising both the quotient
20003 and the remainder. The structure shall contain (in either order) the members *quot* (the quotient)
20004 and *rem* (the remainder), each of which has type **intmax_t**.

20005 If either part of the result cannot be represented, the behavior is undefined.

20006 **ERRORS**

20007 No errors are defined.

20008 **EXAMPLES**

20009 None.

20010 **APPLICATION USAGE**

20011 None.

20012 **RATIONALE**

20013 None.

20014 **FUTURE DIRECTIONS**

20015 None.

20016 **SEE ALSO**

20017 *imaxabs*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<inttypes.h>**

20018 **CHANGE HISTORY**

20019 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

20020 **NAME**

20021 index — character string operations (**LEGACY**)

20022 **SYNOPSIS**

20023 XSI
```
#include <strings.h>
```

20024
```
char *index(const char *s, int c);
```

20025

20026 **DESCRIPTION**

20027 The *index*( ) function is identical to *strchr*( ).

20028 **RETURN VALUE**

20029 See *strchr*( ).

20030 **ERRORS**

20031 See *strchr*( ).

20032 **EXAMPLES**

20033 None.

20034 **APPLICATION USAGE**

20035 *strchr*( ) is preferred over this function.

20036 For maximum portability, it is recommended to replace the function call to *index*( ) as follows:

20037
```
#define index(a,b) strchr((a),(b))
```

20038 **RATIONALE**

20039 None.

20040 **FUTURE DIRECTIONS**

20041 This function may be withdrawn in a future version.

20042 **SEE ALSO**

20043 *strchr*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<strings.h>**

20044 **CHANGE HISTORY**

20045 First released in Issue 4, Version 2.

20046 **Issue 5**

20047 Moved from X/OPEN UNIX extension to BASE.

20048 **Issue 6**

20049 This function is marked LEGACY.

20050 **NAME**

20051 inet_addr, inet_lnaof (**LEGACY**), inet_makeaddr (**LEGACY**), inet_netof (**LEGACY**),
20052 inet_network (**LEGACY**), inet_ntoa — IPv4 address manipulation

20053 **SYNOPSIS**

20054 ```
#include <arpa/inet.h>
```

20055 ```
in_addr_t inet_addr(const char *cp);
```
20056 ```
in_addr_t inet_lnaof(struct in_addr in);
```
20057 ```
struct in_addr inet_makeaddr(in_addr_t net, in_addr_t lna);
```
20058 ```
in_addr_t inet_netof(struct in_addr in);
```
20059 ```
in_addr_t inet_network(const char *cp);
```
20060 ```
char *inet_ntoa(struct in_addr in);
```

20061 **DESCRIPTION**

20062 The *inet_addr*( ) function shall convert the string pointed to by *cp*, in the standard IPv4 dotted
20063 decimal notation, to an integer value suitable for use as an Internet address.

20064 The *inet_lnaof*( ) function shall take an Internet host address specified by *in* and extract the local
20065 network address part, in host byte order.

20066 The *inet_makeaddr*( ) function shall take the Internet network number specified by *net* and the
20067 local network address specified by *lna*, both in host byte order, and construct an Internet address
20068 from them.

20069 The *inet_netof*( ) function shall take an Internet host address specified by *in* and extract the
20070 network number part, in host byte order.

20071 The *inet_network*( ) function shall convert the string pointed to by *cp*, in the standard IPv4 dotted
20072 decimal notation, to an integer value suitable for use as an Internet network number.

20073 The *inet_ntoa*( ) function shall convert the Internet host address specified by *in* to a string in the
20074 Internet standard dot notation.

20075 All Internet addresses shall be returned in network order (bytes ordered from left to right).

20076 Values specified using IPv4 dotted decimal notation take one of the following forms:

20077 `a.b.c.d` When four parts are specified, each is interpreted as a byte of data and assigned,
20078 from left to right, to the four bytes of an Internet address.

20079 `a.b.c` When a three-part address is specified, the last part is interpreted as a 16-bit
20080 quantity and placed in the rightmost two bytes of the network address. This makes
20081 the three-part address format convenient for specifying Class B network addresses
20082 as **128.net.host**.

20083 `a.b` When a two-part address is supplied, the last part is interpreted as a 24-bit
20084 quantity and placed in the rightmost three bytes of the network address. This
20085 makes the two-part address format convenient for specifying Class A network
20086 addresses as **net.host**.

20087 `a` When only one part is given, the value is stored directly in the network address
20088 without any byte rearrangement.

20089 All numbers supplied as parts in IPv4 dotted decimal notation may be decimal, octal, or
20090 hexadecimal, as specified in the ISO C standard (that is, a leading `"0x"` or `"0X"` implies
20091 hexadecimal; otherwise, a leading `'0'` implies octal; otherwise, the number is interpreted as
20092 decimal).
```

20093 **RETURN VALUE**

20094 Upon successful completion, *inet_addr*( ) shall return the Internet address. Otherwise, it shall
20095 return (**in_addr_t**)(−1).

20096 The *inet_lnaof*( ) function shall return the local network address part.

20097 The *inet_makeaddr*( ) function shall return the constructed Internet address.

20098 The *inet_netof*( ) function shall return the network number.

20099 Upon successful completion, *inet_network*( ) shall return the converted Internet network number.
20100 Otherwise, it shall return (**in_addr_t**)(−1).

20101 The *inet_ntoa*( ) function shall return a pointer to the network address in Internet standard dot
20102 notation.

20103 **ERRORS**

20104 No errors are defined.

20105 **EXAMPLES**

20106 None.

20107 **APPLICATION USAGE**

20108 The *inet_lnaof*( ), *inet_makeaddr*( ), *inet_netof*( ), and *inet_network*( ) functions are marked LEGACY
20109 and should not be used by new applications.

20110 The return value of *inet_ntoa*( ) may point to static data that may be overwritten by subsequent
20111 calls to *inet_ntoa*( ).

20112 **RATIONALE**

20113 None.

20114 **FUTURE DIRECTIONS**

20115 The *inet_lnaof*( ), *inet_makeaddr*( ), *inet_netof*( ), and *inet_network*( ) functions may be withdrawn in |
20116 a future version. |

20117 **SEE ALSO**

20118 *endhostent*( ), *endnetent*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<arpa/inet.h>** |

20119 **CHANGE HISTORY**

20120 First released in Issue 6. Derived from the XNS, Issue 5.2 specification. |

20121 **NAME**

20122 inet_lnaof — IPv4 address manipulation

20123 **SYNOPSIS**

20124 `#include <arpa/inet.h>`

20125 `in_addr_t inet_lnaof(struct in_addr` *`in`*`);`

20126 **DESCRIPTION**

20127 Refer to *inet_addr*( ).

20128 **NAME**

20129 inet_makeaddr — IPv4 address manipulation

20130 **SYNOPSIS**

20131 `#include <arpa/inet.h>`

20132 `struct in_addr inet_makeaddr(in_addr_t` *`net`*`, in_addr_t` *`lna`*`);`

20133 **DESCRIPTION**

20134 Refer to *inet_addr*( ).

20135 **NAME**

20136 inet_netof — IPv4 address manipulation

20137 **SYNOPSIS**

20138 `#include <arpa/inet.h>`

20139 `in_addr_t inet_netof(struct in_addr` *`in`*`);`

20140 **DESCRIPTION**

20141 Refer to *inet_addr*( ).

20142 **NAME**

20143 inet_network — IPv4 address manipulation

20144 **SYNOPSIS**

20145 `#include <arpa/inet.h>`

20146 `in_addr_t inet_network(const char *`*`cp`*`);`

20147 **DESCRIPTION**

20148 Refer to *inet_addr*( ).

20149 **NAME**

20150 inet_ntoa — IPv4 address manipulation

20151 **SYNOPSIS**

20152 `#include <arpa/inet.h>`

20153 `char *inet_ntoa(struct in_addr` *`in`*`);`

20154 **DESCRIPTION**

20155 Refer to *inet_addr*( ).

20156 **NAME**

20157 inet_ntop, inet_pton — convert IPv4 and IPv6 addresses between binary and text form

20158 **SYNOPSIS**

20159 IP6
```
#include <arpa/inet.h>

const char *inet_ntop(int af, const void *restrict src,
    char *restrict dst, socklen_t size);
int inet_pton(int af, const char *restrict src, void *restrict dst);
```

20164 ***Notes to Reviewers***

20165 *This section with side shading will not appear in the final copy. - Ed.*

20166 D3, XSH, ERN 330 (AI 2000-05-024) To supply editing instructions regarding making these
20167 functions mandatory and shading IPv6-specific information.

20168 **DESCRIPTION**

20169 The *inet_ntop*( ) function converts a numeric address into a text string suitable for presentation.
20170 The *af* argument specifies the family of the address. This can be AF_INET or AF_INET6. The *src*
20171 argument points to a buffer holding an IPv4 address if the *af* argument is AF_INET, or an IPv6
20172 address if the *af* argument is AF_INET6. The *dst* argument points to a buffer where the function
20173 stores the resulting text string; it shall not be NULL. The *size* argument specifies the size of this
20174 buffer, which shall be large enough to hold the text string (INET_ADDRSTRLEN characters for
20175 IPv4, INET6_ADDRSTRLEN characters for IPv6).

20176 The *inet_pton*( ) function converts an address in its standard text presentation form into its
20177 numeric binary form. The *af* argument specifies the family of the address. The AF_INET and
20178 AF_INET6 address families are supported. The *src* argument points to the string being passed in.
20179 The *dst* argument points to a buffer into which the function stores the numeric address; this shall
20180 be large enough to hold the numeric address (32 bits for AF_INET, 128 bits for AF_INET6).

20181 If the *af* argument of *inet_pton*( ) is AF_INET, the *src* string shall be in the standard IPv4 dotted-
20182 decimal form:

20183 `ddd.ddd.ddd.ddd`

20184 where `"ddd"` is a one to three digit decimal number between 0 and 255 (see *inet_addr*( )). The
20185 *inet_pton*( ) function does not accept other formats (such as the octal numbers, hexadecimal
20186 numbers, and fewer than four numbers that *inet_addr*( ) accepts).

20187 If the *af* argument of *inet_pton*( ) is AF_INET6, the *src* string shall be in one of the following
20188 standard IPv6 text forms:

20189 1. The preferred form is `"x:x:x:x:x:x:x:x"`, where the `'x'`s are the hexadecimal values
20190 of the eight 16-bit pieces of the address. Leading zeros in individual fields can be omitted,
20191 but there shall be at least one numeral in every field.

20192 2. A string of contiguous zero fields in the preferred form can be shown as `"::"`. The `"::"`
20193 can only appear once in an address. Unspecified addresses (`"0:0:0:0:0:0:0:0"`) may
20194 be represented simply as `"::"`.

20195 3. A third form that is sometimes more convenient when dealing with a mixed environment
20196 of IPv4 and IPv6 nodes is `"x:x:x:x:x:x:d.d.d.d"`, where the `'x'`s are the
20197 hexadecimal values of the six high-order 16-bit pieces of the address, and the `'d'`s are the
20198 decimal values of the four low-order 8-bit pieces of the address (standard IPv4
20199 representation).

20200 **Note:** A more extensive description of the standard representations of IPv6 addresses can
20201 be found in RFC 2373.

20202 **RETURN VALUE**

20203 The *inet_ntop*( ) function shall return a pointer to the buffer containing the text string if the |
20204 conversion succeeds, and NULL otherwise, and set *errno* to indicate the error. |

20205 The *inet_pton*( ) function shall return 1 if the conversion succeeds, with the address pointed to by
20206 *dst* in network byte order. It shall return 0 if the input is not a valid IPv4 dotted-decimal string or
20207 a valid IPv6 address string, or –1 with *errno* set to [EAFNOSUPPORT] if the *af* argument is
20208 unknown.

20209 **ERRORS**

20210 The *inet_ntop*( ) and *inet_pton*( ) functions shall fail if:

20211 [EAFNOSUPPORT]
20212 The *af* argument is invalid.

20213 [ENOSPC] The size of the *inet_ntop*( ) result buffer is inadequate.

20214 **EXAMPLES**

20215 None.

20216 **APPLICATION USAGE**

20217 None.

20218 **RATIONALE**

20219 None.

20220 **FUTURE DIRECTIONS**

20221 None.

20222 **SEE ALSO**

20223 The Base Definitions volume of IEEE Std. 1003.1-200x, **<arpa/inet.h>** |

20224 **CHANGE HISTORY**

20225 First released in Issue 6. Derived from the XNS, Issue 5.2 specification. |

20226 Marked as part of the IPv6 option. |

20227 The **restrict** keyword is added to the *inet_ntop*( ) and *inet_pton*( ) prototypes for alignment with |
20228 the ISO/IEC 9899:1999 standard. |

NAME

initstate, random, setstate, srandom — pseudorandom number functions

SYNOPSIS

XSI

```
#include <stdlib.h>

char *initstate(unsigned seed, char *state, size_t size);
long random(void);
char *setstate(const char *state);
void srandom(unsigned seed);
```

DESCRIPTION

The *random*( ) function uses a non-linear additive feedback random-number generator employing a default state array size of 31 **long** integers to return successive pseudo-random numbers in the range from 0 to 2^{31}–1. The period of this random-number generator is approximately 16 x (2^{31}–1). The size of the state array determines the period of the random-number generator. Increasing the state array size increases the period.

With 256 bytes of state information, the period of the random-number generator is greater than 2^{69}.

Like *rand*( ), *random*( ) produces by default a sequence of numbers that can be duplicated by calling *srandom*( ) with 1 as the seed.

The *srandom*( ) function initializes the current state array using the value of *seed*.

The *initstate*( ) and *setstate*( ) functions handle restarting and changing random-number generators. The *initstate*( ) function allows a state array, pointed to by the *state* argument, to be initialized for future use. The *size* argument, which specifies the size in bytes of the state array, is used by *initstate*( ) to decide what type of random-number generator to use; the larger the state array, the more random the numbers. Values for the amount of state information are 8, 32, 64, 128, and 256 bytes. Other values greater than 8 bytes are rounded down to the nearest one of these values. If *initstate*( ) is called with $8 \le size < 32$, then *random*( ) uses a simple linear congruential random number generator. The *seed* argument specifies a starting point for the random-number sequence and provides for restarting at the same point. The *initstate*( ) function shall return a pointer to the previous state information array.

If *initstate*( ) has not been called, then *random*( ) behaves as though *initstate*( ) had been called with *seed*=1 and *size*=128.

Once a state has been initialized, *setstate*( ) allows switching between state arrays. The array defined by the *state* argument is used for further random-number generation until *initstate*( ) is called or *setstate*( ) is called again. The *setstate*( ) function shall return a pointer to the previous state array.

RETURN VALUE

If *initstate*( ) is called with *size* less than 8, it shall return NULL.

The *random*( ) function shall return the generated pseudo-random number.

The *srandom*( ) function shall return no value.

Upon successful completion, *initstate*( ) and *setstate*( ) shall return a pointer to the previous state array; otherwise, a null pointer shall be returned.

20271 **ERRORS**

20272 No errors are defined.

20273 **EXAMPLES**

20274 None.

20275 **APPLICATION USAGE**

20276 After initialization, a state array can be restarted at a different point in one of two ways:

20277 1. The *initstate*( ) function can be used, with the desired seed, state array, and size of the
20278 array.

20279 2. The *setstate*( ) function, with the desired state, can be used, followed by *srandom*( ) with the
20280 desired seed. The advantage of using both of these functions is that the size of the state
20281 array does not have to be saved once it is initialized.

20282 Although some implementations of *random*( ) have written messages to standard error, such
20283 implementations do not conform to this volume of IEEE Std. 1003.1-200x.

20284 Issue 5 restores the historical behavior of this function.

20285 Threaded applications should use *rand_r*( ), *erand48*( ), *nrand48*( ), or *jrand48*( ) instead of
20286 *random*( ) when an independent random number sequence in multiple threads is required.

20287 **RATIONALE**

20288 None.

20289 **FUTURE DIRECTIONS**

20290 None.

20291 **SEE ALSO**

20292 *drand48*( ), *rand*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>** |

20293 **CHANGE HISTORY**

20294 First released in Issue 4, Version 2.

20295 **Issue 5**

20296 Moved from X/OPEN UNIX extension to BASE.

20297 In the DESCRIPTION, the phrase ‘‘values smaller than 8’’ is replaced with ‘‘values greater than
20298 or equal to 8, or less than 32’’, ‘‘*size*<8’’ is replaced with ‘‘8≤*size* <32’’, and a new first paragraph
20299 is added to the RETURN VALUE section. A note is added to the APPLICATION USAGE
20300 indicating that these changes restore the historical behavior of the function.

20301 **Issue 6**

20302 In the DESCRIPTION, duplicate text ‘‘For values greater than or equal to 8 . . .’’ is removed.

NAME

insque, remque — insert or remove an element in a queue

SYNOPSIS

XSI
```
#include <search.h>

void insque(void *element, void *pred);
void remque(void *element);
```

DESCRIPTION

The *insque*( ) and *remque*( ) functions manipulate queues built from doubly-linked lists. The queue can be either circular or linear. An application using *insque*( ) or *remque*( ) shall ensure it defines a structure in which the first two members of the structure are pointers to the same type of structure, and any further members are application-specific. The first member of the structure is a forward pointer to the next entry in the queue. The second member is a backward pointer to the previous entry in the queue. If the queue is linear, the queue is terminated with null pointers. The names of the structure and of the pointer members are not subject to any special restriction.

The *insque*( ) function inserts the element pointed to by *element* into a queue immediately after the element pointed to by *pred*.

The *remque*( ) function removes the element pointed to by *element* from a queue.

If the queue is to be used as a linear list, invoking *insque*(&*element*, NULL), where *element* is the initial element of the queue, shall initialize the forward and backward pointers of *element* to null pointers.

If the queue is to be used as a circular list, the application shall ensure it initializes the forward pointer and the backward pointer of the initial element of the queue to the element's own address.

RETURN VALUE

The *insque*( ) and *remque*( ) functions do not return a value.

ERRORS

No errors are defined.

EXAMPLES

Creating a Linear Linked List

The following example creates a linear linked list.

```
#include <search.h>
...
struct myque element1;
struct myque element2;

char *data1 = "DATA1";
char *data2 = "DATA2";
...
element1.data = data1;
element2.data = data2;

insque (&element1, NULL);
insque (&element2, &element1);
```

20346 **Creating a Circular Linked List**

20347 The following example creates a circular linked list.

```
20348 #include <search.h>
20349 ...
20350 struct myque element1;
20351 struct myque element2;
20352 char *data1 = "DATA1";
20353 char *data2 = "DATA2";
20354 ...
20355 element1.data = data1;
20356 element2.data = data2;
20357 element1.fwd = &element1;
20358 element1.bck = &element1;
20359 insque (&element2, &element1);
```

20360 **Removing an Element**

20361 The following example removes the element pointed to by *element1*.

```
20362 #include <search.h>
20363 ...
20364 struct myque element1;
20365 ...
20366 remque (&element1);
```

20367 **APPLICATION USAGE**

20368 The historical implementations of these functions described the arguments as being of type
20369 **struct qelem*** rather than as being of type **void*** as defined here. In those implementations,
20370 **struct qelem** was commonly defined in **<search.h>** as:

```
20371 struct qelem {
20372     struct qelem  *q_forw;
20373     struct qelem  *q_back;
20374 };
```

20375 Applications using these functions, however, were never able to use this structure directly since
20376 it provided no room for the actual data contained in the elements. Most applications defined
20377 structures that contained the two pointers as the initial elements and also provided space for, or
20378 pointers to, the object's data. Applications that used these functions to update more than one
20379 type of table also had the problem of specifying two or more different structures with the same
20380 name, if they literally used **struct qelem** as specified.

20381 As described here, the implementations were actually expecting a structure type where the first
20382 two members were forward and backward pointers to structures. With C compilers that didn't
20383 provide function prototypes, applications used structures as specified in the DESCRIPTION
20384 above and the compiler did what the application expected.

20385 If this method had been carried forward with an ISO C standard compiler and the historical
20386 function prototype, most applications would have to be modified to cast pointers to the
20387 structures actually used to be pointers to **struct qelem** to avoid compilation warnings. By
20388 specifying **void*** as the argument type, applications do not need to change (unless they
20389 specifically referenced **struct qelem** and depended on it being defined in **<search.h>**).

20390 **RATIONALE**

20391 None.

20392 **FUTURE DIRECTIONS**

20393 None.

20394 **SEE ALSO**

20395 The Base Definitions volume of IEEE Std. 1003.1-200x, **<search.h>**

20396 **CHANGE HISTORY**

20397 First released in Issue 4, Version 2.

20398 **Issue 5**

20399 Moved from X/OPEN UNIX extension to BASE.

20400 **Issue 6**

20401 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

20402 **NAME**

20403 ioctl — control a STREAMS device (**STREAMS**)

20404 **SYNOPSIS**

20405 XSR
```
#include <stropts.h>

int ioctl(int fildes, int request, ... /* arg */);
```
20406

20407

20408 **DESCRIPTION**

20409 The *ioctl*( ) function performs a variety of control functions on STREAMS devices. For non-
20410 STREAMS devices, the functions performed by this call are unspecified. The *request* argument
20411 and an optional third argument (with varying type) are passed to and interpreted by the
20412 appropriate part of the STREAM associated with *fildes*.

20413 The *fildes* argument is an open file descriptor that refers to a device.

20414 The *request* argument selects the control function to be performed and shall depend on the
20415 STREAMS device being addressed.

20416 The *arg* argument represents additional information that is needed by this specific STREAMS
20417 device to perform the requested function. The type of *arg* depends upon the particular control
20418 request, but it is either an integer or a pointer to a device-specific data structure.

20419 The *ioctl*( ) commands applicable to STREAMS, their arguments, and error conditions that apply
20420 to each individual command are described below.

20421 The following *ioctl*( ) commands, with error values indicated, are applicable to all STREAMS
20422 files:

20423 I_PUSH Pushes the module whose name is pointed to by *arg* onto the top of the
20424 current STREAM, just below the STREAM head. It then calls the *open*( )
20425 function of the newly-pushed module.

20426 The *ioctl*( ) function with the I_PUSH command shall fail if:

20427 [EINVAL] Invalid module name. |

20428 [ENXIO] Open function of new module failed. |

20429 [ENXIO] Hangup received on *fildes*. |

20430 I_POP Removes the module just below the STREAM head of the STREAM pointed to
20431 by *fildes*. The *arg* argument should be 0 in an I_POP request.

20432 The *ioctl*( ) function with the I_POP command shall fail if:

20433 [EINVAL] No module present in the STREAM. |

20434 [ENXIO] Hangup received on *fildes*. |

20435 I_LOOK Retrieves the name of the module just below the STREAM head of the
20436 STREAM pointed to by *fildes*, and places it in a character string pointed to by
20437 *arg*. The buffer pointed to by *arg* should be at least FMNAMESZ+1 bytes long,
20438 where FMNAMESZ is defined in **<stropts.h>**.

20439 The *ioctl*( ) function with the I_LOOK command shall fail if:

20440 [EINVAL] No module present in the STREAM. |

20441 I_FLUSH This request flushes read and/or write queues, depending on the value of *arg*.
20442 Valid *arg* values are:

20443 FLUSHR Flush all read queues.

20444 FLUSHW Flush all write queues.

20445 FLUSHRW Flush all read and all write queues.

20446 The *ioctl*( ) function with the I_FLUSH command shall fail if:

20447 [EINVAL] Invalid *arg* value.

20448 [EAGAIN] or [ENOSR]
20449 Unable to allocate buffers for flush message.

20450 [ENXIO] Hangup received on *fildes*.

20451 I_FLUSHBAND Flushes a particular band of messages. The *arg* argument points to a **bandinfo**
20452 structure. The *bi_flag* member may be one of FLUSHR, FLUSHW, or
20453 FLUSHRW as described above. The *bi_pri* member determines the priority
20454 band to be flushed.

20455 I_SETSIG Requests that the STREAMS implementation send the SIGPOLL signal to the
20456 calling process when a particular event has occurred on the STREAM
20457 associated with *fildes*. I_SETSIG supports an asynchronous processing
20458 capability in STREAMS. The value of *arg* is a bitmask that specifies the events
20459 for which the process should be signaled. It is the bitwise-inclusive OR of any
20460 combination of the following constants:

20461 S_RDNORM A normal (priority band set to 0) message has arrived at the
20462 head of a STREAM head read queue. A signal shall be
20463 generated even if the message is of zero length.

20464 S_RDBAND A message with a non-zero priority band has arrived at the
20465 head of a STREAM head read queue. A signal shall be
20466 generated even if the message is of zero length.

20467 S_INPUT A message, other than a high-priority message, has arrived
20468 at the head of a STREAM head read queue. A signal shall be
20469 generated even if the message is of zero length.

20470 S_HIPRI A high-priority message is present on a STREAM head read
20471 queue. A signal shall be generated even if the message is of
20472 zero length.

20473 S_OUTPUT The write queue for normal data (priority band 0) just
20474 below the STREAM head is no longer full. This notifies the
20475 process that there is room on the queue for sending (or
20476 writing) normal data downstream.

20477 S_WRNORM Same as S_OUTPUT.

20478 S_WRBAND The write queue for a non-zero priority band just below the
20479 STREAM head is no longer full. This notifies the process
20480 that there is room on the queue for sending (or writing)
20481 priority data downstream.

20482 S_MSG A STREAMS signal message that contains the SIGPOLL
20483 signal has reached the front of the STREAM head read
20484 queue.

20485 S_ERROR Notification of an error condition has reached the STREAM
20486 head.

S_HANGUP Notification of a hangup has reached the STREAM head.

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.

If *arg* is 0, the calling process shall be unregistered and shall not receive further SIGPOLL signals for the stream associated with *fildes*.

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.

The *ioctl*( ) function with the I_SETSIG command shall fail if:

[EINVAL] The value of *arg* is invalid. |

[EINVAL] The value of *arg* is 0 and the calling process is not registered to receive the SIGPOLL signal. |

[EAGAIN] There were insufficient resources to store the signal request. |

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 *arg*, where the events are those specified in the description of I_SETSIG above.

The *ioctl*( ) function with the I_GETSIG command shall fail if:

[EINVAL] Process is not registered to receive the SIGPOLL signal. |

I_FIND This request compares the names of all modules currently present in the STREAM to the name pointed to by *arg*, and returns 1 if the named module is present in the STREAM, or returns 0 if the named module is not present.

The *ioctl*( ) function with the I_FIND command shall fail if:

[EINVAL] *arg* does not contain a valid module name. |

I_PEEK This request allows a process to retrieve the information in the first message on the STREAM head read queue without taking the message off the queue. It is analogous to *getmsg*( ) except that this command does not remove the message from the queue. The *arg* argument points to a **strpeek** structure.

The application shall ensure that the *maxlen* member in the **ctlbuf** and **databuf** **strbuf** structures is set to the number of bytes of control information and/or data information, respectively, to retrieve. The *flags* member may be marked RS_HIPRI or 0, as described by *getmsg*( ). If the process sets *flags* to RS_HIPRI, for example, I_PEEK shall only look for a high-priority message on the STREAM head read queue.

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 *flags* 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 *flags* contains the value RS_HIPRI or 0.

I_SRDOPT Sets the read mode using the value of the argument *arg*. Read modes are described in *read*( ). Valid *arg* flags are:

RNORM Byte-stream mode, the default.

RMSGD Message-discard mode.

RMSGN Message-nondiscard mode.

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.

In addition, treatment of control messages by the STREAM head may be changed by setting any of the following flags in *arg*:

RPROTNORM Fail *read*( ) with [EBADMSG] if a message containing a control part is at the front of the STREAM head read queue.

RPROTDAT Deliver the control part of a message as data when a process issues a *read*( ).

RPROTDIS Discard the control part of a message, delivering any data portion, when a process issues a *read*( ).

The *ioctl*( ) function with the I_SRDOPT command shall fail if:

[EINVAL] The *arg* argument is not valid.

I_GRDOPT Returns the current read mode setting as, described above, in an **int** pointed to by the argument *arg*. Read modes are described in *read*( ).

I_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 *arg*. The return value for the command is the number of messages on the STREAM head read queue. For example, if 0 is returned in *arg*, but the *ioctl*( ) return value is greater than 0, this indicates that a zero-length message is next on the queue.

I_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 *arg* argument points to a **strfdinsert** structure.

The application shall ensure that the *len* 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 *fildes* member specifies the file descriptor of the other STREAM, and the *offset* member, which must be suitably aligned for use as a **t_uscalar_t**, specifies the offset from the start of the control buffer where I_FDINSERT shall store a **t_uscalar_t** whose interpretation is specific to the STREAM end. The application shall ensure that the *len* 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.

The *flags* member specifies the type of message to be created. A normal message is created if *flags* is set to 0, and a high-priority message is created if *flags* is set to RS_HIPRI. For non-priority messages, I_FDINSERT shall block if the STREAM write queue is full due to internal flow control conditions. For priority messages, I_FDINSERT does not block on this condition. For non-priority messages, I_FDINSERT does not block when the write queue is full

and O_NONBLOCK is set. Instead, it fails and sets *errno* to [EAGAIN].

I_FDINSERT also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the STREAM, regardless of priority or whether O_NONBLOCK has been specified. No partial message is sent.

The *ioctl*( ) function with the I_FDINSERT command shall fail if:

[EAGAIN] A non-priority message is specified, the O_NONBLOCK flag is set, and the STREAM write queue is full due to internal flow control conditions.

[EAGAIN] or [ENOSR]
Buffers cannot be allocated for the message that is to be created.

[EINVAL] One of the following:

— The *fildes* member of the **strfdinsert** structure is not a valid, open STREAM file descriptor.

— The size of a **t_uscalar_t** plus *offset* is greater than the *len* member for the buffer specified through **ctlbuf**.

— The *offset* member does not specify a properly-aligned location in the data buffer.

— An undefined value is stored in *flags*.

[ENXIO] Hangup received on the STREAM identified by either the *fildes* argument or the *fildes* member of the **strfdinsert** structure.

[ERANGE] The *len* member for the buffer specified through **databuf** does not fall within the range specified by the maximum and minimum packet sizes of the topmost STREAM module or the *len* member for the buffer specified through **databuf** is larger than the maximum configured size of the data part of a message; or the *len* member for the buffer specified through **ctlbuf** is larger than the maximum configured size of the control part of a message.

I_STR Constructs an internal STREAMS *ioctl*( ) message from the data pointed to by *arg*, and sends that message downstream.

This mechanism is provided to send *ioctl*( ) requests to downstream modules and drivers. It allows information to be sent with *ioctl*( ), and returns to the process any information sent upstream by the downstream recipient. I_STR blocks until the system responds with either a positive or negative acknowledgement message, or until the request times out after some period of time. If the request times out, it fails with *errno* set to [ETIME].

At most, one I_STR can be active on a STREAM. Further I_STR calls shall block until the active I_STR completes at the STREAM head. The default timeout interval for these requests is 15 seconds. The O_NONBLOCK flag has no effect on this call.

To send requests downstream, the application shall ensure that *arg* points to a **strioctl** structure.

The *ic_cmd* member is the internal *ioctl*() command intended for a downstream module or driver and *ic_timout* is the number of seconds (–1=infinite, 0=use implementation-defined timeout interval, >0=as specified) an I_STR request shall wait for acknowledgement before timing out. *ic_len* is the number of bytes in the data argument, and *ic_dp* is a pointer to the data argument. The *ic_len* member has two uses: on input, it contains the length of the data argument passed in, and on return from the command, it contains the number of bytes being returned to the process (the buffer pointed to by *ic_dp* should be large enough to contain the maximum amount of data that any module or the driver in the STREAM can return).

The STREAM head shall convert the information pointed to by the **strioctl** structure to an internal *ioctl*() command message and sends it downstream.

The *ioctl*() function with the I_STR command shall fail if:

[EAGAIN] or [ENOSR]
: Unable to allocate buffers for the *ioctl*() message.

[EINVAL]
: The *ic_len* member is less than 0 or larger than the maximum configured size of the data part of a message, or *ic_timout* is less than –1.

[ENXIO]
: Hangup received on *fildes*.

[ETIME]
: A downstream *ioctl*() timed out before acknowledgement was received.

An I_STR can also fail while waiting for an acknowledgement if a message indicating an error or a hangup is received at the STREAM head. In addition, an error code can be returned in the positive or negative acknowledgement message, in the event the *ioctl*() command sent downstream fails. For these cases, I_STR fails with *errno* set to the value in the message.

I_SWROPT
: Sets the write mode using the value of the argument *arg*. Valid bit settings for *arg* are:

  SNDZERO
  : Send a zero-length message downstream when a *write*() of 0 bytes occurs. To not send a zero-length message when a *write*() of 0 bytes occurs, the application shall ensure that this bit is not set in *arg* (for example, *arg* would be set to 0).

  The *ioctl*() function with the I_SWROPT command shall fail if:

  [EINVAL]
  : *arg* is not the above value.

I_GWROPT
: Returns the current write mode setting, as described above, in the **int** that is pointed to by the argument *arg*.

I_SENDFD
: I_SENDFD creates a new reference to the open file description associated with the file descriptor *arg*, and writes a message on the STREAMS-based pipe *fildes* containing this reference, together with the user ID and group ID of the calling process.

  The *ioctl*() function with the I_SENDFD command shall fail if:

  [EAGAIN]
  : The sending STREAM is unable to allocate a message block to contain the file pointer; or the read queue of the receiving STREAM head is full and cannot accept the message sent by I_SENDFD.

[EBADF] The *arg* argument is not a valid, open file descriptor.

[EINVAL] The *fildes* argument is not connected to a STREAM pipe.

[ENXIO] Hangup received on *fildes*.

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 *arg* argument is a pointer to a **strrecvfd** data structure as defined in **<stropts.h>**.

The *fd* member is a file descriptor. The *uid* and *gid* members are the effective user ID and effective group ID, respectively, of the sending process.

If O_NONBLOCK is not set, I_RECVFD blocks until a message is present at the STREAM head. If O_NONBLOCK is set, I_RECVFD fails with *errno* set to [EAGAIN] if no message is present at the STREAM head.

If the message at the STREAM head is a message sent by an I_SENDFD, a new file descriptor is allocated for the open file descriptor referenced in the message. The new file descriptor is placed in the *fd* member of the **strrecvfd** structure pointed to by *arg*.

The *ioctl*( ) function with the I_RECVFD command shall fail if:

[EAGAIN] A message is not present at the STREAM head read queue and the O_NONBLOCK flag is set.

[EBADMSG] The message at the STREAM head read queue is not a message containing a passed file descriptor.

[EMFILE] The process has the maximum number of file descriptors currently open that it is allowed.

[ENXIO] Hangup received on *fildes*.

I_LIST This request allows the process to list all the module names on the STREAM, up to and including the topmost driver name. If *arg* is a null pointer, the return value is the number of modules, including the driver, that are on the STREAM pointed to by *fildes*. This lets the process allocate enough space for the module names. Otherwise, it should point to a **str_list** structure.

The *sl_nmods* member indicates the number of entries the process has allocated in the array. Upon return, the *sl_modlist* member of the **str_list** structure contains the list of module names, and the number of entries that have been filled into the *sl_modlist* array is found in the *sl_nmods* member (the number includes the number of modules including the driver). The return value from *ioctl*( ) is 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 (*sl_nmods*) is satisfied.

The *ioctl*( ) function with the I_LIST command shall fail if:

[EINVAL] The *sl_nmods* member is less than 1.

[EAGAIN] or [ENOSR]
Unable to allocate buffers.

I_ATMARK This request allows the process to see if the message at the head of the STREAM head read queue is marked by some module downstream. The *arg*

20710 argument determines how the checking is done when there may be multiple
20711 marked messages on the STREAM head read queue. It may take on the
20712 following values:

20713 ANYMARK Check if the message is marked.

20714 LASTMARK Check if the message is the last one marked on the queue.

20715 The bitwise-inclusive OR of the flags ANYMARK and LASTMARK is
20716 permitted.

20717 The return value is 1 if the mark condition is satisfied; otherwise, the value is
20718 0.

20719 The *ioctl*( ) function with the I_ATMARK command shall fail if:

20720 [EINVAL] Invalid *arg* value. |

20721 I_CKBAND Check if the message of a given priority band exists on the STREAM head
20722 read queue. This returns 1 if a message of the given priority exists, 0 if no such
20723 message exists, or –1 on error. *arg* should be of type **int**.

20724 The *ioctl*( ) function with the I_CKBAND command shall fail if:

20725 [EINVAL] Invalid *arg* value. |

20726 I_GETBAND Return the priority band of the first message on the STREAM head read queue
20727 in the integer referenced by *arg*.

20728 The *ioctl*( ) function with the I_GETBAND command shall fail if:

20729 [ENODATA] No message on the STREAM head read queue. |

20730 I_CANPUT Check if a certain band is writable. *arg* is set to the priority band in question.
20731 The return value is 0 if the band is flow-controlled, 1 if the band is writable, or
20732 –1 on error.

20733 The *ioctl*( ) function with the I_CANPUT command shall fail if:

20734 [EINVAL] Invalid *arg* value. |

20735 I_SETCLTIME This request allows the process to set the time the STREAM head shall delay
20736 when a STREAM is closing and there is data on the write queues. Before
20737 closing each module or driver, if there is data on its write queue, the STREAM
20738 head shall delay for the specified amount of time to allow the data to drain. If,
20739 after the delay, data is still present, it shall be flushed. The *arg* argument is a
20740 pointer to an integer specifying the number of milliseconds to delay, rounded
20741 up to the nearest valid value. If I_SETCLTIME is not performed on a STREAM, |
20742 an implementation-defined default timeout interval is used. |

20743 The *ioctl*( ) function with the I_SETCLTIME command shall fail if:

20744 [EINVAL] Invalid *arg* value. |

20745 I_GETCLTIME This request returns the close time delay in the integer pointed to by *arg*.

Multiplexed STREAMS Configurations

The following commands are used for connecting and disconnecting multiplexed STREAMS configurations. These commands use an implementation-defined default timeout interval.

I_LINK Connects two STREAMs, where *fildes* is the file descriptor of the STREAM connected to the multiplexing driver, and *arg* is the file descriptor of the STREAM connected to another driver. The STREAM designated by *arg* 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 returns a multiplexer ID number (an identifier used to disconnect the multiplexer; see I_UNLINK) on success, and –1 on failure.

The *ioctl*( ) function with the I_LINK command shall fail if:

[ENXIO] Hangup received on *fildes*.

[ETIME] Timeout before acknowledgement message was received at STREAM head.

[EAGAIN] or [ENOSR]
Unable to allocate STREAMS storage to perform the I_LINK.

[EBADF] The *arg* argument is not a valid, open file descriptor.

[EINVAL] The *fildes* argument does not support multiplexing; or *arg* 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.

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 *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_LINK fails with *errno* set to the value in the message.

I_UNLINK Disconnects the two STREAMs specified by *fildes* and *arg*. *fildes* is the file descriptor of the STREAM connected to the multiplexing driver. The *arg* argument is the multiplexer ID number that was returned by the I_LINK *ioctl*( ) command when a STREAM was connected downstream from the multiplexing driver. If *arg* is MUXID_ALL, then all STREAMs that were connected to *fildes* are disconnected. As in I_LINK, this command requires acknowledgement.

The *ioctl*( ) function with the I_UNLINK command shall fail if:

[ENXIO] Hangup received on *fildes*.

[ETIME] Timeout before acknowledgement message was received at STREAM head.

[EAGAIN] or [ENOSR]
Unable to allocate buffers for the acknowledgement message.

[EINVAL] Invalid multiplexer ID number.

20789 An I_UNLINK can also fail while waiting for the multiplexing driver to
20790 acknowledge the request if a message indicating an error or a hangup is
20791 received at the STREAM head of *fildes*. In addition, an error code can be
20792 returned in the positive or negative acknowledgement message. For these
20793 cases, I_UNLINK fails with *errno* set to the value in the message.

20794 I_PLINK Creates a *persistent connection* between two STREAMs, where *fildes* is the file
20795 descriptor of the STREAM connected to the multiplexing driver, and *arg* is the
20796 file descriptor of the STREAM connected to another driver. This call creates a
20797 persistent connection which can exist even if the file descriptor *fildes*
20798 associated with the upper STREAM to the multiplexing driver is closed. The
20799 STREAM designated by *arg* gets connected via a persistent connection below
20800 the multiplexing driver. I_PLINK requires the multiplexing driver to send an
20801 acknowledgement message to the STREAM head. This call returns a
20802 multiplexer ID number (an identifier that may be used to disconnect the
20803 multiplexer; see I_PUNLINK) on success, and –1 on failure.

20804 The *ioctl*( ) function with the I_PLINK command shall fail if:

20805 [ENXIO] Hangup received on *fildes*. |

20806 [ETIME] Timeout before acknowledgement message was received at |
20807 STREAM head.

20808 [EAGAIN] or [ENOSR] |
20809 Unable to allocate STREAMS storage to perform the
20810 I_PLINK.

20811 [EBADF] The *arg* argument is not a valid, open file descriptor. |

20812 [EINVAL] The *fildes* argument does not support multiplexing; or *arg* is |
20813 not a STREAM or is already connected downstream from a
20814 multiplexer; or the specified I_PLINK operation would
20815 connect the STREAM head in more than one place in the
20816 multiplexed STREAM.

20817 An I_PLINK can also fail while waiting for the multiplexing driver to
20818 acknowledge the request, if a message indicating an error or a hangup is
20819 received at the STREAM head of *fildes*. In addition, an error code can be
20820 returned in the positive or negative acknowledgement message. For these
20821 cases, I_PLINK fails with *errno* set to the value in the message.

20822 I_PUNLINK Disconnects the two STREAMs specified by *fildes* and *arg* from a persistent
20823 connection. The *fildes* argument is the file descriptor of the STREAM
20824 connected to the multiplexing driver. The *arg* argument is the multiplexer ID
20825 number that was returned by the I_PLINK *ioctl*( ) command when a STREAM
20826 was connected downstream from the multiplexing driver. If *arg* is
20827 MUXID_ALL, then all STREAMs which are persistent connections to *fildes* are
20828 disconnected. As in I_PLINK, this command requires the multiplexing driver
20829 to acknowledge the request.

20830 The *ioctl*( ) function with the I_PUNLINK command shall fail if:

20831 [ENXIO] Hangup received on *fildes*. |

20832 [ETIME] Timeout before acknowledgement message was received at |
20833 STREAM head.

[EAGAIN] or [ENOSR] | Unable to allocate buffers for the acknowledgement message. |

[EINVAL] Invalid multiplexer ID number. |

An I_PUNLINK 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 *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_PUNLINK fails with *errno* set to the value in the message.

RETURN VALUE

Upon successful completion, *ioctl*( ) shall return a value other than –1 that depends upon the STREAMS device control function. Otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

Under the following general conditions, *ioctl*( ) shall fail if:

[EBADF] The *fildes* argument is not a valid open file descriptor. |

[EINTR] A signal was caught during the *ioctl*( ) operation. |

[EINVAL] The STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer. |

If an underlying device driver detects an error, then *ioctl*( ) shall fail if:

[EINVAL] The *request* or *arg* argument is not valid for this device. |

[EIO] Some physical I/O error has occurred. |

[ENOTTY] The *fildes* argument is not associated with a STREAMS device that accepts control functions. |

[ENXIO] The *request* and *arg* arguments are valid for this device driver, but the service requested cannot be performed on this particular sub-device. |

[ENODEV] The *fildes* argument refers to a valid STREAMS device, but the corresponding device driver does not support the *ioctl*( ) function. |

If a STREAM is connected downstream from a multiplexer, any *ioctl*( ) command except I_UNLINK and I_PUNLINK shall set *errno* to [EINVAL].

EXAMPLES

None.

APPLICATION USAGE

The implementation-defined timeout interval for STREAMS has historically been 15 seconds. |

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *fcntl*( ), *getmsg*( ), *open*( ), *pipe*( ), *poll*( ), *putmsg*( ), *read*( ), *sigaction*( ), *write*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stropts.h>**, Section 2.6 (on page 539) |

20875 **CHANGE HISTORY**

20876 First released in Issue 4, Version 2.

20877 **Issue 5**

20878 Moved from X/OPEN UNIX extension to BASE.

20879 **Issue 6**

20880 The Open Group corrigenda item U028/4 has been applied, correcting text in the I_FDINSERT,
20881 [EINVAL] case to refer to *ctlbuf*.

20882 This function is marked as part of the XSI STREAMS Option Group.

20883 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

20884 **NAME**

20885 isalnum — test for an alphanumeric character

20886 **SYNOPSIS**

20887 ```
#include <ctype.h>
```

20888 ```
int isalnum(int c);
```

20889 **DESCRIPTION**

20890 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20891 conflict between the requirements described here and the ISO C standard is unintentional. This
20892 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

20893 The *isalnum*( ) function tests whether *c* is a character of class **alpha** or **digit** in the program's |
20894 current locale; see the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale. |

20895 In all cases *c* is an **int**, the value of which the application shall ensure is representable as an
20896 **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the
20897 behavior is undefined.

20898 **RETURN VALUE**

20899 The *isalnum*( ) function shall return non-zero if *c* is an alphanumeric character; otherwise, it shall
20900 return 0.

20901 **ERRORS**

20902 No errors are defined.

20903 **EXAMPLES**

20904 None.

20905 **APPLICATION USAGE**

20906 To ensure applications portability, especially across natural languages, only this function and
20907 those listed in the SEE ALSO section should be used for character classification.

20908 **RATIONALE**

20909 None.

20910 **FUTURE DIRECTIONS**

20911 None.

20912 **SEE ALSO**

20913 *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *isxdigit*( ),
20914 *setlocale*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>**, **<stdio.h>**, the Base |
20915 Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale |

20916 **CHANGE HISTORY**

20917 First released in Issue 1. Derived from Issue 1 of the SVID. |

20918 **Issue 4**

20919 The text of the DESCRIPTION and RETURN VALUE sections is revised, although there are no
20920 functional differences between this issue and Issue 3. Operation in the C locale is no longer
20921 described explicitly on this reference page.

20922 **Issue 6**

20923 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

20924 **NAME**

20925 isalpha — test for an alphabetic character

20926 **SYNOPSIS**

20927
```
#include <ctype.h>
```

20928
```
int isalpha(int c);
```

20929 **DESCRIPTION**

20930 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20931 conflict between the requirements described here and the ISO C standard is unintentional. This
20932 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

20933 The *isalpha*( ) function shall test whether *c* is a character of class **alpha** in the program's current
20934 locale; see the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale.

20935 In all cases *c* is an **int**, the value of which the application shall ensure is representable as an
20936 **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the
20937 behavior is undefined.

20938 **RETURN VALUE**

20939 The *isalpha*( ) function shall return non-zero if *c* is an alphabetic character; otherwise, it shall
20940 return 0.

20941 **ERRORS**

20942 No errors are defined.

20943 **EXAMPLES**

20944 None.

20945 **APPLICATION USAGE**

20946 To ensure applications portability, especially across natural languages, only this function and
20947 those listed in the SEE ALSO section should be used for character classification.

20948 **RATIONALE**

20949 None.

20950 **FUTURE DIRECTIONS**

20951 None.

20952 **SEE ALSO**

20953 *isalnum*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ),
20954 *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>**, **<stdio.h>**,
20955 the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale

20956 **CHANGE HISTORY**

20957 First released in Issue 1. Derived from Issue 1 of the SVID.

20958 **Issue 4**

20959 The text of the DESCRIPTION and RETURN VALUE sections is revised, although there are no
20960 functional differences between this issue and Issue 3. Operation in the C locale is no longer
20961 described explicitly on this reference page.

20962 **Issue 6**

20963 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

20964 **NAME**

20965 isascii — test for a 7-bit US-ASCII character

20966 **SYNOPSIS**

20967 XSI
```
#include <ctype.h>
```
20968
```
int isascii(int c);
```
20969

20970 **DESCRIPTION**

20971 The *isascii*( ) function tests whether *c* is a 7-bit US-ASCII character code.

20972 The *isascii*( ) function is defined on all integer values.

20973 **RETURN VALUE**

20974 The *isascii*( ) function shall return non-zero if *c* is a 7-bit US-ASCII character code between 0 and
20975 octal 0177 inclusive; otherwise, it shall return 0.

20976 **ERRORS**

20977 No errors are defined.

20978 **EXAMPLES**

20979 None.

20980 **APPLICATION USAGE**

20981 None.

20982 **RATIONALE**

20983 None.

20984 **FUTURE DIRECTIONS**

20985 None.

20986 **SEE ALSO**

20987 The Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>** |

20988 **CHANGE HISTORY**

20989 First released in Issue 1. Derived from Issue 1 of the SVID. |

20990 **NAME**

20991 isastream — test a file descriptor (**STREAMS**)

20992 **SYNOPSIS**

20993 XSR
```
#include <stropts.h>
```

20994
```
int isastream(int fildes);
```

20995

20996 **DESCRIPTION**

20997 The *isastream*( ) function shall test whether *fildes*, an open file descriptor, is associated with a
20998 STREAMS-based file.

20999 **RETURN VALUE**

21000 Upon successful completion, *isastream*( ) shall return 1 if *fildes* refers to a STREAMS-based file
21001 and 0 if not. Otherwise, *isastream*( ) shall return –1 and set *errno* to indicate the error.

21002 **ERRORS**

21003 The *isastream*( ) function shall fail if:

21004 [EBADF] The *fildes* argument is not a valid open file descriptor. |

21005 **EXAMPLES**

21006 None.

21007 **APPLICATION USAGE**

21008 None.

21009 **RATIONALE**

21010 None.

21011 **FUTURE DIRECTIONS**

21012 None.

21013 **SEE ALSO**

21014 The Base Definitions volume of IEEE Std. 1003.1-200x, **<stropts.h>** |

21015 **CHANGE HISTORY**

21016 First released in Issue 4, Version 2.

21017 **Issue 5**

21018 Moved from X/OPEN UNIX extension to BASE.

21019 **NAME**

21020 isatty — test for a terminal device

21021 **SYNOPSIS**

21022 `#include <unistd.h>`

21023 `int isatty(int fildes);`

21024 **DESCRIPTION**

21025 The *isatty*( ) function shall test whether *fildes*, an open file descriptor, is associated with a
21026 terminal device.

21027 **RETURN VALUE**

21028 The *isatty*( ) function shall return 1 if *fildes* is associated with a terminal; otherwise, it shall return
21029 0 and may set *errno* to indicate the error.

21030 **ERRORS**

21031 The *isatty*( ) function may fail if:

21032 [EBADF] The *fildes* argument is not a valid open file descriptor.

21033 [ENOTTY] The *fildes* argument is not associated with a terminal.

21034 **EXAMPLES**

21035 None.

21036 **APPLICATION USAGE**

21037 The *isatty*( ) function does not necessarily indicate that a human being is available for interaction
21038 via *fildes*. It is quite possible that non-terminal devices are connected to the communications
21039 line.

21040 **RATIONALE**

21041 None.

21042 **FUTURE DIRECTIONS**

21043 None.

21044 **SEE ALSO**

21045 The Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>**

21046 **CHANGE HISTORY**

21047 First released in Issue 1. Derived from Issue 1 of the SVID.

21048 **Issue 4**

21049 The **<unistd.h>** header is added to the SYNOPSIS section.

21050 In the RETURN VALUE section, the sentence indicating that this function may set *errno* is
21051 marked as an extension.

21052 The errors [EBADF] and [ENOTTY] are marked as extensions.

21053 **Issue 6**

21054 The following new requirements on POSIX implementations derive from alignment with the
21055 Single UNIX Specification:

21056 • The optional setting of *errno* to indicate an error is added.

21057 • The [EBADF] and [ENOTTY] optional error conditions are added.

21058 **NAME**

21059 isblank — test for a blank character

21060 **SYNOPSIS**

21061
```
#include <ctype.h>
```

21062
```
int isblank(int c);
```

21063 **DESCRIPTION**

21064 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21065 conflict between the requirements described here and the ISO C standard is unintentional. This
21066 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21067 The *isblank*( ) function shall test whether *c* is a character of class **blank** in the program's current
21068 locale; see the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale. In all cases *c*
21069 is a type **int**, the value of which the application shall ensure is a character representable as an
21070 **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the
21071 behavior is undefined.

21072 **RETURN VALUE**

21073 The *isblank*( ) function shall return non-zero if *c* is a <blank> character; otherwise, it shall return
21074 0.

21075 **ERRORS**

21076 No errors are defined.

21077 **EXAMPLES**

21078 None.

21079 **APPLICATION USAGE**

21080 To ensure applications portability, especially across natural languages, only this function and
21081 those listed in the SEE ALSO section should be used for character classification.

21082 **RATIONALE**

21083 None.

21084 **FUTURE DIRECTIONS**

21085 None.

21086 **SEE ALSO**

21087 *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ),
21088 *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>**

21089 **CHANGE HISTORY**

21090 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

21091 **NAME**

21092 iscntrl — test for a control character

21093 **SYNOPSIS**

21094
```
#include <ctype.h>
```

21095
```
int iscntrl(int c);
```

21096 **DESCRIPTION**

21097 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21098 conflict between the requirements described here and the ISO C standard is unintentional. This
21099 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21100 The *iscntrl*( ) function shall test whether *c* is a character of class **cntrl** in the program's current
21101 locale; see the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale.

21102 In all cases *c* is a type **int**, the value of which the application shall ensure is a character
21103 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21104 any other value, the behavior is undefined.

21105 **RETURN VALUE**

21106 The *iscntrl*( ) function shall return non-zero if *c* is a control character; otherwise, it shall return 0.

21107 **ERRORS**

21108 No errors are defined.

21109 **EXAMPLES**

21110 None.

21111 **APPLICATION USAGE**

21112 To ensure applications portability, especially across natural languages, only this function and
21113 those listed in the SEE ALSO section should be used for character classification.

21114 **RATIONALE**

21115 None.

21116 **FUTURE DIRECTIONS**

21117 None.

21118 **SEE ALSO**

21119 *isalnum*( ), *isalpha*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ),
21120 *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>**, the Base
21121 Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale

21122 **CHANGE HISTORY**

21123 First released in Issue 1. Derived from Issue 1 of the SVID.

21124 **Issue 4**

21125 The text of the DESCRIPTION and RETURN VALUE sections is revised, although there are no
21126 functional differences between this issue and Issue 3. Operation in the C locale is no longer
21127 described explicitly on this reference page.

21128 **Issue 6**

21129 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

NAME

isdigit — test for a decimal digit

SYNOPSIS

```
#include <ctype.h>

int isdigit(int c);
```

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 *isdigit*( ) function shall test whether *c* is a character of class **digit** in the program's current locale; see the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale.

In all cases *c* is an **int**, the value of which the application shall ensure is a character representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

RETURN VALUE

The *isdigit*( ) function shall return non-zero if *c* is a decimal digit; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum( ), *isalpha*( ), *iscntrl*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *isxdigit*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>**

CHANGE HISTORY

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

Issue 4

The text of the DESCRIPTION is revised, although there are no functional differences between this issue and Issue 3.

Issue 6

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

21167 **NAME**

21168 isfdtype — determine whether a file descriptor refers to a socket

21169 **SYNOPSIS**

21170 `#include <sys/stat.h>`

21171 `int isfdtype(int fildes, int fdtype);`

21172 **DESCRIPTION**

21173 The *isfdtype*( ) function shall determine whether the descriptor *fildes* has the properties identified
21174 by the value of *fdtype*, returning 1 if so, and 0 if not.

21175 If *fdtype* has the value S_IFSOCK:

21176 • The *isfdtype*( ) function shall return 1 if the descriptor refers to a socket.

21177 • It is implementation-defined whether *isfdtype*( ) shall return 1 if the descriptor refers to a |
21178 pipe.

21179 • The function shall return 0 for descriptors that refer neither to a socket nor to a pipe.

21180 **RETURN VALUE**

21181 Upon successful completion, the *isfdtype*( ) function shall return a value of 1 or 0 indicating
21182 whether the descriptor is of the indicated type. Otherwise, it shall return a value of –1 and set
21183 *errno* to indicate the error.

21184 **ERRORS**

21185 If any of the following conditions occur, the *isfdtype*( ) function shall return –1 and set *errno* to
21186 the corresponding value:

21187 [EBADF] The *fildes* argument is not a valid file descriptor.

21188 **EXAMPLES**

21189 None.

21190 **APPLICATION USAGE**

21191 None.

21192 **RATIONALE**

21193 None.

21194 **FUTURE DIRECTIONS**

21195 None.

21196 **SEE ALSO**

21197 *isatty*( ), *socket*( ), *stat*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/stat.h>** |

21198 **CHANGE HISTORY** |

21199 First released in Issue 6. Derived from IEEE Std. 1003.1g-2000. |

21200 **NAME**

21201 isfinite — test for finite value

21202 **SYNOPSIS**

21203 `#include <math.h>`

21204 `int isfinite(real-floating` *x*`);`

21205 **DESCRIPTION**

21206 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21207 conflict between the requirements described here and the ISO C standard is unintentional. This
21208 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21209 The *isfinite*( ) macro shall determine whether its argument has a finite value (zero, subnormal, or
21210 normal, and not infinite or NaN). First, an argument represented in a format wider than its
21211 semantic type is converted to its semantic type. Then determination is based on the type of the
21212 argument.

21213 **RETURN VALUE**

21214 The *isfinite*( ) macro shall return a non-zero value if and only if its argument has a finite value.

21215 **ERRORS**

21216 No errors are defined.

21217 **EXAMPLES**

21218 None.

21219 **APPLICATION USAGE**

21220 None.

21221 **RATIONALE**

21222 None.

21223 **FUTURE DIRECTIONS**

21224 None.

21225 **SEE ALSO**

21226 *fpclassify*( ), *isinf*( ), *isnan*( ), *isnormal*( ), *signbit*( ), the Base Definitions volume of
21227 IEEE Std. 1003.1-200x **<math.h>**

21228 **CHANGE HISTORY**

21229 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

21230 **NAME**

21231 isgraph — test for a visible character

21232 **SYNOPSIS**

21233 `#include <ctype.h>`

21234 `int isgraph(int c);`

21235 **DESCRIPTION**

21236 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21237 conflict between the requirements described here and the ISO C standard is unintentional. This
21238 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21239 The *isgraph*( ) function shall test whether *c* is a character of class **graph** in the program's current
21240 locale; see the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale.

21241 In all cases *c* is an **int**, the value of which the application shall ensure is a character representable
21242 as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value,
21243 the behavior is undefined.

21244 **RETURN VALUE**

21245 The *isgraph*( ) function shall return non-zero if *c* is a character with a visible representation;
21246 otherwise, it shall return 0.

21247 **ERRORS**

21248 No errors are defined.

21249 **EXAMPLES**

21250 None.

21251 **APPLICATION USAGE**

21252 To ensure applications portability, especially across natural languages, only this function and
21253 those listed in the SEE ALSO section should be used for character classification.

21254 **RATIONALE**

21255 None.

21256 **FUTURE DIRECTIONS**

21257 None.

21258 **SEE ALSO**

21259 *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *isxdigit*( ),
21260 *setlocale*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>**, the Base Definitions
21261 volume of IEEE Std. 1003.1-200x, Chapter 7, Locale

21262 **CHANGE HISTORY**

21263 First released in Issue 1. Derived from Issue 1 of the SVID.

21264 **Issue 4**

21265 The text of the DESCRIPTION and RETURN VALUE sections is revised, although there are no
21266 functional differences between this issue and Issue 3. Operation in the C locale is no longer
21267 described explicitly on this reference page.

21268 **Issue 6**

21269 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

21270 **NAME**

21271 isgreater — test if x greater than y

21272 **SYNOPSIS**

21273 `#include <math.h>`

21274 `int isgreater(real-floating` *x*`, real-floating` *y*`);`

21275 **DESCRIPTION**

21276 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21277 conflict between the requirements described here and the ISO C standard is unintentional. This
21278 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21279 The *isgreater*( ) macro shall determine whether its first argument is greater than its second
21280 argument. The value of *isgreater*(*x*, *y*) shall be equal to (*x*) > (*y*); however, unlike (*x*) > (*y*),
21281 *isgreater*(*x*, *y*) shall not raise the invalid floating-point exception when *x* and *y* are unordered.

21282 **RETURN VALUE**

21283 Upon successful completion, the *isgreater*( ) macro shall return the value of (*x*) > (*y*).

21284 If *x* or *y* is NaN, 0 shall be returned.

21285 **ERRORS**

21286 No errors are defined.

21287 **EXAMPLES**

21288 None.

21289 **APPLICATION USAGE**

21290 The relational and equality operators support the usual mathematical relationships between
21291 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21292 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21293 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21294 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21295 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21296 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21297 indicates that the argument shall be an expression of **real-floating** type.

21298 **RATIONALE**

21299 None.

21300 **FUTURE DIRECTIONS**

21301 None.

21302 **SEE ALSO**

21303 *isgreaterequal*( ), *isless*( ), *islessequal*( ), *islessgreater*( ), *isunordered*( ), the Base Definitions volume of
21304 IEEE Std. 1003.1-200x **<math.h>**

21305 **CHANGE HISTORY**

21306 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

21307 **NAME**

21308 isgreaterequal — test if x greater than or equal to y

21309 **SYNOPSIS**

21310 `#include <math.h>`

21311 `int isgreaterequal(real-floating` *x*`, real-floating` *y*`);`

21312 **DESCRIPTION**

21313 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21314 conflict between the requirements described here and the ISO C standard is unintentional. This
21315 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21316 The *isgreaterequal*( ) macro shall determine whether its first argument is greater than or equal to
21317 its second argument. The value of *isgreaterequal*(*x*, *y*) shall be equal to (*x*) >= (*y*); however, unlike
21318 (*x*) >= (*y*), *isgreaterequal*(*x*, *y*) shall not raise the invalid floating-point exception when *x* and *y* are
21319 unordered.

21320 **RETURN VALUE**

21321 Upon successful completion, the *isgreaterequal*( ) macro shall return the value of (*x*) >= (*y*).

21322 If *x* or *y* is NaN, 0 shall be returned.

21323 **ERRORS**

21324 No errors are defined.

21325 **EXAMPLES**

21326 None.

21327 **APPLICATION USAGE**

21328 The relational and equality operators support the usual mathematical relationships between
21329 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21330 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21331 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21332 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21333 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21334 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21335 indicates that the argument shall be an expression of **real-floating** type.

21336 **RATIONALE**

21337 None.

21338 **FUTURE DIRECTIONS**

21339 None.

21340 **SEE ALSO**

21341 *isgreater*( ), *isless*( ), *islessequal*( ), *islessgreater*( ), *isunordered*( ), the Base Definitions volume of
21342 IEEE Std. 1003.1-200x **<math.h>**

21343 **CHANGE HISTORY**

21344 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

21345 **NAME**

21346 isinf — test for infinity

21347 **SYNOPSIS**

21348 `#include <math.h>`

21349 `int isinf(real-floating` *x*`);`

21350 **DESCRIPTION**

21351 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21352 conflict between the requirements described here and the ISO C standard is unintentional. This
21353 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21354 The *isinf*( ) macro shall determine whether its argument value is an infinity (positive or
21355 negative). First, an argument represented in a format wider than its semantic type is converted
21356 to its semantic type. Then determination is based on the type of the argument.

21357 **RETURN VALUE**

21358 The *isinf*( ) macro shall return a non-zero value if and only if its argument has an infinite value.

21359 **ERRORS**

21360 No errors are defined.

21361 **EXAMPLES**

21362 None.

21363 **APPLICATION USAGE**

21364 None.

21365 **RATIONALE**

21366 None.

21367 **FUTURE DIRECTIONS**

21368 None.

21369 **SEE ALSO**

21370 *fpclassify*( ), *isfinite*( ), *isnan*( ), *isnormal*( ), *signbit*( ), the Base Definitions volume of
21371 IEEE Std. 1003.1-200x **<math.h>**

21372 **CHANGE HISTORY**

21373 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

21374 **NAME**

21375 isless — test if x is less than y

21376 **SYNOPSIS**

21377 `#include <math.h>`

21378 `int isless(real-floating x, real-floating y);`

21379 **DESCRIPTION**

21380 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21381 conflict between the requirements described here and the ISO C standard is unintentional. This
21382 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21383 The *isless*( ) macro shall determine whether its first argument is less than its second argument.
21384 The value of *isless*(*x*, *y*) shall be equal to (*x*) < (*y*); however, unlike (*x*) < (*y*), *isless*(*x*, *y*) shall not
21385 raise the invalid floating-point exception when *x* and *y* are unordered.

21386 **RETURN VALUE**

21387 Upon successful completion, the *isless*( ) macro shall return the value of (*x*) < (*y*).

21388 If *x* or *y* is NaN, 0 shall be returned.

21389 **ERRORS**

21390 No errors are defined.

21391 **EXAMPLES**

21392 None.

21393 **APPLICATION USAGE**

21394 The relational and equality operators support the usual mathematical relationships between
21395 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21396 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21397 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21398 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21399 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21400 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21401 indicates that the argument shall be an expression of **real-floating** type.

21402 **RATIONALE**

21403 None.

21404 **FUTURE DIRECTIONS**

21405 None.

21406 **SEE ALSO**

21407 *isgreater*( ), *isgreaterequal*( ), *islessequal*( ), *islessgreater*( ), *isunordered*( ), the Base Definitions volume
21408 of IEEE Std. 1003.1-200x, **<math.h>**

21409 **CHANGE HISTORY**

21410 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

21411 **NAME**

21412 islessequal — test if x is less than or equal to y

21413 **SYNOPSIS**

21414 `#include <math.h>`

21415 `int islessequal(real-floating x, real-floating y);`

21416 **DESCRIPTION**

21417 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21418 conflict between the requirements described here and the ISO C standard is unintentional. This
21419 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21420 The *islessequal*( ) macro shall determine whether its first argument is less than or equal to its
21421 second argument. The value of *islessequal*(*x*, *y*) shall be equal to (*x*) <= (*y*); however, unlike
21422 (*x*) <= (*y*), *islessequal*(*x*, *y*) shall not raise the invalid floating-point exception when *x* and *y* are
21423 unordered.

21424 **RETURN VALUE**

21425 Upon successful completion, the *islessequal*( ) macro shall return the value of (*x*) <= (*y*).

21426 If *x* or *y* is NaN, 0 shall be returned.

21427 **ERRORS**

21428 No errors are defined.

21429 **EXAMPLES**

21430 None.

21431 **APPLICATION USAGE**

21432 The relational and equality operators support the usual mathematical relationships between
21433 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21434 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21435 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21436 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21437 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21438 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21439 indicates that the argument shall be an expression of **real-floating** type.

21440 **RATIONALE**

21441 None.

21442 **FUTURE DIRECTIONS**

21443 None.

21444 **SEE ALSO**

21445 *isgreater*( ), *isgreaterequal*( ), *isless*( ), *islessgreater*( ), *isunordered*( ), the Base Definitions volume of
21446 IEEE Std. 1003.1-200x **<math.h>**

21447 **CHANGE HISTORY**

21448 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

21449 **NAME**

21450 islessgreater — test if x is less than or greater than y

21451 **SYNOPSIS**

21452 `#include <math.h>`

21453 `int islessgreater(real-floating x, real-floating y);`

21454 **DESCRIPTION**

21455 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21456 conflict between the requirements described here and the ISO C standard is unintentional. This
21457 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21458 The *islessgreater*( ) macro shall determine whether its first argument is less than or greater than
21459 its second argument. The *islessgreater*(*x*, *y*) macro is similar to (*x*) < (*y*) | | (*x*) > (*y*); however,
21460 *islessgreater*(*x*, *y*) shall not raise the invalid floating-point exception when *x* and *y* are unordered
21461 (nor shall it evaluate *x* and *y* twice).

21462 **RETURN VALUE**

21463 Upon successful completion, the *islessgreater*( ) macro shall return the value of
21464 (*x*) < (*y*) | | (*x*) > (*y*).

21465 If *x* or *y* is NaN, 0 shall be returned.

21466 **ERRORS**

21467 No errors are defined.

21468 **EXAMPLES**

21469 None.

21470 **APPLICATION USAGE**

21471 The relational and equality operators support the usual mathematical relationships between
21472 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21473 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21474 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21475 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21476 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21477 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21478 indicates that the argument shall be an expression of **real-floating** type.

21479 **RATIONALE**

21480 None.

21481 **FUTURE DIRECTIONS**

21482 None.

21483 **SEE ALSO**

21484 *isgreater*( ), *isgreaterequal*( ), *isless*( ), *islessequal*( ), *isunordered*( ), the Base Definitions volume of
21485 IEEE Std. 1003.1-200x **<math.h>**

21486 **CHANGE HISTORY**

21487 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

islower — test for a lowercase letter

SYNOPSIS

```
#include <ctype.h>

int islower(int c);
```

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 *islower*( ) function shall test whether *c* is a character of class **lower** in the program's current locale; see the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale.

In all cases *c* is an **int**, the value of which the application shall ensure is a character representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

RETURN VALUE

The *islower*( ) function shall return non-zero if *c* is a lowercase letter; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

Testing for a Lowercase Letter

The following example tests whether the value is a lowercase letter, based on the locale of the user, then uses it as part of a key value.

```
#include <ctype.h>
#include <stdlib.h>
#include <locale.h>
...
char *keystr;
int elementlen, len;
char c;
...
setlocale(LC_ALL, "");
...
len = 0;
while (len < elementlen) {
    c = (char) (rand() % 256);
...
    if (islower(c))
        keystr[len++] = c;
    }
...
```

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

21531 **RATIONALE**

21532 None.

21533 **FUTURE DIRECTIONS**

21534 None.

21535 **SEE ALSO**

21536 *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ),
21537 *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>**, the Base |
21538 Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale |

21539 **CHANGE HISTORY**

21540 First released in Issue 1. Derived from Issue 1 of the SVID. |

21541 **Issue 4**

21542 The text of the DESCRIPTION and RETURN VALUE sections is revised, although there are no
21543 functional differences between this issue and Issue 3. Operation in the C locale is no longer
21544 described explicitly on this reference page.

21545 **Issue 6**

21546 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

21547 **NAME**

21548 isnan — test for a NaN

21549 **SYNOPSIS**

21550 `#include <math.h>`

21551 `int isnan(real-floating x);`

21552 **DESCRIPTION**

21553 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21554 conflict between the requirements described here and the ISO C standard is unintentional. This
21555 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21556 The *isnan*( ) macro shall determine whether its argument value is a NaN. First, an argument
21557 represented in a format wider than its semantic type is converted to its semantic type. Then
21558 determination is based on the type of the argument.

21559 **RETURN VALUE**

21560 The *isnan*( ) macro shall return a non-zero value if and only if its argument has a NaN value.

21561 **ERRORS**

21562 No errors are defined.

21563 **EXAMPLES**

21564 None.

21565 **APPLICATION USAGE**

21566 None.

21567 **RATIONALE**

21568 None.

21569 **FUTURE DIRECTIONS**

21570 None.

21571 **SEE ALSO**

21572 *fpclassify*( ), *isfinite*( ), *isinf*( ), *isnormal*( ), *signbit*( ), the Base Definitions volume of
21573 IEEE Std. 1003.1-200x, **<math.h>**

21574 **CHANGE HISTORY**

21575 First released in Issue 3.

21576 **Issue 4**

21577 The words ‘‘not supporting NaN’’ are added to the APPLICATION USAGE section.

21578 **Issue 5**

21579 The DESCRIPTION is updated to indicate the return value when NaN is not supported. This
21580 text was previously published in the APPLICATION USAGE section.

21581 **Issue 6**

21582 Entry re-written for alignment with the ISO / IEC 9899: 1999 standard.

21583 **NAME**

21584 isnormal — test for a normal value

21585 **SYNOPSIS**

21586 `#include <math.h>`

21587 `int isnormal(real-floating x);`

21588 **DESCRIPTION**

21589 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21590 conflict between the requirements described here and the ISO C standard is unintentional. This
21591 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21592 The *isnormal*() macro shall determine whether its argument value is normal (neither zero,
21593 subnormal, infinite, nor NaN). First, an argument represented in a format wider than its
21594 semantic type is converted to its semantic type. Then determination is based on the type of the
21595 argument.

21596 **RETURN VALUE**

21597 The *isnormal*() macro shall return a non-zero value if and only if its argument has a normal
21598 value.

21599 **ERRORS**

21600 No errors are defined.

21601 **EXAMPLES**

21602 None.

21603 **APPLICATION USAGE**

21604 None.

21605 **RATIONALE**

21606 None.

21607 **FUTURE DIRECTIONS**

21608 None.

21609 **SEE ALSO**

21610 *fpclassify*(), *isfinite*(), *isinf*(), *isnan*(), *signbit*(), the Base Definitions volume of
21611 IEEE Std. 1003.1-200x, **<math.h>**

21612 **CHANGE HISTORY**

21613 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

isprint — test for a printing character

SYNOPSIS

```
#include <ctype.h>

int isprint(int c);
```

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 *isprint*( ) function shall test whether *c* is a character of class **print** in the program's current locale; see the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale.

In all cases *c* is an **int**, the value of which the application shall ensure is a character representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

RETURN VALUE

The *isprint*( ) function shall return non-zero if *c* is a printing character; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

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

Issue 4

The text of the DESCRIPTION and RETURN VALUE sections is revised, although there are no functional differences between this issue and Issue 3. Operation in the C locale is no longer described explicitly on this reference page.

Issue 6

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

21653 **NAME**

21654 ispunct — test for a punctuation character

21655 **SYNOPSIS**

21656
```
#include <ctype.h>
```

21657
```
int ispunct(int c);
```

21658 **DESCRIPTION**

21659 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21660 conflict between the requirements described here and the ISO C standard is unintentional. This
21661 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21662 The *ispunct*( ) function shall test whether *c* is a character of class **punct** in the program's current
21663 locale; see the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale.

21664 In all cases *c* is an **int**, the value of which the application shall ensure is a character representable
21665 as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value,
21666 the behavior is undefined.

21667 **RETURN VALUE**

21668 The *ispunct*( ) function shall return non-zero if *c* is a punctuation character; otherwise, it shall
21669 return 0.

21670 **ERRORS**

21671 No errors are defined.

21672 **EXAMPLES**

21673 None.

21674 **APPLICATION USAGE**

21675 To ensure applications portability, especially across natural languages, only this function and
21676 those listed in the SEE ALSO section should be used for character classification.

21677 **RATIONALE**

21678 None.

21679 **FUTURE DIRECTIONS**

21680 None.

21681 **SEE ALSO**

21682 *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *isspace*( ), *isupper*( ), *isxdigit*( ),
21683 *setlocale*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>**, the Base Definitions
21684 volume of IEEE Std. 1003.1-200x, Chapter 7, Locale

21685 **CHANGE HISTORY**

21686 First released in Issue 1. Derived from Issue 1 of the SVID.

21687 **Issue 4**

21688 The text of the DESCRIPTION and RETURN VALUE sections is revised, although there are no
21689 functional differences between this issue and Issue 3. Operation in the C locale is no longer
21690 described explicitly on this reference page.

21691 **Issue 6**

21692 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

21693 **NAME**

21694 isspace — test for a white-space character

21695 **SYNOPSIS**

21696
```
#include <ctype.h>
```

21697
```
int isspace(int c);
```

21698 **DESCRIPTION**

21699 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21700 conflict between the requirements described here and the ISO C standard is unintentional. This
21701 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21702 The *isspace*( ) function shall test whether *c* is a character of class **space** in the program's current
21703 locale; see the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale.

21704 In all cases *c* is an **int**, the value of which the application shall ensure is a character representable
21705 as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value,
21706 the behavior is undefined.

21707 **RETURN VALUE**

21708 The *isspace*( ) function shall return non-zero if *c* is a white-space character; otherwise, it shall
21709 return 0.

21710 **ERRORS**

21711 No errors are defined.

21712 **EXAMPLES**

21713 None.

21714 **APPLICATION USAGE**

21715 To ensure applications portability, especially across natural languages, only this function and
21716 those listed in the SEE ALSO section should be used for character classification.

21717 **RATIONALE**

21718 None.

21719 **FUTURE DIRECTIONS**

21720 None.

21721 **SEE ALSO**

21722 *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isupper*( ),
21723 *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>**, the Base
21724 Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale

21725 **CHANGE HISTORY**

21726 First released in Issue 1. Derived from Issue 1 of the SVID.

21727 **Issue 4**

21728 The text of the DESCRIPTION and RETURN VALUE sections is revised, although there are no
21729 functional differences between this issue and Issue 3. Operation in the C locale is no longer
21730 described explicitly on this reference page.

21731 **Issue 6**

21732 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

21733 **NAME**

21734 isunordered — test if arguments are unordered

21735 **SYNOPSIS**

21736 `#include <math.h>`

21737 `int isunordered(real-floating x, real-floating y);`

21738 **DESCRIPTION**

21739 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21740 conflict between the requirements described here and the ISO C standard is unintentional. This
21741 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21742 The *isunordered*( ) macro shall determine whether its arguments are unordered.

21743 **RETURN VALUE**

21744 Upon successful completion, the *isunordered*( ) macro shall return 1 if its arguments are
21745 unordered, and 0 otherwise.

21746 If *x* or *y* is NaN, 0 shall be returned.

21747 **ERRORS**

21748 No errors are defined.

21749 **EXAMPLES**

21750 None.

21751 **APPLICATION USAGE**

21752 The relational and equality operators support the usual mathematical relationships between
21753 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21754 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21755 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21756 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21757 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21758 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21759 indicates that the argument shall be an expression of **real-floating** type.

21760 **RATIONALE**

21761 None.

21762 **FUTURE DIRECTIONS**

21763 None.

21764 **SEE ALSO**

21765 *isgreater*( ), *isgreaterequal*( ), *isless*( ), *islessequal*( ), *islessgreater*( ), the Base Definitions volume of
21766 IEEE Std. 1003.1-200x, **<math.h>**

21767 **CHANGE HISTORY**

21768 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

isupper — test for an uppercase letter

SYNOPSIS

```
#include <ctype.h>

int isupper(int c);
```

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 *isupper*( ) function shall test whether *c* is a character of class **upper** in the program's current locale; see the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale.

In all cases *c* is an **int**, the value of which the application shall ensure is a character representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

RETURN VALUE

The *isupper*( ) function shall return non-zero if *c* is an uppercase letter; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

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

Issue 4

The text of the DESCRIPTION and RETURN VALUE sections is revised, although there are no functional differences between this issue and Issue 3. Operation in the C locale is no longer described explicitly on this reference page.

Issue 6

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

21808 **NAME**

21809 iswalnum — test for an alphanumeric wide-character code

21810 **SYNOPSIS**

21811 ```
#include <wctype.h>
```

21812 ```
int iswalnum(wint_t wc);
```

21813 **DESCRIPTION**

21814 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21815 conflict between the requirements described here and the ISO C standard is unintentional. This
21816 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21817 The *iswalnum*( ) function shall test whether *wc* is a wide-character code representing a character
21818 of class **alpha** or **digit** in the program's current locale; see the Base Definitions volume of
21819 IEEE Std. 1003.1-200x, Chapter 7, Locale.

21820 In all cases *wc* is a **wint_t**, the value of which the application shall ensure is a wide-character
21821 code corresponding to a valid character in the current locale, or equal to the value of the macro
21822 WEOF. If the argument has any other value, the behavior is undefined.

21823 **RETURN VALUE**

21824 The *iswalnum*( ) function shall return non-zero if *wc* is an alphanumeric wide-character code;
21825 otherwise, it shall return 0.

21826 **ERRORS**

21827 No errors are defined.

21828 **EXAMPLES**

21829 None.

21830 **APPLICATION USAGE**

21831 To ensure applications portability, especially across natural languages, only this function and
21832 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21833 **RATIONALE**

21834 None.

21835 **FUTURE DIRECTIONS**

21836 None.

21837 **SEE ALSO**

21838 *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ),
21839 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
21840 IEEE Std. 1003.1-200x, **<wctype.h>**, **<wchar.h>**, **<stdio.h>**, the Base Definitions volume of
21841 IEEE Std. 1003.1-200x, Chapter 7, Locale

21842 **CHANGE HISTORY**

21843 First released as a World-wide Portability Interface in Issue 4.

21844 **Issue 5**

21845 The following change has been made in this issue for alignment with
21846 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

21847 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21848 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

21849 **Issue 6**

21850 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

21851 **NAME**

21852 iswalpha — test for an alphabetic wide-character code

21853 **SYNOPSIS**

21854 `#include <wctype.h>`

21855 `int iswalpha(wint_t wc);`

21856 **DESCRIPTION**

21857 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21858 conflict between the requirements described here and the ISO C standard is unintentional. This
21859 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21860 The *iswalpha*( ) function shall test whether *wc* is a wide-character code representing a character of
21861 class **alpha** in the program's current locale; see the Base Definitions volume of
21862 IEEE Std. 1003.1-200x, Chapter 7, Locale.

21863 In all cases *wc* is a **wint_t**, the value of which the application shall ensure is a wide-character
21864 code corresponding to a valid character in the current locale, or equal to the value of the macro
21865 WEOF. If the argument has any other value, the behavior is undefined.

21866 **RETURN VALUE**

21867 The *iswalpha*( ) function shall return non-zero if *wc* is an alphabetic wide-character code;
21868 otherwise, it shall return 0.

21869 **ERRORS**

21870 No errors are defined.

21871 **EXAMPLES**

21872 None.

21873 **APPLICATION USAGE**

21874 To ensure applications portability, especially across natural languages, only this function and
21875 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21876 **RATIONALE**

21877 None.

21878 **FUTURE DIRECTIONS**

21879 None.

21880 **SEE ALSO**

21881 *iswalnum*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ),
21882 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
21883 IEEE Std. 1003.1-200x, **<wctype.h>**, **<wchar.h>**, **<stdio.h>**, the Base Definitions volume of
21884 IEEE Std. 1003.1-200x, Chapter 7, Locale

21885 **CHANGE HISTORY**

21886 First released in Issue 4.

21887 **Issue 5**

21888 The following change has been made in this issue for alignment with
21889 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

21890 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21891 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

21894 **NAME**

21895 iswblank — test for a blank wide-character code

21896 **SYNOPSIS**

21897
```
#include <wctype.h>
```

21898
```
int iswblank(wint_t wc);
```

21899 **DESCRIPTION**

21900 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21901 conflict between the requirements described here and the ISO C standard is unintentional. This
21902 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21903 The *iswblank*( ) function shall test whether *wc* is a wide-character code representing a character of
21904 class **blank** in the program's current locale; see the Base Definitions volume of
21905 IEEE Std. 1003.1-200x, Chapter 7, Locale. In all cases, *wc* is a **wint_t**, the value of which the
21906 application shall ensure is a wide-character code corresponding to a valid character in the
21907 current locale, or equal to the value of the macro WEOF. If the argument has any other value, the
21908 behavior is undefined.

21909 **RETURN VALUE**

21910 The *iswblank*( ) function shall return non-zero if *wc* is a <blank> wide-character code; otherwise,
21911 it shall return 0.

21912 **ERRORS**

21913 No errors are defined.

21914 **EXAMPLES**

21915 None.

21916 **APPLICATION USAGE**

21917 To ensure applications portability, especially across natural languages, only this function and
21918 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21919 **RATIONALE**

21920 None.

21921 **FUTURE DIRECTIONS**

21922 None.

21923 **SEE ALSO**

21924 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ),
21925 *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
21926 IEEE Std. 1003.1-200x, **<wchar.h>**, **<wctype.h>**, **<stdio.h>**

21927 **CHANGE HISTORY**

21928 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

21929 **NAME**

21930 iswcntrl — test for a control wide-character code

21931 **SYNOPSIS**

21932 `#include <wctype.h>`

21933 `int iswcntrl(wint_t wc);`

21934 **DESCRIPTION**

21935 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21936 conflict between the requirements described here and the ISO C standard is unintentional. This
21937 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

21938 The *iswcntrl*( ) function shall test whether *wc* is a wide-character code representing a character of
21939 class **control** in the program's current locale; see the Base Definitions volume of
21940 IEEE Std. 1003.1-200x, Chapter 7, Locale.

21941 In all cases *wc* is a **wint_t**, the value of which the application shall ensure is a wide-character
21942 code corresponding to a valid character in the current locale, or equal to the value of the macro
21943 WEOF. If the argument has any other value, the behavior is undefined.

21944 **RETURN VALUE**

21945 The *iswcntrl*( ) function shall return non-zero if *wc* is a control wide-character code; otherwise, it
21946 shall return 0.

21947 **ERRORS**

21948 No errors are defined.

21949 **EXAMPLES**

21950 None.

21951 **APPLICATION USAGE**

21952 To ensure applications portability, especially across natural languages, only this function and
21953 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21954 **RATIONALE**

21955 None.

21956 **FUTURE DIRECTIONS**

21957 None.

21958 **SEE ALSO**

21959 *iswalnum*( ), *iswalpha*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ),
21960 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
21961 IEEE Std. 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of
21962 IEEE Std. 1003.1-200x, Chapter 7, Locale

21963 **CHANGE HISTORY**

21964 First released in Issue 4.

21965 **Issue 5**

21966 The following change has been made in this issue for alignment with
21967 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

21968 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21969 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

21970 **Issue 6**

21971 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

iswctype — test character for a specified class

SYNOPSIS

```
#include <wctype.h>

int iswctype(wint_t wc, wctype_t charclass);
```

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 *iswctype*( ) function shall determine whether the wide-character code *wc* has the character class *charclass*, returning true or false. The *iswctype*( ) function is defined on WEOF and wide-character codes corresponding to the valid character encodings in the current locale. If the *wc* argument is not in the domain of the function, the result is undefined. If the value of *charclass* is invalid (that is, not obtained by a call to *wctype*( ) or *charclass* is invalidated by a subsequent call to *setlocale*( ) that has affected category *LC_CTYPE*) the result is unspecified.

RETURN VALUE

The *iswctype*( ) function shall return non-zero (true) if and only if *wc* has the property described by *charclass*. CX If *charclass* is 0, *iswctype*( ) shall return 0.

ERRORS

No errors are defined.

EXAMPLES

Testing for a Valid Character

```
#include <wctype.h>
...
int yes_or_no;
wint_t wc;
wctype_t valid_class;
...
if ((valid_class=wctype("vowel")) == (wctype_t)0)
    /* Invalid character class. */
yes_or_no=iswctype(wc,valid_class);
```

APPLICATION USAGE

The twelve strings "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper", and "xdigit" are reserved for the standard character classes. In the table below, the functions in the left column are equivalent to the functions in the right column.

```
iswalnum(wc)    iswctype(wc, wctype("alnum"))
iswalpha(wc)    iswctype(wc, wctype("alpha"))
iswblank(wc)    iswctype(wc, wctype("blank"))
iswcntrl(wc)    iswctype(wc, wctype("cntrl"))
iswdigit(wc)    iswctype(wc, wctype("digit"))
iswgraph(wc)    iswctype(wc, wctype("graph"))
iswlower(wc)    iswctype(wc, wctype("lower"))
iswprint(wc)    iswctype(wc, wctype("print"))
iswpunct(wc)    iswctype(wc, wctype("punct"))
iswspace(wc)    iswctype(wc, wctype("space"))
```

```
iswupper(wc)    iswctype(wc, wctype("upper"))
iswxdigit(wc)   iswctype(wc, wctype("xdigit"))
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswalpha*( ), *iswcntrl*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), *wctype*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wctype.h>**, **<wchar.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>**.

22035 **NAME**

22036 iswdigit — test for a decimal digit wide-character code

22037 **SYNOPSIS**

22038
```
#include <wctype.h>
```

22039
```
int iswdigit(wint_t wc);
```

22040 **DESCRIPTION**

22041 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22042 conflict between the requirements described here and the ISO C standard is unintentional. This
22043 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

22044 The *iswdigit*( ) function shall test whether *wc* is a wide-character code representing a character of
22045 class **digit** in the program's current locale; see the Base Definitions volume of
22046 IEEE Std. 1003.1-200x, Chapter 7, Locale.

22047 In all cases *wc* is a **wint_t**, the value of which the application shall ensure is a wide-character
22048 code corresponding to a valid character in the current locale, or equal to the value of the macro
22049 WEOF. If the argument has any other value, the behavior is undefined.

22050 **RETURN VALUE**

22051 The *iswdigit*( ) function shall return non-zero if *wc* is a decimal digit wide-character code;
22052 otherwise, it shall return 0.

22053 **ERRORS**

22054 No errors are defined.

22055 **EXAMPLES**

22056 None.

22057 **APPLICATION USAGE**

22058 To ensure applications portability, especially across natural languages, only this function and
22059 those listed in the SEE ALSO section should be used for classification of wide-character codes.

22060 **RATIONALE**

22061 None.

22062 **FUTURE DIRECTIONS**

22063 None.

22064 **SEE ALSO**

22065 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ),
22066 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
22067 IEEE Std. 1003.1-200x, **<wctype.h>**, **<wchar.h>**

22068 **CHANGE HISTORY**

22069 First released in Issue 4.

22070 **Issue 5**

22071 The following change has been made in this issue for alignment with
22072 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

22073 • The SYNOPSIS has been changed to indicate that this function and associated data types are
22074 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

22075 **Issue 6**

22076 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

22077 **NAME**

22078 iswgraph — test for a visible wide-character code

22079 **SYNOPSIS**

22080 `#include <wctype.h>`

22081 `int iswgraph(wint_t wc);`

22082 **DESCRIPTION**

22083 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22084 conflict between the requirements described here and the ISO C standard is unintentional. This
22085 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

22086 The *iswgraph*( ) function shall test whether *wc* is a wide-character code representing a character
22087 of class **graph** in the program's current locale; see the Base Definitions volume of
22088 IEEE Std. 1003.1-200x, Chapter 7, Locale.

22089 In all cases *wc* is a **wint_t**, the value of which the application shall ensure is a wide-character
22090 code corresponding to a valid character in the current locale, or equal to the value of the macro
22091 WEOF. If the argument has any other value, the behavior is undefined.

22092 **RETURN VALUE**

22093 The *iswgraph*( ) function shall return non-zero if *wc* is a wide-character code with a visible
22094 representation; otherwise, it shall return 0.

22095 **ERRORS**

22096 No errors are defined.

22097 **EXAMPLES**

22098 None.

22099 **APPLICATION USAGE**

22100 To ensure applications portability, especially across natural languages, only this function and
22101 those listed in the SEE ALSO section should be used for classification of wide-character codes.

22102 **RATIONALE**

22103 None.

22104 **FUTURE DIRECTIONS**

22105 None.

22106 **SEE ALSO**

22107 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ),
22108 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
22109 IEEE Std. 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of
22110 IEEE Std. 1003.1-200x, Chapter 7, Locale

22111 **CHANGE HISTORY**

22112 First released in Issue 4.

22113 **Issue 5**

22114 The following change has been made in this issue for alignment with
22115 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

22116 • The SYNOPSIS has been changed to indicate that this function and associated data types are
22117 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

22120 **NAME**

22121 iswlower — test for a lowercase letter wide-character code

22122 **SYNOPSIS**

22123 `#include <wctype.h>`

22124 `int iswlower(wint_t wc);`

22125 **DESCRIPTION**

22126 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22127 conflict between the requirements described here and the ISO C standard is unintentional. This
22128 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

22129 The *iswlower*( ) function shall test whether *wc* is a wide-character code representing a character
22130 of class **lower** in the program's current locale; see the Base Definitions volume of
22131 IEEE Std. 1003.1-200x, Chapter 7, Locale.

22132 In all cases *wc* is a **wint_t**, the value of which the application shall ensure is a wide-character
22133 code corresponding to a valid character in the current locale, or equal to the value of the macro
22134 WEOF. If the argument has any other value, the behavior is undefined.

22135 **RETURN VALUE**

22136 The *iswlower*( ) function shall return non-zero if *wc* is a lowercase letter wide-character code;
22137 otherwise, it shall return 0.

22138 **ERRORS**

22139 No errors are defined.

22140 **EXAMPLES**

22141 None.

22142 **APPLICATION USAGE**

22143 To ensure applications portability, especially across natural languages, only this function and
22144 those listed in the SEE ALSO section should be used for classification of wide-character codes.

22145 **RATIONALE**

22146 None.

22147 **FUTURE DIRECTIONS**

22148 None.

22149 **SEE ALSO**

22150 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswprint*( ), *iswpunct*( ),
22151 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
22152 IEEE Std. 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of
22153 IEEE Std. 1003.1-200x, Chapter 7, Locale

22154 **CHANGE HISTORY**

22155 First released in Issue 4.

22156 **Issue 5**

22157 The following change has been made in this issue for alignment with
22158 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

22159 • The SYNOPSIS has been changed to indicate that this function and associated data types are
22160 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

22163 **NAME**

22164 iswprint — test for a printing wide-character code

22165 **SYNOPSIS**

22166 `#include <wctype.h>`

22167 `int iswprint(wint_t wc);`

22168 **DESCRIPTION**

22169 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22170 conflict between the requirements described here and the ISO C standard is unintentional. This
22171 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

22172 The *iswprint*( ) function shall test whether *wc* is a wide-character code representing a character of
22173 class **print** in the program's current locale; see the Base Definitions volume of
22174 IEEE Std. 1003.1-200x, Chapter 7, Locale.

22175 In all cases *wc* is a **wint_t**, the value of which the application shall ensure is a wide-character
22176 code corresponding to a valid character in the current locale, or equal to the value of the macro
22177 WEOF. If the argument has any other value, the behavior is undefined.

22178 **RETURN VALUE**

22179 The *iswprint*( ) function shall return non-zero if *wc* is a printing wide-character code; otherwise, it
22180 shall return 0.

22181 **ERRORS**

22182 No errors are defined.

22183 **EXAMPLES**

22184 None.

22185 **APPLICATION USAGE**

22186 To ensure applications portability, especially across natural languages, only this function and
22187 those listed in the SEE ALSO section should be used for classification of wide-character codes.

22188 **RATIONALE**

22189 None.

22190 **FUTURE DIRECTIONS**

22191 None.

22192 **SEE ALSO**

22193 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswpunct*( ),
22194 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
22195 IEEE Std. 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of
22196 IEEE Std. 1003.1-200x, Chapter 7, Locale

22197 **CHANGE HISTORY**

22198 First released in Issue 4.

22199 **Issue 5**

22200 The following change has been made in this issue for alignment with
22201 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

22202 • The SYNOPSIS has been changed to indicate that this function and associated data types are
22203 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

22206 **NAME**

22207 iswpunct — test for a punctuation wide-character code

22208 **SYNOPSIS**

22209 `#include <wctype.h>`

22210 `int iswpunct(wint_t wc);`

22211 **DESCRIPTION**

22212 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22213 conflict between the requirements described here and the ISO C standard is unintentional. This
22214 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

22215 The *iswpunct*( ) function shall test whether *wc* is a wide-character code representing a character
22216 of class **punct** in the program's current locale; see the Base Definitions volume of
22217 IEEE Std. 1003.1-200x, Chapter 7, Locale.

22218 In all cases *wc* is a **wint_t**, the value of which the application shall ensure is a wide-character
22219 code corresponding to a valid character in the current locale, or equal to the value of the macro
22220 WEOF. If the argument has any other value, the behavior is undefined.

22221 **RETURN VALUE**

22222 The *iswpunct*( ) function shall return non-zero if *wc* is a punctuation wide-character code;
22223 otherwise, it shall return 0.

22224 **ERRORS**

22225 No errors are defined.

22226 **EXAMPLES**

22227 None.

22228 **APPLICATION USAGE**

22229 To ensure applications portability, especially across natural languages, only this function and
22230 those listed in the SEE ALSO section should be used for classification of wide-character codes.

22231 **RATIONALE**

22232 None.

22233 **FUTURE DIRECTIONS**

22234 None.

22235 **SEE ALSO**

22236 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ),
22237 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
22238 IEEE Std. 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of
22239 IEEE Std. 1003.1-200x, Chapter 7, Locale

22240 **CHANGE HISTORY**

22241 First released in Issue 4.

22242 **Issue 5**

22243 The following change has been made in this issue for alignment with
22244 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

22245 • The SYNOPSIS has been changed to indicate that this function and associated data types are
22246 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

22247 **Issue 6**

22248 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

22249 **NAME**

22250 iswspace — test for a white-space wide-character code

22251 **SYNOPSIS**

22252 `#include <wctype.h>`

22253 `int iswspace(wint_t wc);`

22254 **DESCRIPTION**

22255 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22256 conflict between the requirements described here and the ISO C standard is unintentional. This
22257 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

22258 The *iswspace*( ) function shall test whether *wc* is a wide-character code representing a character of
22259 class **space** in the program's current locale; see the Base Definitions volume of
22260 IEEE Std. 1003.1-200x, Chapter 7, Locale.

22261 In all cases *wc* is a **wint_t**, the value of which the application shall ensure is a wide-character
22262 code corresponding to a valid character in the current locale, or equal to the value of the macro
22263 WEOF. If the argument has any other value, the behavior is undefined.

22264 **RETURN VALUE**

22265 The *iswspace*( ) function shall return non-zero if *wc* is a white-space wide-character code;
22266 otherwise, it shall return 0.

22267 **ERRORS**

22268 No errors are defined.

22269 **EXAMPLES**

22270 None.

22271 **APPLICATION USAGE**

22272 To ensure applications portability, especially across natural languages, only this function and
22273 those listed in the SEE ALSO section should be used for classification of wide-character codes.

22274 **RATIONALE**

22275 None.

22276 **FUTURE DIRECTIONS**

22277 None.

22278 **SEE ALSO**

22279 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ),
22280 *iswpunct*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
22281 IEEE Std. 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of
22282 IEEE Std. 1003.1-200x, Chapter 7, Locale

22283 **CHANGE HISTORY**

22284 First released in Issue 4.

22285 **Issue 5**

22286 The following change has been made in this issue for alignment with
22287 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

22288 • The SYNOPSIS has been changed to indicate that this function and associated data types are
22289 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

22290 **Issue 6**

22291 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

22292 **NAME**

22293 iswupper — test for an uppercase letter wide-character code

22294 **SYNOPSIS**

22295 ```
#include <wctype.h>
```

22296 ```
int iswupper(wint_t wc);
```

22297 **DESCRIPTION**

22298 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22299 conflict between the requirements described here and the ISO C standard is unintentional. This
22300 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

22301 The *iswupper*( ) function shall test whether *wc* is a wide-character code representing a character
22302 of class **upper** in the program's current locale; see the Base Definitions volume of
22303 IEEE Std. 1003.1-200x, Chapter 7, Locale.

22304 In all cases *wc* is a **wint_t**, the value of which the application shall ensure is a wide-character
22305 code corresponding to a valid character in the current locale, or equal to the value of the macro
22306 WEOF. If the argument has any other value, the behavior is undefined.

22307 **RETURN VALUE**

22308 The *iswupper*( ) function shall return non-zero if *wc* is an uppercase letter wide-character code;
22309 otherwise, it shall return 0.

22310 **ERRORS**

22311 No errors are defined.

22312 **EXAMPLES**

22313 None.

22314 **APPLICATION USAGE**

22315 To ensure applications portability, especially across natural languages, only this function and
22316 those listed in the SEE ALSO section should be used for classification of wide-character codes.

22317 **RATIONALE**

22318 None.

22319 **FUTURE DIRECTIONS**

22320 None.

22321 **SEE ALSO**

22322 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ),
22323 *iswpunct*( ), *iswspace*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of
22324 IEEE Std. 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of
22325 IEEE Std. 1003.1-200x, Chapter 7, Locale

22326 **CHANGE HISTORY**

22327 First released in Issue 4.

22328 **Issue 5**

22329 The following change has been made in this issue for alignment with
22330 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

22331 • The SYNOPSIS has been changed to indicate that this function and associated data types are
22332 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

22333 **Issue 6**

22334 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

22335 **NAME**

22336 iswxdigit — test for a hexadecimal digit wide-character code

22337 **SYNOPSIS**

22338 `#include <wctype.h>`

22339 `int iswxdigit(wint_t wc);`

22340 **DESCRIPTION**

22341 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22342 conflict between the requirements described here and the ISO C standard is unintentional. This
22343 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

22344 The *iswxdigit*( ) function shall test whether *wc* is a wide-character code representing a character
22345 of class **xdigit** in the program's current locale; see the Base Definitions volume of
22346 IEEE Std. 1003.1-200x, Chapter 7, Locale.

22347 In all cases *wc* is a **wint_t**, the value of which the application shall ensure is a wide-character
22348 code corresponding to a valid character in the current locale, or equal to the value of the macro
22349 WEOF. If the argument has any other value, the behavior is undefined.

22350 **RETURN VALUE**

22351 The *iswxdigit*( ) function shall return non-zero if *wc* is a hexadecimal digit wide-character code;
22352 otherwise, it shall return 0.

22353 **ERRORS**

22354 No errors are defined.

22355 **EXAMPLES**

22356 None.

22357 **APPLICATION USAGE**

22358 To ensure applications portability, especially across natural languages, only this function and
22359 those listed in the SEE ALSO section should be used for classification of wide-character codes.

22360 **RATIONALE**

22361 None.

22362 **FUTURE DIRECTIONS**

22363 None.

22364 **SEE ALSO**

22365 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ),
22366 *iswpunct*( ), *iswspace*( ), *iswupper*( ), *setlocale*( ), the Base Definitions volume of
22367 IEEE Std. 1003.1-200x, **<wctype.h>**, **<wchar.h>**

22368 **CHANGE HISTORY**

22369 First released in Issue 4.

22370 **Issue 5**

22371 The following change has been made in this issue for alignment with
22372 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

22373 • The SYNOPSIS has been changed to indicate that this function and associated data types are
22374 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

22375 **Issue 6**

22376 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

NAME

isxdigit — test for a hexadecimal digit

SYNOPSIS

```
#include <ctype.h>

int isxdigit(int c);
```

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 *isxdigit*( ) function shall test whether *c* is a character of class **xdigit** in the program's current locale; see the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale.

In all cases *c* is an **int**, the value of which the application shall ensure is a character representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

RETURN VALUE

The *isxdigit*( ) function shall return non-zero if *c* is a hexadecimal digit; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>**

CHANGE HISTORY

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

Issue 4

The text of the DESCRIPTION is revised, although there are no functional differences between this issue and Issue 3.

Issue 6

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

22415 **NAME**

22416 j0, j1, jn — Bessel functions of the first kind

22417 **SYNOPSIS**

22418 XSI
```
#include <math.h>

double j0(double x);
double j1(double x);
double jn(int n, double x);
```
22422

22423 **DESCRIPTION**

22424 The *j0*( ), *j1*( ), and *jn*( ) functions shall compute Bessel functions of *x* of the first kind of orders 0,
22425 1, and *n* respectively.

22426 An application wishing to check for error situations should set *errno* to 0 before calling *j0*( ), *j1*( ),
22427 or *jn*( ). If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

22428 **RETURN VALUE**

22429 Upon successful completion, *j0*( ), *j1*( ), and *jn*( ) shall return the relevant Bessel value of *x* of the
22430 first kind.

22431 If the *x* argument is too large in magnitude, 0 shall be returned and *errno* may be set to
22432 [ERANGE]. |

22433 If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM]. |

22434 If the correct result would cause underflow, 0 shall be returned and *errno* may be set to
22435 [ERANGE].

22436 **ERRORS**

22437 The *j0*( ), *j1*( ), and *jn*( ) functions may fail if:

22438 [EDOM] The value of *x* is NaN. |

22439 [ERANGE] The value of *x* was too large in magnitude, or underflow occurred. |

22440 No other errors shall occur.

22441 **EXAMPLES**

22442 None.

22443 **APPLICATION USAGE**

22444 None.

22445 **RATIONALE**

22446 None.

22447 **FUTURE DIRECTIONS**

22448 None.

22449 **SEE ALSO**

22450 *isnan*( ), *y0*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>** |

22451 **CHANGE HISTORY**

22452 First released in Issue 1. Derived from Issue 1 of the SVID. |

22453 **Issue 4**

22454 References to *matherr*( ) are removed.

22455 The RETURN VALUE and ERRORS sections are substantially rewritten to rationalize error
22456 handling in the mathematics functions.

22457 **Issue 5**

22458 The DESCRIPTION is updated to indicate how an application should check for an error. This
22459 text was previously published in the APPLICATION USAGE section.

22460 **NAME**

22461 jrand48 — generate a uniformly distributed pseudo-random long signed integer

22462 **SYNOPSIS**

22463 XSI
```
#include <stdlib.h>

long jrand48(unsigned short xsubi[3]);
```
22464 |

22465 |

22466 **DESCRIPTION**

22467 Refer to *drand48*( ).

22468 **NAME**

22469 kill — send a signal to a process or a group of processes

22470 **SYNOPSIS**

22471 `#include <signal.h>`

22472 `int kill(pid_t pid, int sig);`

22473 **DESCRIPTION**

22474 The *kill*( ) function shall send a signal to a process or a group of processes specified by *pid*. The
22475 signal to be sent is specified by *sig* and is either one from the list given in **<signal.h>** or 0. If *sig* is
22476 0 (the null signal), error checking is performed but no signal is actually sent. The null signal can
22477 be used to check the validity of *pid*.

22478 For a process to have permission to send a signal to a process designated by *pid*, unless the
22479 sending process has appropriate privileges, the application shall ensure that the real or effective
22480 user ID of the sending process matchs the real or saved set-user-ID of the receiving process.

22481 If *pid* is greater than 0, *sig* shall be sent to the process whose process ID is equal to *pid*.

22482 If *pid* is 0, *sig* shall be sent to all processes (excluding an unspecified set of system processes)
22483 whose process group ID is equal to the process group ID of the sender, and for which the
22484 process has permission to send a signal.

22485 If *pid* is –1, *sig* shall be sent to all processes (excluding an unspecified set of system processes) for |
22486 which the process has permission to send that signal. |

22487 If *pid* is negative, but not –1, *sig* shall be sent to all processes (excluding an unspecified set of
22488 system processes) whose process group ID is equal to the absolute value of *pid*, and for which
22489 the process has permission to send a signal.

22490 If the value of *pid* causes *sig* to be generated for the sending process, and if *sig* is not blocked for
22491 the calling thread and if no other thread has *sig* unblocked or is waiting in a *sigwait*( ) function
22492 for *sig*, either *sig* or at least one pending unblocked signal shall be delivered to the sending
22493 thread before *kill*( ) returns.

22494 The user ID tests described above shall not be applied when sending SIGCONT to a process that
22495 is a member of the same session as the sending process.

22496 An implementation that provides extended security controls may impose further |
22497 implementation-defined restrictions on the sending of signals, including the null signal. In |
22498 particular, the system may deny the existence of some or all of the processes specified by *pid*.

22499 The *kill*( ) function is successful if the process has permission to send *sig* to any of the processes
22500 specified by *pid*. If *kill*( ) fails, no signal shall be sent.

22501 **RETURN VALUE**

22502 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
22503 indicate the error.

22504 **ERRORS**

22505 The *kill*( ) function shall fail if:

22506 [EINVAL] The value of the *sig* argument is an invalid or unsupported signal number. |

22507 [EPERM] The process does not have permission to send the signal to any receiving |
22508 process.

22509 [ESRCH] No process or process group can be found corresponding to that specified by |
22510 *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 implementation-defined 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 or nonexistence of a process from being used as a covert channel, such processes should appear nonexistent to the sender; that is, [ESRCH] should be returned, rather than [EPERM], if *pid* refers only to such processes.

Existing implementations vary on the result of a *kill*( ) with *pid* indicating an inactive process (a terminated process that has not been waited for by its parent). Some indicate success on such a call (subject to permission checking), while others give an error of [ESRCH]. Since the definition of process lifetime in this volume of IEEE Std. 1003.1-200x covers inactive processes, the [ESRCH] error as described is inappropriate in this case. In particular, this means that an application cannot have a parent process check for termination of a particular child with *kill*( ). (Usually this is done with the null signal; this can be done reliably with *waitpid*( ).)

There is some belief that the name *kill*( ) is misleading, since the function is not always intended to cause process termination. However, the name is common to all historical implementations, and any change would be in conflict with the goal of minimal changes to existing application code.

FUTURE DIRECTIONS

None.

SEE ALSO

getpid( ), *raise*( ), *setsid*( ), *sigaction*( ), *sigqueue*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<signal.h>**, **<sys/types.h>**

CHANGE HISTORY

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

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

The DESCRIPTION is clarified in various places.

The following change is incorporated for alignment with the FIPS requirements:

- In the DESCRIPTION, the second paragraph is reworded to indicate that the saved set-user-ID of the calling process is checked in place of its effective user ID. This functionality is marked as an extension.

Issue 5

The DESCRIPTION is updated for alignment with POSIX Threads Extension.

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:

- In the DESCRIPTION, the second paragraph is reworded to indicate that the saved set-user-ID of the calling process is checked in place of its effective user ID. This is a FIPS requirement.
- 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 behavior when *pid* is –1 is now specified. It was previously explicitly unspecified in the POSIX.1-1988 standard.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

22599 **NAME**

22600 killpg — send a signal to a process group

22601 **SYNOPSIS**

22602 XSI
```
#include <signal.h>

int killpg(pid_t pgrp, int sig);
```
22603

22604

22605 **DESCRIPTION**

22606 The *killpg*( ) function shall send the signal specified by *sig* to the process group specified by *pgrp*.

22607 If *pgrp* is greater than 1, *killpg*(*pgrp*, *sig*) shall be equivalent to *kill*(–*pgrp*, *sig*). If *pgrp* is less than or
22608 equal to 1, the behavior of *killpg*( ) is undefined.

22609 **RETURN VALUE**

22610 Refer to *kill*( ).

22611 **ERRORS**

22612 Refer to *kill*( ).

22613 **EXAMPLES**

22614 None.

22615 **APPLICATION USAGE**

22616 None.

22617 **RATIONALE**

22618 None.

22619 **FUTURE DIRECTIONS**

22620 None.

22621 **SEE ALSO**

22622 *getpgid*( ), *getpid*( ), *kill*( ), *raise*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
22623 **<signal.h>**

22624 **CHANGE HISTORY**

22625 First released in Issue 4, Version 2.

22626 **Issue 5**

22627 Moved from X/OPEN UNIX extension to BASE.

22628 **NAME**

22629 l64a — convert a 32-bit integer to a radix-64 ASCII string

22630 **SYNOPSIS**

22631 XSI
```
#include <stdlib.h>
```

22632
```
char *l64a(long value);
```

22633

22634 **DESCRIPTION**

22635 Refer to *a64l*( ).

22636 **NAME**

22637 labs, llabs — return a long integer absolute value

22638 **SYNOPSIS**

22639 `#include <stdlib.h>`

22640 `long labs(long i);`

22641 `long long llabs(long long i);`

22642 **DESCRIPTION**

22643 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22644 conflict between the requirements described here and the ISO C standard is unintentional. This
22645 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

22646 These functions shall compute the absolute value of the **long** integer operand, *i*. If the result
22647 cannot be represented, the behavior is undefined.

22648 **RETURN VALUE**

22649 These functions shall return the absolute value of the **long** integer operand.

22650 **ERRORS**

22651 No errors are defined.

22652 **EXAMPLES**

22653 None.

22654 **APPLICATION USAGE**

22655 None.

22656 **RATIONALE**

22657 None.

22658 **FUTURE DIRECTIONS**

22659 None.

22660 **SEE ALSO**

22661 *abs*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>**

22662 **CHANGE HISTORY**

22663 First released in Issue 4. Derived from the ISO C standard.

22664 **Issue 6**

22665 The *llabs*( ) function is added for alignment with the ISO / IEC 9899: 1999 standard.

NAME

lchown — change the owner and group of a symbolic link

SYNOPSIS

XSI
```
#include <unistd.h>

int lchown(const char *path, uid_t owner, gid_t group);
```

DESCRIPTION

The *lchown*( ) function shall have the same effect as *chown*( ) except in the case where the named file is a symbolic link. In this case, *lchown*( ) shall change the ownership of the symbolic link file itself, while *chown*( ) changes the ownership of the file or directory to which the symbolic link refers.

RETURN VALUE

Upon successful completion, *lchown*( ) shall return 0. Otherwise, it shall return –1 and set *errno* to indicate an error.

ERRORS

The *lchown*( ) function shall fail if:

[EACCES] Search permission is denied on a component of the path prefix of *path*.

[EINVAL] The owner or group ID is not a value supported by the implementation.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

[ENAMETOOLONG] The length of a path name exceeds {PATH_MAX} or a path name component is longer than {NAME_MAX}.

[ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

[ENOTDIR] A component of the path prefix of *path* is not a directory.

[EOPNOTSUPP] The *path* argument names a symbolic link and the implementation does not support setting the owner or group of a symbolic link.

[EPERM] The effective user ID does not match the owner of the file and the process does not have appropriate privileges.

[EROFS] The file resides on a read-only file system.

The *lchown*( ) function may fail if:

[EIO] An I/O error occurred while reading or writing to the file system.

[EINTR] A signal was caught during execution of the function.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG] Path name resolution of a symbolic link produced an intermediate result 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

None.

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.

22736 **NAME**

22737 lcong48 — seed a uniformly distributed pseudo-random signed long integer generator

22738 **SYNOPSIS**

22739 XSI
```
#include <stdlib.h>

void lcong48(unsigned short param[7]);
```

22740 |

22741 |

22742 **DESCRIPTION**

22743 Refer to *drand48*( ).

22744 **NAME**

22745 ldexp, ldexpf, ldexpl — load exponent of a floating point number

22746 **SYNOPSIS**

22747
```
#include <math.h>
```

22748
```
double ldexp(double x, int exp);
```
22749
```
float ldexpf(float x, int exp);
```
22750
```
long double ldexpl(long double x, int exp);
```

22751 **DESCRIPTION**

22752 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22753 conflict between the requirements described here and the ISO C standard is unintentional. This
22754 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

22755 These functions shall compute the quantity $x * 2^{exp}$.

22756 An application wishing to check for error situations should set *errno* to 0 before calling *ldexp*().
22757 If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

22758 **RETURN VALUE**

22759 Upon successful completion, these functions shall return *x* multiplied by 2, raised to the power
22760 *exp*.

22761 XSI If the value of *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

22762 If *ldexp*() would cause overflow, ±HUGE_VAL shall be returned (according to the sign of *x*), and
22763 *errno* shall be set to [ERANGE].

22764 If *ldexp*() would cause underflow, 0 shall be returned and *errno* may be set to [ERANGE].

22765 **ERRORS**

22766 These functions shall fail if:

22767 [ERANGE] The value to be returned would have caused overflow.

22768 These functions may fail if:

22769 XSI [EDOM] The argument *x* is NaN.

22770 [ERANGE] The value to be returned would have caused underflow.

22771 No other errors shall occur.

22772 **EXAMPLES**

22773 None.

22774 **APPLICATION USAGE**

22775 None.

22776 **RATIONALE**

22777 None.

22778 **FUTURE DIRECTIONS**

22779 None.

22780 **SEE ALSO**

22781 *frexp*(), *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 4

References to *matherr*( ) are removed.

The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the ISO C standard and to rationalize error handling in the mathematics functions.

The return value specified for [EDOM] is marked as an extension.

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 *ldexpf*( ) and *ldexpl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

22795 **NAME**

22796 ldiv, lldiv — compute quotient and remainder of a long division

22797 **SYNOPSIS**

22798
```
#include <stdlib.h>
```

22799
```
ldiv_t ldiv(long numer, long denom);
```
22800
```
lldiv_t lldiv(long long numer, long long denom);
```

22801 **DESCRIPTION**

22802 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22803 conflict between the requirements described here and the ISO C standard is unintentional. This
22804 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

22805 These functions shall compute the quotient and remainder of the division of the numerator
22806 *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the **long**
22807 integer of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be
22808 represented, the behavior is undefined; otherwise, *quot* * *denom*+*rem* shall equal *numer*.

22809 **RETURN VALUE**

22810 These functions shall return a structure of type **ldiv_t**, comprising both the quotient and the
22811 remainder. The structure includes the following members, in any order:

22812
```
long   quot;    /* Quotient */
```
22813
```
long   rem;     /* Remainder */
```

22814 **ERRORS**

22815 No errors are defined.

22816 **EXAMPLES**

22817 None.

22818 **APPLICATION USAGE**

22819 None.

22820 **RATIONALE**

22821 None.

22822 **FUTURE DIRECTIONS**

22823 None.

22824 **SEE ALSO**

22825 *div*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>**

22826 **CHANGE HISTORY**

22827 First released in Issue 4. Derived from the ISO C standard.

22828 **Issue 6**

22829 The *lldiv*( ) function is added for alignment with the ISO / IEC 9899: 1999 standard.

22830 **NAME**

22831 lfind — find entry in a linear search table

22832 **SYNOPSIS**

22833 XSI
```
#include <search.h>
```

22834
```
void *lfind(const void *key, const void *base, size_t *nelp,
    size_t width, int (*compar)(const void *, const void *));
```

22836

22837 **DESCRIPTION**

22838 Refer to *lsearch*( ).

22839 **NAME**

22840 lgamma, lgammaf, lgammal — log gamma function

22841 **SYNOPSIS**

22842 XSI
```
#include <math.h>

double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);
extern int signgam;
```

22847

22848 **DESCRIPTION**

22849 These functions function shall compute $\log_e|\Gamma(x)|$ where $\Gamma(x)$ is defined as $\int_0^\infty e^{-t}t^{x-1}dt$. The sign of
22850 $\Gamma(x)$ is returned in the external integer *signgam*. The argument *x* need not be a non-positive
22852 integer ($\Gamma(x)$ is defined over the reals, except the non-positive integers).

22853 An application wishing to check for error situations should set *errno* to 0 before calling *lgamma*( ).
22854 If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

22855 These functions need not be reentrant. A function that is not required to be reentrant is not
22856 required to be thread-safe.

22857 **RETURN VALUE**

22858 Upon successful completion, these functions shall return the logarithmic gamma of *x*.

22859 If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

22860 If *x* is a non-positive integer, either HUGE_VAL or NaN shall be returned and *errno* may be set to
22861 [ERANGE].

22862 If the correct value would cause overflow, *lgamma*( ) shall return HUGE_VAL and may set *errno*
22863 to [ERANGE].

22864 If the correct value would cause underflow, *lgamma*( ) shall return 0 and may set *errno* to
22865 [ERANGE].

22866 **ERRORS**

22867 These functions may fail if:

22868 [EDOM] The value of *x* is NaN.

22869 [ERANGE] The value of *x* is a non-positive integer, or the value to be returned would
22870 have caused overflow or underflow.

22871 No other errors shall occur.

22872 **EXAMPLES**

22873 None.

22874 **APPLICATION USAGE**

22875 None.

22876 **RATIONALE**

22877 None.

22878 **FUTURE DIRECTIONS**

22879 None.

SEE ALSO

exp( ), *isnan*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 4

This page no longer points to *gamma*( ), but contains all information relating to *lgamma*( ).

The RETURN VALUE and ERRORS sections are substantially rewritten to rationalize error handling in the mathematics functions.

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.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

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

- The *lgammaf*( ) and *lgammal*( ) functions are added.
- The RETURN VALUE and ERRORS sections are updated so that when *x* is a non-positive integer, *errno* may be set to [ERANGE]; previously, this was [EDOM].

22897 **NAME**

22898 link — link to a file

22899 **SYNOPSIS**

22900 `#include <unistd.h>`

22901 `int link(const char *`*`path1`*`, const char *`*`path2`*`);`

22902 **DESCRIPTION**

22903 The *link*( ) function shall create a new link (directory entry) for the existing file, *path1*.

22904 The *path1* argument points to a path name naming an existing file. The *path2* argument points to
22905 a path name naming the new directory entry to be created. The *link*( ) function shall atomically
22906 create a new link for the existing file and the link count of the file shall be incremented by one.

22907 If *path1* names a directory, *link*( ) shall fail unless the process has appropriate privileges and the
22908 implementation supports using *link*( ) on directories.

22909 Upon successful completion, *link*( ) shall mark for update the *st_ctime* field of the file. Also, the
22910 *st_ctime* and *st_mtime* fields of the directory that contains the new entry shall be marked for
22911 update.

22912 If *link*( ) fails, no link shall be created and the link count of the file shall remain unchanged.

22913 The implementation may require that the calling process has permission to access the existing
22914 file.

22915 **RETURN VALUE**

22916 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
22917 indicate the error.

22918 **ERRORS**

22919 The *link*( ) function shall fail if:

22920 [EACCES] A component of either path prefix denies search permission, or the requested |
22921 link requires writing in a directory that denies write permission, or the calling
22922 process does not have permission to access the existing file and this is
22923 required by the implementation.

22924 [EEXIST] The *path2* argument resolves to an existing file or refers to a symbolic link. |

22925 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path1* or |
22926 *path2* argument. |

22927 [EMLINK] The number of links to the file named by *path1* would exceed {LINK_MAX}. |

22928 [ENAMETOOLONG] |
22929 The length of the *path1* or *path2* argument exceeds {PATH_MAX} or a path |
22930 name component is longer than {NAME_MAX}. |

22931 [ENOENT] A component of either path prefix does not exist; the file named by *path1* does |
22932 not exist; or *path1* or *path2* points to an empty string.

22933 [ENOSPC] The directory to contain the link cannot be extended. |

22934 [ENOTDIR] A component of either path prefix is not a directory. |

22935 [EPERM] The file named by *path1* is a directory and either the calling process does not |
22936 have appropriate privileges or the implementation prohibits using *link*( ) on
22937 directories.

[EROFS] The requested link requires writing in a directory on a read-only file system.

[EXDEV] The link named by *path2* and the file named by *path1* are on different file systems and the implementation does not support links between file systems, or *path1* refers to a named STREAM. XSR

The *link*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path1* or *path2* argument.

[ENAMETOOLONG] As a result of encountering a symbolic link in resolution of the *path1* or *path2* argument, the length of the substituted path name string exceeded {PATH_MAX}.

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 is used to link 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.

FUTURE DIRECTIONS

None.

SEE ALSO

symlink( ), *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 4

The **<unistd.h>** header is added to the SYNOPSIS section.

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The type of arguments *path1* and *path2* are changed from **char*** to **const char***.

The following change is incorporated for alignment with the FIPS requirements:

- In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path name component is larger that {NAME_MAX} is now defined as mandatory and marked as an extension.

Issue 4, Version 2

The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- The [ELOOP] error is returned if too many symbolic links are encountered during path name resolution.
- The [EXDEV] error may also be returned if *path1* refers to a named STREAM.
- A second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of path name resolution of a symbolic link.

Issue 6

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

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:

- An explanation is added of action when *path2* refers to a symbolic link.

23018 • The [ELOOP] optional error condition is added.

23019 **NAME**

23020 lio_listio — list directed I/O (**REALTIME**)

23021 **SYNOPSIS**

23022 AIO
```
#include <aio.h>

int lio_listio(int mode, struct aiocb *restrict const list[restrict],
    int nent, struct sigevent *restrict sig);
```

23026 **DESCRIPTION**

23027 The *lio_listio*( ) function allows the calling process to initiate a list of I/O requests with a single
23028 function call.

23029 The *mode* argument takes one of the values LIO_WAIT or LIO_NOWAIT declared in **<aio.h>** and
23030 determines whether the function returns when the I/O operations have been completed, or as
23031 soon as the operations have been queued. If the *mode* argument is LIO_WAIT, the function waits
23032 until all I/O is complete and the *sig* argument is ignored.

23033 If the *mode* argument is LIO_NOWAIT, the function shall return immediately, and asynchronous
23034 notification shall occur, according to the *sig* argument, when all the I/O operations complete. If
23035 *sig* is NULL, then no asynchronous notification shall occur. If *sig* is not NULL, asynchronous
23036 notification occurs as specified in Section 2.4.1 (on page 528) when all the requests in *list* have
23037 completed.

23038 The I/O requests enumerated by *list* are submitted in an unspecified order.

23039 The *list* argument is an array of pointers to **aiocb** structures. The array contains *nent* elements.
23040 The array may contain NULL elements, which shall be ignored.

23041 The *aio_lio_opcode* field of each **aiocb** structure specifies the operation to be performed. The
23042 supported operations are LIO_READ, LIO_WRITE, and LIO_NOP; these symbols are defined in
23043 **<aio.h>**. The LIO_NOP operation causes the list entry to be ignored. If the *aio_lio_opcode*
23044 element is equal to LIO_READ, then an I/O operation is submitted as if by a call to *aio_read*( )
23045 with the *aiocbp* equal to the address of the **aiocb** structure. If the *aio_lio_opcode* element is equal
23046 to LIO_WRITE, then an I/O operation is submitted as if by a call to *aio_write*( ) with the *aiocbp*
23047 equal to the address of the **aiocb** structure.

23048 The *aio_fildes* member specifies the file descriptor on which the operation is to be performed.

23049 The *aio_buf* member specifies the address of the buffer to or from which the data is transferred.

23050 The *aio_nbytes* member specifies the number of bytes of data to be transferred.

23051 The members of the **aiocb** structure further describe the I/O operation to be performed, in a
23052 manner identical to that of the corresponding **aiocb** structure when used by the *aio_read*( ) and
23053 *aio_write*( ) functions.

23054 The *nent* argument specifies how many elements are members of the list; that is, the length of the
23055 array.

23056 The behavior of this function is altered according to the definitions of synchronized I/O data
23057 integrity completion and synchronized I/O file integrity completion if synchronized I/O is
23058 enabled on the file associated with *aio_fildes*.

23059 For regular files, no data transfer shall occur past the offset maximum established in the open
23060 file description associated with *aiocbp->aio_fildes*.

23061 **RETURN VALUE**

23062 If the *mode* argument has the value LIO_NOWAIT, the *lio_listio*( ) function shall return the value
23063 zero if the I/O operations are successfully queued; otherwise, the function shall return the value
23064 –1 and set *errno* to indicate the error.

23065 If the *mode* argument has the value LIO_WAIT, the *lio_listio*( ) function shall return the value
23066 zero when all the indicated I/O has completed successfully. Otherwise, *lio_listio*( ) shall return a
23067 value of –1 and set *errno* to indicate the error.

23068 In either case, the return value only indicates the success or failure of the *lio_listio*( ) call itself,
23069 not the status of the individual I/O requests. In some cases one or more of the I/O requests
23070 contained in the list may fail. Failure of an individual request does not prevent completion of
23071 any other individual request. To determine the outcome of each I/O request, the application
23072 shall examine the error status associated with each **aiocb** control block. The error statuses so
23073 returned are identical to those returned as the result of an *aio_read*( ) or *aio_write*( ) function.

23074 **ERRORS**

23075 The *lio_listio*( ) function shall fail if:

23076 [EAGAIN] The resources necessary to queue all the I/O requests were not available. The |
23077 application may check the error status for each **aiocb** to determine the
23078 individual request(s) that failed.

23079 [EAGAIN] The number of entries indicated by *nent* would cause the system-wide limit
23080 {AIO_MAX} to be exceeded.

23081 [EINVAL] The *mode* argument is not a proper value, or the value of *nent* was greater than |
23082 {AIO_LISTIO_MAX}.

23083 [EINTR] A signal was delivered while waiting for all I/O requests to complete during |
23084 an LIO_WAIT operation. Note that, since each I/O operation invoked by
23085 *lio_listio*( ) may possibly provoke a signal when it completes, this error return
23086 may be caused by the completion of one (or more) of the very I/O operations
23087 being awaited. Outstanding I/O requests are not canceled, and the application
23088 shall examine each list element to determine whether the request was
23089 initiated, canceled, or completed.

23090 [EIO] One or more of the individual I/O operations failed. The application may |
23091 check the error status for each **aiocb** structure to determine the individual
23092 request(s) that failed.

23093 In addition to the errors returned by the *lio_listio*( ) function, if the *lio_listio*( ) function succeeds
23094 or fails with errors of [EAGAIN], [EINTR], or [EIO], then some of the I/O specified by the list
23095 may have been initiated. If the *lio_listio*( ) function fails with an error code other than [EAGAIN],
23096 [EINTR], or [EIO], no operations from the list shall have been initiated. The I/O operation
23097 indicated by each list element can encounter errors specific to the individual read or write
23098 function being performed. In this event, the error status for each **aiocb** control block contains the
23099 associated error code. The error codes that can be set are the same as would be set by a *read*( ) or
23100 *write*( ) function, with the following additional error codes possible:

23101 [EAGAIN] The requested I/O operation was not queued due to resource limitations. |

23102 [ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit |
23103 *aio_cancel*( ) request. |

23104 [EFBIG] The *aiocbp->aio_lio_opcode* is LIO_WRITE, the file is a regular file, *aiocbp-* |
23105 *>aio_nbytes* is greater than 0, and the *aiocbp->aio_offset* is greater than or equal |
23106 to the offset maximum in the open file description associated with *aiocbp-* |

23107 *>aio_fildes*.

23108 [EINPROGRESS] The requested I/O is in progress.

23109 [EOVERFLOW] The *aiocbp->aio_lio_opcode* is LIO_READ, the file is a regular file, *aiocbp->aio_nbytes* is greater than 0, and the *aiocbp->aio_offset* is before the end-of-file and is greater than or equal to the offset maximum in the open file description associated with *aiocbp->aio_fildes*.

23113 **EXAMPLES**

23114 None.

23115 **APPLICATION USAGE**

23116 None.

23117 **RATIONALE**

23118 Although it may appear that there are inconsistencies in the specified circumstances for error
23119 codes, the [EIO] error condition applies when any circumstance relating to an individual
23120 operation makes that operation fail. This might be due to a badly formulated request (for
23121 example, the *aio_lio_opcode* field is invalid, and *aio_error*( ) returns [EINVAL]) or might arise from
23122 application behavior (for example, the file descriptor is closed before the operation is initiated,
23123 and *aio_error*( ) returns [EBADF]).

23124 The limitation on the set of error codes returned when operations from the list shall have been
23125 initiated enables applications to know when operations have been started and whether
23126 *aio_error*( ) is valid for a specific operation.

23127 **FUTURE DIRECTIONS**

23128 None.

23129 **SEE ALSO**

23130 *aio_read*( ), *aio_write*( ), *aio_error*( ), *aio_return*( ), *aio_cancel*( ), *close*( ), *exec*, *exit*( ), *fork*( ), *lseek*( ),
23131 *read*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<aio.h>**

23132 **CHANGE HISTORY**

23133 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

23134 Large File Summit extensions are added.

23135 **Issue 6**

23136 The [ENOSYS] error condition has been removed as stubs need not be provided if an
23137 implementation does not support the Asynchronous Input and Output option.

23138 The *lio_listio*( ) function is marked as part of the Asynchronous Input and Output option.

23139 The following new requirements on POSIX implementations derive from alignment with the
23140 Single UNIX Specification:

23141 • In the DESCRIPTION, text is added to indicate that for regular files no data transfer occurs
23142 past the offset maximum established in the open file description associated with *aiocbp->aio_fildes*. This change is to support large files.

23144 • The [EBIG] and [EOVERFLOW] error conditions are defined. This change is to support large
23145 files.

23146 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

23147 The **restrict** keyword is added to the *lio_listio*( ) prototype for alignment with the
23148 ISO/IEC 9899:1999 standard.

23149 **NAME**

23150 listen — listen for socket connections and limit the queue of incoming connections

23151 **SYNOPSIS**

23152 `#include <sys/socket.h>`

23153 `int listen(int socket, int backlog);`

23154 **DESCRIPTION**

23155 The *listen*( ) function shall mark a connection-mode socket, specified by the *socket* argument, as
23156 accepting connections.

23157 The *backlog* argument provides a hint to the implementation which the implementation shall use
23158 to limit the number of outstanding connections in the socket's listen queue. Implementations
23159 may impose a limit on *backlog* and silently reduce the specified value. Normally, a larger *backlog*
23160 argument value shall result in a larger or equal length of the listen queue. Implementations shall
23161 support values of *backlog* up to SOMAXCONN, defined in **<sys/socket.h>**.

23162 The implementation may include incomplete connections in its listen queue. The limits on the
23163 number of incomplete connections and completed connections queued may be different.

23164 The implementation may have an upper limit on the length of the listen queue—either global or
23165 per accepting socket. If *backlog* exceeds this limit, the length of the listen queue is set to the limit.

23166 If *listen*( ) is called with a *backlog* argument value that is less than 0, the function behaves as if it
23167 had been called with a *backlog* argument value of 0.

23168 A *backlog* argument of 0 may allow the socket to accept connections, in which case the length of
23169 the listen queue may be set to an implementation-defined minimum value.

23170 The socket in use may require the process to have appropriate privileges to use the *listen*( )
23171 function.

23172 **RETURN VALUE**

23173 Upon successful completions, *listen*( ) shall return 0; otherwise, –1 shall be returned and *errno* set
23174 to indicate the error.

23175 **ERRORS**

23176 The *listen*( ) function shall fail if:

23177 [EBADF] The *socket* argument is not a valid file descriptor.

23178 [EDESTADDRREQ]
23179 The socket is not bound to a local address, and the protocol does not support
23180 listening on an unbound socket.

23181 [EINVAL] The *socket* is already connected.

23182 [ENOTSOCK] The *socket* argument does not refer to a socket.

23183 [EOPNOTSUPP] The socket protocol does not support *listen*( ).

23184 The *listen*( ) function may fail if:

23185 [EACCES] The calling process does not have the appropriate privileges.

23186 [EINVAL] The *socket* has been shut down.

23187 [ENOBUFS] Insufficient resources are available in the system to complete the call.

23188 **EXAMPLES**

23189 None.

23190 **APPLICATION USAGE**

23191 None.

23192 **RATIONALE**

23193 None.

23194 **FUTURE DIRECTIONS**

23195 None.

23196 **SEE ALSO**

23197 *accept*( ), *connect*( ), *socket*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/socket.h>** |

23198 **CHANGE HISTORY**

23199 First released in Issue 6. Derived from the XNS, Issue 5.2 specification. |

23200 The DESCRIPTION is updated to describe the relationship of SOMAXCONN and the *backlog* |
23201 argument.

23202 **NAME**

23203 llrint, llrintf, llrintl, lrint, lrintf, lrinfl — round to nearest integer value using current rounding
23204 direction

23205 **SYNOPSIS**

23206 `#include <math.h>`

23207 `long long llrint(double x);`
23208 `long long llrintf(float x);`
23209 `long long llrintl(long double x);`
23210 `long lrint(double x);`
23211 `long lrintf(float x);`
23212 `long lrintl(long double x);`

23213 **DESCRIPTION**

23214 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23215 conflict between the requirements described here and the ISO C standard is unintentional. This
23216 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

23217 These functions shall round their argument to the nearest integer value, rounding according to
23218 the current rounding direction.

23219 An application wishing to check for error situations should set *errno* to 0 before calling these
23220 functions. If *errno* is non-zero on return, an error has occurred.

23221 **RETURN VALUE**

23222 Upon successful completion, these functions shall return the rounded integer value.

23223 If the rounded value is outside the range of the return type, the numeric result is unspecified.

23224 If the magnitude of *x* is too large, the numeric result is unspecified and *errno* may be set to
23225 [ERANGE].

23226 **ERRORS**

23227 These functions may fail if:

23228 [ERANGE] The magnitude of *x* is too large.

23229 **EXAMPLES**

23230 None.

23231 **APPLICATION USAGE**

23232 None.

23233 **RATIONALE**

23234 None.

23235 **FUTURE DIRECTIONS**

23236 None.

23237 **SEE ALSO**

23238 The Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

23239 **CHANGE HISTORY**

23240 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

23241 **NAME**

23242 llround, llroundf, llroundl, lround, lroundf, lroundl — round to nearest integer value

23243 **SYNOPSIS**

23244 `#include <math.h>`

23245 `long long llround(double x);`
23246 `long long llroundf(float x);`
23247 `long long llroundl(long double x);`
23248 `long lround(double x);`
23249 `long lroundf(float x);`
23250 `long lroundl(long double x);`

23251 **DESCRIPTION**

23252 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23253 conflict between the requirements described here and the ISO C standard is unintentional. This
23254 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

23255 These functions shall round their argument to the nearest integer value, rounding halfway cases
23256 away from zero, regardless of the current rounding direction.

23257 An application wishing to check for error situations should set *errno* to 0 before calling these
23258 functions. If *errno* is non-zero on return, an error has occurred.

23259 **RETURN VALUE**

23260 Upon successful completion, these functions shall return the rounded integer value.

23261 If the rounded value is outside the range of the return type, the numeric result is unspecified.

23262 If the magnitude of *x* is too large, the numeric result is unspecified and *errno* may be set to
23263 [ERANGE].

23264 **ERRORS**

23265 These functions may fail if:

23266 [ERANGE] The magnitude of *x* is too large.

23267 **EXAMPLES**

23268 None.

23269 **APPLICATION USAGE**

23270 None.

23271 **RATIONALE**

23272 None.

23273 **FUTURE DIRECTIONS**

23274 None.

23275 **SEE ALSO**

23276 The Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

23277 **CHANGE HISTORY**

23278 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

localeconv — return locale-specific information

SYNOPSIS

```
#include <locale.h>

struct lconv *localeconv(void);
```

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 *localeconv*( ) function shall set the components of an object with the type **struct lconv** with the values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

The members of the structure with type **char*** are pointers to strings, any of which (except **decimal_point**) can point to "", to indicate that the value is not available in the current locale or is of zero length. The members with type **char** are non-negative numbers, any of which can be {CHAR_MAX} to indicate that the value is not available in the current locale.

The members include the following:

char *decimal_point
: The radix character used to format non-monetary quantities.

char *thousands_sep
: The character used to separate groups of digits before the decimal-point character in formatted non-monetary quantities.

char *grouping
: A string whose elements taken as one-byte integer values indicate the size of each group of digits in formatted non-monetary quantities.

char *int_curr_symbol
: The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in the ISO 4217:1995 standard. The fourth character (immediately preceding the null byte) is the character used to separate the international currency symbol from the monetary quantity.

char *currency_symbol
: The local currency symbol applicable to the current locale.

char *mon_decimal_point
: The radix character used to format monetary quantities.

char *mon_thousands_sep
: The separator for groups of digits before the decimal-point in formatted monetary quantities.

char *mon_grouping
: A string whose elements taken as one-byte integer values indicate the size of each group of digits in formatted monetary quantities.

char *positive_sign
: The string used to indicate a non-negative valued formatted monetary quantity.

23322 **char *negative_sign**
23323 The string used to indicate a negative valued formatted monetary quantity.

23324 **char int_frac_digits**
23325 The number of fractional digits (those after the decimal-point) to be displayed in an
23326 internationally formatted monetary quantity.

23327 **char frac_digits**
23328 The number of fractional digits (those after the decimal-point) to be displayed in a
23329 formatted monetary quantity.

23330 **char p_cs_precedes**
23331 Set to 1 if the **currency_symbol** or **int_curr_symbol** precedes the value for a non-negative |
23332 formatted monetary quantity. Set to 0 if the symbol succeeds the value.

23333 **char p_sep_by_space**
23334 Set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value for a |
23335 non-negative formatted monetary quantity. Set to 1 if a space separates the symbol from the
23336 XSI value; and set to 2 if a space separates the symbol and the sign string, if adjacent.

23337 **char n_cs_precedes**
23338 Set to 1 if the **currency_symbol** or **int_curr_symbol** precedes the value for a negative |
23339 formatted monetary quantity. Set to 0 if the symbol succeeds the value.

23340 **char n_sep_by_space**
23341 Set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value for a |
23342 negative formatted monetary quantity. Set to 1 if a space separates the symbol from the
23343 XSI value; and set to 2 if a space separates the symbol and the sign string, if adjacent.

23344 **char p_sign_posn**
23345 Set to a value indicating the positioning of the **positive_sign** for a non-negative formatted
23346 monetary quantity.

23347 **char n_sign_posn**
23348 Set to a value indicating the positioning of the **negative_sign** for a negative formatted
23349 monetary quantity. |

23350 **char int_p_cs_precedes** |
23351 Set to 1 or 0 if the **int_currency_symbol** respectively precedes or succeeds the value for a |
23352 non-negative internationally formatted monetary quantity. |

23353 **char int_n_cs_precedes** |
23354 Set to 1 or 0 if the **int_currency_symbol** respectively precedes or succeeds the value for a |
23355 negative internationally formatted monetary quantity. |

23356 **char int_p_sep_by_space** |
23357 Set to a value indicating the separation of the **int_currency_symbol**, the sign string, and the |
23358 value for a non-negative internationally formatted monetary quantity. |

23359 **char int_n_sep_by_space** |
23360 Set to a value indicating the separation of the **int_currency_symbol**, the sign string, and the |
23361 value for a negative internationally formatted monetary quantity. |

23362 **char int_p_sign_posn** |
23363 Set to a value indicating the positioning of the **positive_sign** for a non-negative |
23364 internationally formatted monetary quantity. |

23365 **char int_n_sign_posn** |
23366 Set to a value indicating the positioning of the **negative_sign** for a negative internationally |

23367 formatted monetary quantity.

23368 The elements of **grouping** and **mon_grouping** are interpreted according to the following:

23369 {CHAR_MAX} No further grouping is to be performed.

23370 0 The previous element is to be repeatedly used for the remainder of the digits.

23371 *other* The integer value is the number of digits that comprise the current group. The
23372 next element is examined to determine the size of the next group of digits
23373 before the current group.

23374 The values of **p_sep_by_space**, **n_sep_by_space**, **int_p_sep_by_space**, and **int_n_sep_by_space**
23375 are interpreted according to the following:

23376 0 No space separates the currency symbol and value.

23377 1 If the currency symbol and sign string are adjacent, a space separates them from the value;
23378 otherwise, a space separates the currency symbol from the value.

23379 2 If the currency symbol and sign string are adjacent, a space separates them; otherwise, a
23380 space separates the sign string from the value.

23381 The values of **p_sign_posn**, **n_sign_posn**, **int_p_sign_posn**, and **int_n_sign_posn** are
23382 interpreted according to the following:

23383 0 Parentheses surround the quantity and **currency_symbol** or **int_curr_symbol**.

23384 1 The sign string precedes the quantity and **currency_symbol** or **int_curr_symbol**.

23385 2 The sign string succeeds the quantity and **currency_symbol** or **int_curr_symbol**.

23386 3 The sign string immediately precedes the **currency_symbol** or **int_curr_symbol**.

23387 4 The sign string immediately succeeds the **currency_symbol** or **int_curr_symbol**.

23388 The implementation shall behave as if no function in this volume of IEEE Std. 1003.1-200x calls
23389 *localeconv*( ).

23390 CX The *localeconv*( ) function need not be reentrant. A function that is not required to be reentrant is
23391 not required to be thread-safe.

23392 **RETURN VALUE**

23393 The *localeconv*( ) function shall return a pointer to the filled-in object. The application shall not
23394 modify the structure pointed to by the return value which may be overwritten by a subsequent
23395 call to *localeconv*( ). In addition, calls to *setlocale*( ) with the categories *LC_ALL*, *LC_MONETARY*,
23396 or *LC_NUMERIC* may overwrite the contents of the structure.

23397 **ERRORS**

23398 No errors are defined.

23399 **EXAMPLES**

23400 None.

23401 **APPLICATION USAGE**

23402 The following table illustrates the rules which may be used by four countries to format monetary
23403 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

For these four countries, the respective values for the monetary members of the structure returned by *localeconv*( ) are:

	Italy	Netherlands	Norway	Switzerland
int_curr_symbol	"ITL."	"NLG "	"NOK "	"CHF "
currency_symbol	"L."	"F"	"kr"	"SFrs."
mon_decimal_point	""	","	","	"."
mon_thousands_sep	"."	"."	"."	","
mon_grouping	"\3"	"\3"	"\3"	"\3"
positive_sign	""	""	""	""
negative_sign	"-"	"-"	"-"	"C"
int_frac_digits	0	2	2	2
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1
p_sep_by_space	0	1	0	0
n_cs_precedes	1	1	1	1
n_sep_by_space	0	1	0	0
p_sign_posn	1	1	1	1
n_sign_posn	1	4	2	2
int_p_cs_precedes	1	1	1	1
int_n_cs_precedes	1	1	1	1
int_p_sep_by_space	0	0	0	0
int_n_sep_by_space	0	0	0	0
int_p_sign_posn	1	1	1	1
int_n_sign_posn	1	4	4	2

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalpha( ), *isascii*( ), *nl_langinfo*( ), *printf*( ), *scanf*( ), *setlocale*( ), *strcat*( ), *strchr*( ), *strcmp*( ), *strcoll*( ), *strcpy*( ), *strftime*( ), *strlen*( ), *strpbrk*( ), *strspn*( ), *strtok*( ), *strxfrm*( ), *strtod*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<langinfo.h>**, **<locale.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

Issue 6

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

The RETURN VALUE section is rewritten to avoid use of the term ‘‘must’’.

This reference page is updated for alignment with the ISO / IEC 9899: 1999 standard.

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);
struct tm *localtime_r(const time_t *restrict timer,
    struct tm *restrict result);
```

DESCRIPTION

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. Local timezone information is used as though *localtime*() calls *tzset*().

The *localtime*() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

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*.

RETURN VALUE

The *localtime*() function shall return a pointer to the broken-down time structure.

Upon successful completion, *localtime_r*() shall return a pointer to the structure pointed to by the argument *result*.

ERRORS

No errors are defined.

EXAMPLES

Getting the Local Date and Time

The following example uses the *time*() function to calculate the time elapsed, in seconds, since January 1, 1970 0:00 UTC (the Epoch), *localtime*() to convert that value to a broken-down time, and *asctime*() to convert the broken-down time values into a printable string.

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

main()
{
    time_t result;

    result = time(NULL);
    printf("%s%ld secs since the Epoch\n",
        asctime(localtime(&result)),
            (long)result);
    return(0);
}
```

23492 This example writes the current time to *stdout* in a form like this:

```
23493 Wed Jun 26 10:32:15 1996
23494 835810335 secs since the Epoch
```

23495 **Getting the Modification Time for a File**

23496 The following example gets the modification time for a file. The *localtime*( ) function converts the
23497 **time_t** value of the last modification date, obtained by a previous call to *stat*( ), into a **tm**
23498 structure that contains the year, month, day, and so on.

```
23499 #include <time.h>
23500 ...
23501 struct stat statbuf;
23502 ...
23503 tm = localtime(&statbuf.st_mtime);
23504 ...
```

23505 **Timing an Event**

23506 The following example gets the current time, converts it to a string using *localtime*( ) and
23507 *asctime*( ), and prints it to standard output using *fputs*( ). It then prints the number of minutes to
23508 an event being timed.

```
23509 #include <time.h>
23510 #include <stdio.h>
23511 ...
23512 time_t now;
23513 int minutes_to_event;
23514 ...
23515 time(&now);
23516 printf("The time is ");
23517 fputs(asctime(localtime(&now)), stdout);
23518 printf("There are still %d minutes to the event.\n",
23519     minutes_to_event);
23520 ...
```

23521 **APPLICATION USAGE**

23522 The *asctime*( ), *ctime*( ), *getdate*( ), *gettimeofday*( ), *gmtime*( ), and *localtime*( ) functions return values
23523 in one of two static objects: a broken-down time structure and an array of **char**. Execution of any
23524 of the functions may overwrite the information returned in either of these objects by any of the
23525 other functions.

23526 The *localtime_r*( ) function is thread-safe and shall return values in a user-supplied buffer instead
23527 of possibly using a static data area that may be overwritten by each call.

23528 **RATIONALE**

23529 None.

23530 **FUTURE DIRECTIONS**

23531 None.

23532 **SEE ALSO**

23533 *asctime*( ), *clock*( ), *ctime*( ), *difftime*( ), *getdate*( ), *gettimeofday*( ), *gmtime*( ), *mktime*( ), *strftime*( ),
23534 *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 4

The APPLICATION USAGE section is expanded to provide a more complete description of how static areas are used by the **time*( ) functions.

The following change is incorporated for alignment with the ISO C standard:

- The *timer* argument is now a type **const time_t**.

Issue 5

A note indicating that the *localtime*( ) function need not be reentrant is added to the DESCRIPTION.

The *localtime_r*( ) function is included for alignment with the POSIX Threads Extension.

Issue 6

The *localtime_r*( ) function is marked as part of the Thread-Safe Functions option.

Extensions beyond the ISO C standard are now marked.

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.

The **restrict** keyword is added to the *localtime_r*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

23553 **NAME**

23554 lockf — record locking on files

23555 **SYNOPSIS**

23556 XSI
```
#include <unistd.h>

int lockf(int fildes, int function, off_t size);
```
23557

23558

23559 **DESCRIPTION**

23560 The *lockf*( ) function allows sections of a file to be locked with advisory-mode locks. Calls to
23561 *lockf*( ) from other threads which attempt to lock the locked file section shall either return an
23562 error value or block until the section becomes unlocked. All the locks for a process are removed
23563 when the process terminates. Record locking with *lockf*( ) is supported for regular files and may
23564 be supported for other files.

23565 The *fildes* argument is an open file descriptor. The application shall ensure that the file descriptor
23566 has been opened with write-only permission (O_WRONLY) or with read/write permission
23567 (O_RDWR) to establish a lock with this function.

23568 The *function* argument is a control value which specifies the action to be taken. The permissible
23569 values for *function* are defined in **<unistd.h>** as follows:

23570

23571

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.

23575

23576 F_TEST detects if a lock by another process is present on the specified section.

23577 F_LOCK and F_TLOCK both lock a section of a file if the section is available.

23578 F_ULOCK removes locks from a section of the file.

23579 The *size* argument is the number of contiguous bytes to be locked or unlocked. The section to be
23580 locked or unlocked starts at the current offset in the file and extends forward for a positive size
23581 or backward for a negative size (the preceding bytes up to but not including the current offset).
23582 If *size* is 0, the section from the current offset through the largest possible file offset is locked
23583 (that is, from the current offset through the present or any future end-of-file). An area need not
23584 be allocated to the file to be locked because locks may exist past the end-of-file.

23585 The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained
23586 by a previously locked section for the same process. When this occurs, or if adjacent locked
23587 sections would occur, the sections are combined into a single locked section. If the request
23588 would cause the number of locks to exceed a system-imposed limit, the request shall fail.

23589 F_LOCK and F_TLOCK requests differ only by the action taken if the section is not available.
23590 F_LOCK blocks the calling thread until the section is available. F_TLOCK makes the function
23591 fail if the section is already locked by another process.

23592 File locks are released on first close by the locking process of any file descriptor for the file.

23593 F_ULOCK requests may release (wholly or in part) one or more locked sections controlled by the
23594 process. Locked sections shall be unlocked starting at the current file offset through *size* bytes or
23595 to the end-of-file if *size* is (**off_t**)0. When all of a locked section is not released (that is, when the
23596 beginning or end of the area to be unlocked falls within a locked section), the remaining portions
23597 of that section are still locked by the process. Releasing the center portion of a locked section

shall cause the remaining locked beginning and end portions to become two separate locked sections. If the request would cause the number of locks in the system to exceed a system-imposed limit, the request shall fail.

A potential for deadlock occurs if the threads of a process controlling a locked section are blocked by accessing another process' locked section. If the system detects that deadlock would occur, *lockf*( ) shall fail with an [EDEADLK] error.

The interaction between *fcntl*( ) and *lockf*( ) locks is unspecified.

Blocking on a section is interrupted by any signal.

An F_ULOCK request in which *size* is non-zero and the offset of the last byte of the requested section is the maximum value for an object of type **off_t**, when the process has an existing lock in which *size* is 0 and which includes the last byte of the requested section, shall be treated as a request to unlock from the start of the requested section with a size equal to 0. Otherwise, an F_ULOCK request shall attempt to unlock only the requested section.

Attempting to lock a section of a file that is associated with a buffered stream produces unspecified results.

RETURN VALUE

Upon successful completion, *lockf*( ) shall return 0. Otherwise, it shall return −1, set *errno* to indicate an error, and existing locks shall not be changed.

ERRORS

The *lockf*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid open file descriptor; or *function* is F_LOCK or F_TLOCK and *fildes* is not a valid file descriptor open for writing.

[EACCES] or [EAGAIN]
The *function* argument is F_TLOCK or F_TEST and the section is already locked by another process.

[EDEADLK] The *function* argument is F_LOCK and a deadlock is detected.

[EINTR] A signal was caught during execution of the function.

[EINVAL] The *function* argument is not one of F_LOCK, F_TLOCK, F_TEST, or F_ULOCK; or *size* plus the current file offset is less than 0.

[EOVERFLOW] The offset of the first, or if *size* is not 0 then the last, byte in the requested section cannot be represented correctly in an object of type **off_t**.

The *lockf*( ) function may fail if:

[EAGAIN] The *function* argument is F_LOCK or F_TLOCK and the file is mapped with *mmap*( ).

[EDEADLK] or [ENOLCK]
The *function* argument is F_LOCK, F_TLOCK, or F_ULOCK, and the request would cause the number of locks to exceed a system-imposed limit.

[EOPNOTSUPP] or [EINVAL]
The implementation does not support the locking of files of the type indicated by the *fildes* argument.

EXAMPLES

Locking a Portion of a File

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.

```
#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);
```

APPLICATION USAGE

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.

The *alarm*() function may be used to provide a timeout facility in applications requiring it.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm(), *chmod*(), *close*(), *creat*(), *fcntl*(), *fopen*(), *mmap*(), *open*(), *read*(), *write*(), 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.

Large File Summit extensions are added. In particular, the description of [EINVAL] is clarified and moved from optional to mandatory status.

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.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

23674 **NAME**

23675 log, logf, logl — natural logarithm function

23676 **SYNOPSIS**

23677
```
#include <math.h>
```

23678
```
double log(double x);
```
23679
```
float logf(float x);
```
23680
```
long double logl(long double x);
```

23681 **DESCRIPTION**

23682 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23683 conflict between the requirements described here and the ISO C standard is unintentional. This
23684 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

23685 These functions shall compute the natural logarithm of x, $\log_e(x)$. The application shall ensure
23686 that the value of x is positive.

23687 An application wishing to check for error situations should set *errno* to 0 before calling *log*( ). If
23688 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

23689 **RETURN VALUE**

23690 Upon successful completion, these functions shall return the natural logarithm of x.

23691 XSI If x is NaN, NaN shall be returned and *errno* may be set to [EDOM].

23692 XSI If x is less than 0, −HUGE_VAL or NaN shall be returned, and *errno* shall be set to [EDOM].

23693 If x is 0, −HUGE_VAL shall be returned and *errno* may be set to [ERANGE].

23694 **ERRORS**

23695 These functions shall fail if:

23696 [EDOM] The value of x is negative.

23697 These functions may fail if:

23698 XSI [EDOM] The value of x is NaN.

23699 [ERANGE] The value of x is 0.

23700 XSI No other errors shall occur.

23701 **EXAMPLES**

23702 None.

23703 **APPLICATION USAGE**

23704 None.

23705 **RATIONALE**

23706 None.

23707 **FUTURE DIRECTIONS**

23708 None.

23709 **SEE ALSO**

23710 *exp*( ), *isnan*( ), *log10*( ), *log1p*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

23711 **CHANGE HISTORY**

23712 First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 4

References to *matherr*( ) are removed.

The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the ISO C standard and to rationalize error handling in the mathematics functions.

The return value specified for [EDOM] is marked as an extension.

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 DESCRIPTION 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. |

NAME

log10, log10f, log10l — base 10 logarithm function

SYNOPSIS

```
#include <math.h>

double log10(double x);
float log10f(float x);
long double log10l(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 base 10 logarithm of x, $\log_{10}(x)$. The application shall ensure that the value of x is positive.

An application wishing to check for error situations should set *errno* to 0 before calling *log10*( ). If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the base 10 logarithm of x.

XSI If x is NaN, NaN shall be returned and *errno* may be set to [EDOM].

XSI If x is less than 0, −HUGE_VAL or NaN shall be returned, and *errno* shall be set to [EDOM].

If x is 0, −HUGE_VAL shall be returned and *errno* may be set to [ERANGE].

ERRORS

These functions shall fail if:

[EDOM] The value of x is negative.

These functions may fail if:

XSI [EDOM] The value of x is NaN.

[ERANGE] The value of x is 0.

XSI No other errors shall occur.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isnan( ), *log*( ), *pow*( ), 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 4

References to *matherr*( ) are removed.

The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the ISO C standard and to rationalize error handling in the mathematics functions.

The return value specified for [EDOM] is marked as an extension.

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 DESCRIPTION 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.

NAME

log1p, log1pf, log1pl — compute a natural logarithm

SYNOPSIS

```
#include <math.h>

double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);
```

DESCRIPTION

These functions shall compute $\log_e(1.0 + x)$. The application shall ensure that the value of *x* is greater than –1.0.

RETURN VALUE

Upon successful completion, these functions shall return the natural logarithm of 1.0 + *x*.

If *x* is NaN, *log1p*( ) shall return NaN and may set *errno* to [EDOM].

If *x* is less than –1.0, *log1p*( ) shall return –HUGE_VAL or NaN and set *errno* to [EDOM].

If *x* is –1.0, *log1p*( ) shall return –HUGE_VAL and may set *errno* to [ERANGE].

ERRORS

These functions shall fail if:

[EDOM] The value of *x* is less than –1.0.

These functions may fail and set *errno* to:

[EDOM] The value of *x* is NaN.

[ERANGE] The value of *x* is –1.0.

No other errors shall occur.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

log( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The *log1pf*( ) and *log1pl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

23815 **NAME**

23816 log2, log2f, log2l — compute base 2 logarithm functions

23817 **SYNOPSIS**

23818 `#include <math.h>`

23819 `double log2(double x);`
23820 `float log2f(float x);`
23821 `long double log2l(long double x);`

23822 **DESCRIPTION**

23823 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23824 conflict between the requirements described here and the ISO C standard is unintentional. This
23825 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

23826 These functions shall compute the base 2 logarithm of x, $\log_2(x)$. The application shall ensure
23827 that the value of x is positive.

23828 An application wishing to check for error situations should set *errno* to 0 before calling these
23829 functions. If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

23830 **RETURN VALUE**

23831 Upon successful completion, these functions shall return the base 2 logarithm of x.

23832 If x is NaN, these functions shall return NaN and may set *errno* to [EDOM].

23833 If x is less than 0, these functions shall return –HUGE_VAL or NaN and set *errno* to [EDOM].

23834 If x is 0, these functions shall return –HUGE_VAL and may set *errno* to [ERANGE].

23835 **ERRORS**

23836 These functions shall fail if:

23837 [EDOM] The value of x is less than 0.

23838 These functions may fail if:

23839 [EDOM] The value of x is NaN.

23840 [ERANGE] The value of x is 0.

23841 No other errors shall occur.

23842 **EXAMPLES**

23843 None.

23844 **APPLICATION USAGE**

23845 None.

23846 **RATIONALE**

23847 None.

23848 **FUTURE DIRECTIONS**

23849 None.

23850 **SEE ALSO**

23851 *log*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

23852 **CHANGE HISTORY**

23853 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

logb, logbf, logbl — radix-independent exponent

SYNOPSIS

XSI
```
#include <math.h>

double logb(double x);
float logbf(float x);
long double logbl(long double x);
```

DESCRIPTION

These functions shall compute the exponent of *x*, which is the integral part of $\log_r |x|$, as a signed floating point 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 the **<float.h>** header.

RETURN VALUE

Upon successful completion, these functions shall return the exponent of *x*.

If *x* is 0.0, *logb*( ) shall return –HUGE_VAL and set *errno* to [EDOM].

If *x* is ±Inf, *logb*( ) shall return +Inf.

If *x* is NaN, *logb*( ) shall return NaN and may set *errno* to [EDOM].

ERRORS

These functions shall fail if:

[EDOM] The *x* argument is 0.0.

These functions may fail if:

[EDOM] The *x* argument is NaN.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ilogb( ), *scalb*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<float.h>** **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The *logbf*( ) and *logbl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

23892 **NAME**

23893 longjmp — non-local goto

23894 **SYNOPSIS**

23895 `#include <setjmp.h>`

23896 `void longjmp(jmp_buf env, int val);`

23897 **DESCRIPTION**

23898 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23899 conflict between the requirements described here and the ISO C standard is unintentional. This
23900 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

23901 The *longjmp*( ) function shall restore the environment saved by the most recent invocation of
23902 *setjmp*( ) in the same thread, with the corresponding **jmp_buf** argument. If there is no such
23903 invocation, or if the function containing the invocation of the *setjmp*( ) macro has terminated
23904 execution in the interim, or if the invocation of *setjmp*( ) was within the scope of an identifier
23905 with variably modified type and execution has left that scope in the interim, the behavior is
23906 CX undefined. It is unspecified whether *longjmp*( ) restores the signal mask, leaves the signal mask
23907 unchanged, or restores it to its value at the time *setjmp*( ) was called.

23908 All accessible objects have values as of the time *longjmp*( ) was called, and all other components
23909 of the abstract machine have state (for example, floating-point status flags and open files),
23910 except that the values of objects of automatic storage duration are indeterminate if they meet all
23911 the following conditions:

23912 • They are local to the function containing the corresponding *setjmp*( ) invocation.

23913 • They do not have volatile-qualified type.

23914 • They are changed between the *setjmp*( ) invocation and *longjmp*( ) call.

23915 As it bypasses the usual function call and return mechanisms, *longjmp*( ) shall execute correctly
23916 in contexts of interrupts, signals, and any of their associated functions. However, if *longjmp*( ) is
23917 invoked from a nested signal handler (that is, from a function invoked as a result of a signal
23918 raised during the handling of another signal), the behavior is undefined.

23919 CX The effect of a call to *longjmp*( ) where initialization of the **jmp_buf** structure was not performed
23920 in the calling thread is undefined.

23921 **RETURN VALUE**

23922 After *longjmp*( ) is completed, program execution continues as if the corresponding invocation of
23923 *setjmp*( ) had just returned the value specified by *val*. The *longjmp*( ) function shall not cause
23924 *setjmp*( ) to return 0; if *val* is 0, *setjmp*( ) shall return 1.

23925 **ERRORS**

23926 No errors are defined.

23927 **EXAMPLES**

23928 None.

23929 **APPLICATION USAGE**

23930 Applications whose behavior depends on the value of the signal mask should not use *longjmp*( )
23931 and *setjmp*( ), since their effect on the signal mask is unspecified, but should instead use the
23932 *siglongjmp*( ) and *sigsetjmp*( ) functions (which can save and restore the signal mask under
23933 application control).

23934 **RATIONALE**

23935 None.

23936 **FUTURE DIRECTIONS**

23937 None.

23938 **SEE ALSO**

23939 *setjmp*( ), *sigaction*( ), *siglongjmp*( ), *sigsetjmp*( ), the Base Definitions volume of |
23940 IEEE Std. 1003.1-200x, **<setjmp.h>** |

23941 **CHANGE HISTORY**

23942 First released in Issue 1. Derived from Issue 1 of the SVID. |

23943 **Issue 4**

23944 The APPLICATION USAGE section is deleted.

23945 The following change is incorporated for alignment with the ISO C standard:

23946 • Mention of volatile-qualified types is added to the DESCRIPTION.

23947 **Issue 4, Version 2**

23948 The DESCRIPTION is updated for X/OPEN UNIX conformance and discusses valid possibilities
23949 for the resulting state of the signal mask.

23950 **Issue 5**

23951 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

23952 **Issue 6**

23953 Extensions beyond the ISO C standard are now marked.

23954 The following new requirements on POSIX implementations derive from alignment with the
23955 Single UNIX Specification:

23956 • The DESCRIPTION now explicitly makes *longjmp*( )'s effect on the signal mask unspecified.

23957 The DESCRIPTION is updated for alignment with the ISO/IEC 9899: 1999 standard. |

23958 **NAME**

23959 lrand48 — generate uniformly distributed pseudo-random non-negative long integers

23960 **SYNOPSIS**

23961 XSI
```
#include <stdlib.h>

long lrand48(void);
```
23962 |

23963 |

23964 **DESCRIPTION**

23965 Refer to *drand48*( ).

NAME

lsearch, lfind — linear search and update

SYNOPSIS

XSI
```
#include <search.h>

void *lsearch(const void *key, void *base, size_t *nelp, size_t width,
    int (*compar)(const void *, const void *));
void *lfind(const void *key, const void *base, size_t *nelp,
    size_t width, int (*compar)(const void *, const void *));
```

DESCRIPTION

The *lsearch*( ) function is a linear search routine. It returns a pointer into the table for the matching entry. If the entry does not occur, it is added at the end of the table. The *key* argument points to the entry to be sought in the table. The *base* argument points to the first element in the table. The *width* argument is the size of an element in bytes. The *nelp* argument points to an integer containing the current number of elements in the table. The integer to which *nelp* points is incremented if the entry is added to the table. The *compar* argument points to a comparison function which the application shall supply (for example, *strcmp*( )). It is called with two arguments that point to the elements being compared. The application shall ensure that the function returns 0 if the elements are equal, and non-zero otherwise.

The *lfind*( ) function is the same as *lsearch*( ) except that if the entry is not found, it is not added to the table. Instead, a null pointer is returned.

RETURN VALUE

If the searched for entry is found, both *lsearch*( ) and *lfind*( ) shall return a pointer to it. Otherwise, *lfind*( ) shall return a null pointer and *lsearch*( ) shall return a pointer to the newly added element.

Both functions shall return a null pointer in case of error.

ERRORS

No errors are defined.

EXAMPLES

Storing Strings in a Table

This fragment reads in less than or equal to TABSIZE strings of length less than or equal to ELSIZE and stores them in a table, eliminating duplicates.

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

#define TABSIZE 50
#define ELSIZE 120

...
    char line[ELSIZE], tab[TABSIZE][ELSIZE];
    size_t nel = 0;
    ...
    while (fgets(line, ELSIZE, stdin) != NULL && nel < TABSIZE)
        (void) lsearch(line, tab, &nel,
            ELSIZE, (int (*)(const void *, const void *)) strcmp);
    ...
```

Finding a Matching Entry

The following example finds any line that reads "This is a test.".

```
#include <search.h>
#include <string.h>
...
char line[ELSIZE], tab[TABSIZE][ELSIZE];
size_t nel = 0;
char *findline;
void *entry;

findline = "This is a test.\n";

entry = lfind(findline, tab, &nel, ELSIZE, (
    int (*)(const void *, const void *)) strcmp);
```

APPLICATION USAGE

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Undefined results can occur if there is not enough room in the table to add a new item.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

hcreate( ), *tsearch*( ), 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 4

In the SYNOPSIS section, the type of argument *key* in the declaration of *lsearch*( ) is changed from **void*** to **const void***, the type arguments *key* and *base* have been changed from **void*** to **const void*** in the declaration of *lfind*( ), and the arguments to *compar* are defined for both functions.

In the EXAMPLES section, the sample code is updated to use ISO C standard syntax.

Warnings about the casting of various arguments are removed from the APPLICATION USAGE section, as casting requirements are now clear from the function definitions.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

lseek — move the read/write file offset

SYNOPSIS

```
#include <unistd.h>

off_t lseek(int fildes, off_t offset, int whence);
```

DESCRIPTION

The *lseek*( ) function shall set the file offset for the open file description associated with the file descriptor *fildes*, as follows:

- If *whence* is {SEEK_SET}, the file offset is set to *offset* bytes.
- If *whence* is {SEEK_CUR}, the file offset is set to its current location plus *offset*.
- If *whence* is {SEEK_END}, the file offset is set to the size of the file plus *offset*.

The symbolic constants {SEEK_SET}, {SEEK_CUR}, and {SEEK_END} are defined in **<unistd.h>**.

The behavior of *lseek*( ) on devices which are incapable of seeking is implementation-defined. The value of the file offset associated with such a device is undefined.

The *lseek*( ) function shall allow the file offset to be set beyond the end of the existing data in the file. If data is later written at this point, subsequent reads of data in the gap shall return bytes with the value 0 until data is actually written into the gap.

The *lseek*( ) function shall not, by itself, extend the size of a file.

SHM If *fildes* refers to a shared memory object, the result of the *lseek*( ) function is unspecified.

TYM If *fildes* refers to a typed memory object, the result of the *lseek*( ) function is unspecified.

RETURN VALUE

Upon successful completion, the resulting offset, as measured in bytes from the beginning of the file, shall be returned. Otherwise, (**off_t**)–1 shall be returned, *errno* shall be set to indicate the error, and the file offset shall remain unchanged.

ERRORS

The *lseek*( ) function shall fail if:

[EBADF]	The *fildes* argument is not an open file descriptor.
[EINVAL]	The *whence* argument is not a proper value, or the resulting file offset would be negative for a regular file, block special file, or directory.
[EOVERFLOW]	The resulting file offset would be a value which cannot be represented correctly in an object of type **off_t**.
[ESPIPE]	The *fildes* argument is associated with a pipe, FIFO, or socket.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The ISO C standard includes the functions *fgetpos*( ) and *fsetpos*( ), which work on very large files by use of a special positioning type.

Although *lseek*( ) may position the file offset beyond the end of the file, this function does not itself extend the size of the file. While the only function in this volume of IEEE Std. 1003.1-200x

that may directly extend the size of the file is *write*( ), several functions originally derived from the ISO C standard, such as *fwrite*( ), *fprintf*( ), and so on, may do so (by causing calls on *write*( )).

An invalid file offset that would cause [EINVAL] to be returned may be both implementation-defined and device-dependent (for example, memory may have few invalid values). A negative file offset may be valid for some devices in some implementations.

The POSIX.1-1990 standard did not specifically prohibit *lseek*( ) from returning a negative offset. Therefore, an application was required to clear *errno* prior to the call and check *errno* upon return to determine whether a return value of (**off_t**)–1 is a negative offset or an indication of an error condition. The standard developers did not wish to require this action on the part of a portable application, and chose to require that *errno* be set to [EINVAL] when the resulting file offset would be negative for a regular file, block special file, or directory.

FUTURE DIRECTIONS

None.

SEE ALSO

open( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

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

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

The APPLICATION USAGE section is removed, as the ISO POSIX-1 standard now requires that **off_t** be signed.

Issue 5

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

Large File Summit extensions are added.

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.
- The [EOVERFLOW] error condition is added. This change is to support large files.

An additional [ESPIPE] error condition is added for sockets.

The DESCRIPTION is updated for alignment with IEEE Std. 1003.1j-2000 by specifying that *lseek*( ) results are unspecified for typed memory objects.

24120 **NAME**

24121 lstat — get symbolic link status

24122 **SYNOPSIS**

24123 `#include <sys/stat.h>`

24124 `int lstat(const char *restrict path, struct stat *restrict buf);` |

24125 **DESCRIPTION** |

24126 The *lstat*( ) function shall have the same effect as *stat*( ), except when *path* refers to a symbolic
24127 link. In that case *lstat*( ) shall return information about the link, while *stat*( ) shall return
24128 information about the file the link references.

24129 For symbolic links, the *st_mode* member shall contain meaningful information when used with
24130 the file type macros, and the *st_size* member shall contain the length of the path name contained
24131 in the symbolic link. File mode bits and the contents of the remaining members of the **stat**
24132 structure are unspecified. The value returned in the *st_size* member is the length of the contents
24133 of the symbolic link, and does not count any trailing null.

24134 **RETURN VALUE**

24135 Upon successful completion, *lstat*( ) shall return 0. Otherwise, it shall return –1 and set *errno* to
24136 indicate the error.

24137 **ERRORS**

24138 The *lstat*( ) function shall fail if:

24139 [EACCES] A component of the path prefix denies search permission.

24140 [EIO] An error occurred while reading from the file system.

24141 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
24142 argument.

24143 [ENAMETOOLONG] |
24144 The length of a path name exceeds {PATH_MAX} or a path name component |
24145 is longer than {NAME_MAX}. |

24146 [ENOTDIR] A component of the path prefix is not a directory. |

24147 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string. |

24148 [EOVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file |
24149 serial number cannot be represented correctly in the structure pointed to by
24150 *buf*.

24151 The *lstat*( ) function may fail if:

24152 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
24153 resolution of the *path* argument.

24154 [ENAMETOOLONG] |
24155 As a result of encountering a symbolic link in resolution of the *path* argument,
24156 the length of the substituted path name string exceeded {PATH_MAX}.

24157 [EOVERFLOW] One of the members is too large to store into the structure pointed to by the |
24158 *buf* argument.

EXAMPLES

Obtaining Symbolic Link Status Information

The following example shows how to obtain status information for a symbolic link named **/modules/pass1**. The structure variable *buffer* is defined for the **stat** structure. If the *path* argument specified the file name for the file pointed to by the symbolic link (**/home/cnd/mod1**), the results of calling the function would be the same as those returned by a call to the *stat*() function.

```
#include <sys/stat.h>

struct stat buffer;
int status;
...
status = lstat("/modules/pass1", &buffer);
```

APPLICATION USAGE

None.

RATIONALE

The *lstat*() function is not required to update the time-related fields if the named file is not a symbolic link. While the *st_uid*, *st_gid*, *st_atime*, *st_mtime*, and *st_ctime* members of the **stat** structure may apply to a symbolic link, they are not required to do so. No functions in IEEE Std. 1003.1-200x are required to maintain any of these time fields.

FUTURE DIRECTIONS

None.

SEE ALSO

fstat(), *readlink*(), *stat*(), *symlink*(), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/stat.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Large File Summit extensions are added.

Issue 6

The following changes were made to align with the IEEE P1003.1a draft standard:

- This function is now mandatory.
- The [ELOOP] optional error condition is added.

The **restrict** keyword is added to the *lstat*() prototype for alignment with the ISO/IEC 9899: 1999 standard.

NAME

makecontext, swapcontext — manipulate user contexts

SYNOPSIS

XSI

```
#include <ucontext.h>

void makecontext(ucontext_t *ucp, void (*func)(void),
    int argc, ...);
int swapcontext(ucontext_t *restrict oucp,
    const ucontext_t *restrict ucp);
```

DESCRIPTION

The *makecontext*( ) function shall modify the context specified by *ucp*, which has been initialized using *getcontext*( ). When this context is resumed using *swapcontext*( ) or *setcontext*( ), program execution shall continue by calling *func*, passing it the arguments that follow *argc* in the *makecontext*( ) call.

Before a call is made to *makecontext*( ), the application shall ensure that the context being modified has a stack allocated for it. The application shall ensure that the value of *argc* matches the number of integer arguments passed to *func*; otherwise, the behavior is undefined.

The *uc_link* member is used to determine the context that shall be resumed when the context being modified by *makecontext*( ) returns. The application shall ensure that the *uc_link* member is initialized prior to the call to *makecontext*( ).

The *swapcontext*( ) function shall save the current context in the context structure pointed to by *oucp* and shall set the context to the context structure pointed to by *ucp*.

RETURN VALUE

Upon successful completion, *swapcontext*( ) shall return 0. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *swapcontext*( ) function shall fail if:

[ENOMEM] The *ucp* argument does not have enough stack left to complete the operation.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exit( ), *getcontext*( ), *sigaction*( ), *sigprocmask*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ucontext.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 description of [ENOMEM] is changed to apply to *swapcontext*( ) only.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *swapcontext*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

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 indeterminate.

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 either a null pointer or a unique pointer.

RETURN VALUE

Upon successful completion with *size* not equal to 0, *malloc*( ) shall return a pointer to the allocated space. If *size* is 0, either a null pointer or a unique pointer that can be successfully CX passed to *free*( ) shall be returned. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

ERRORS

The *malloc*( ) function shall fail if:

CX [ENOMEM] Insufficient storage space is available.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

calloc( ), *free*( ), *realloc*( ), 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 4

The setting of *errno* and the [ENOMEM] error are marked as extensions.

The APPLICATION USAGE section is changed to record that **<malloc.h>** need no longer be supported on XSI-conformant systems.

The following change is incorporated for alignment with the ISO C standard:

- The RETURN VALUE section is updated to indicate what is returned if *size* is 0.

Issue 6

Extensions beyond the ISO C standard are now 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.

NAME

mblen — get number of bytes in a character

SYNOPSIS

```
#include <stdlib.h>

int mblen(const char *s, 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.

If *s* is not a null pointer, *mblen*( ) determines the number of bytes constituting the character pointed to by *s*. Except that the shift state of *mbtowc*( ) is not affected, it is equivalent to:

```
mbtowc((wchar_t *)0, s, n);
```

The implementation shall behave as if no function defined in this volume of IEEE Std. 1003.1-200x calls *mblen*( ).

The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a state-dependent encoding, this function is placed into its initial state by a call for which its character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null pointer cause the internal state of the function to be altered as necessary. A call with *s* as a null pointer causes this function to return a non-zero value if encodings have state dependency, and 0 otherwise. If the implementation employs special bytes to change the shift state, these bytes do not produce separate wide-character codes, but are grouped with an adjacent character. Changing the *LC_CTYPE* category causes the shift state of this function to be indeterminate.

RETURN VALUE

If *s* is a null pointer, *mblen*( ) shall return a non-zero or 0 value, if character encodings, respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *mblen*( ) shall either return 0 (if *s* points to the null byte), or return the number of bytes that constitute the character (if the next *n* or fewer bytes form a valid character), or return –1 (if they do not form a valid character) CX and may set *errno* to indicate the error. In no case shall the value returned be greater than *n* or the value of the {MB_CUR_MAX} macro.

ERRORS

The *mblen*( ) function may fail if:

XSI [EILSEQ] Invalid character sequence is detected. |

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbtowc( ), *mbstowcs*( ), *wctomb*( ), *wcstombs*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>** |

24336 **CHANGE HISTORY**

24337 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 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.

If *s* is not a null pointer, *mbrlen*( ) shall determine the number of bytes constituting the character pointed to by *s*. It is equivalent to:

```
mbstate_t internal;
mbrtowc(NULL, s, n, ps != NULL ? ps : &internal);
```

If *ps* is a null pointer, the *mbrlen*( ) function uses its own internal **mbstate_t** object, which is initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by *ps* is used to completely describe the current conversion state of the associated character sequence. The implementation shall behave as if no function defined in this volume of IEEE Std. 1003.1-200x calls *mbrlen*( ).

XSI The behavior of this function is affected by the *LC_CTYPE* category of the current locale.

RETURN VALUE

The *mbrlen*( ) function shall return the first of the following that applies:

0	If the next *n* or fewer bytes complete the character that corresponds to the null wide character.
positive	If the next *n* or fewer bytes complete a valid character; the value returned shall be the number of bytes that complete the character.
(**size_t**)–2	If the next *n* bytes contribute to an incomplete but potentially valid character, and all *n* bytes have been processed. When *n* has at least the value of the {MB_CUR_MAX} macro, this case can only occur if *s* points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).
(**size_t**)–1	If an encoding error occurs, in which case the next *n* or fewer bytes do not contribute to a complete and valid character. In this case, [EILSEQ] shall be stored in *errno* and the conversion state is undefined.

ERRORS

The *mbrlen*( ) function may fail if:

[EINVAL]	*ps* points to an object that contains an invalid conversion state.
[EILSEQ]	Invalid character sequence is detected.

24376 **EXAMPLES**

24377 None.

24378 **APPLICATION USAGE**

24379 None.

24380 **RATIONALE**

24381 None.

24382 **FUTURE DIRECTIONS**

24383 None.

24384 **SEE ALSO**

24385 *mbsinit*( ), *mbrtowc*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>** |

24386 **CHANGE HISTORY**

24387 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
24388 (E). |

24389 **Issue 6** |

24390 The *mbrlen*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard. |

24391 **NAME**

24392 mbrtowc — convert a character to a wide-character code (restartable)

24393 **SYNOPSIS**

24394 ```
#include <wchar.h>
```

24395 ```
size_t mbrtowc(wchar_t *restrict pwc, const char *restrict s,
```
24396 ```
    size_t n, mbstate_t *restrict ps);
```

24397 **DESCRIPTION**

24398 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24399 conflict between the requirements described here and the ISO C standard is unintentional. This
24400 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

24401 If *s* is a null pointer, the *mbrtowc*( ) function shall be equivalent to the call:

24402 ```
mbrtowc(NULL, "", 1, ps)
```

24403 In this case, the values of the arguments *pwc* and *n* are ignored.

24404 If *s* is not a null pointer, the *mbrtowc*( ) function inspects at most *n* bytes beginning at the byte
24405 pointed to by *s* to determine the number of bytes needed to complete the next character
24406 (including any shift sequences). If the function determines that the next character is completed, it
24407 determines the value of the corresponding wide character and then, if *pwc* is not a null pointer,
24408 stores that value in the object pointed to by *pwc*. If the corresponding wide character is the null
24409 wide character, the resulting state described is the initial conversion state.

24410 If *ps* is a null pointer, the *mbrtowc*( ) function uses its own internal **mbstate_t** object, which is
24411 initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object
24412 pointed to by *ps* is used to completely describe the current conversion state of the associated
24413 character sequence. The implementation shall behave as if no function defined in this volume of
24414 IEEE Std. 1003.1-200x calls *mbrtowc*( ).

24415 XSI The behavior of this function is affected by the *LC_CTYPE* category of the current locale.

24416 **RETURN VALUE**

24417 The *mbrtowc*( ) function shall return the first of the following that applies:

24418 0 If the next *n* or fewer bytes complete the character that corresponds to the null
24419 wide character (which is the value stored).

24420 *positive* If the next *n* or fewer bytes complete a valid character (which is the value
24421 stored); the value returned shall be the number of bytes that complete the
24422 character.

24423 (**size_t**)–2 If the next *n* bytes contribute to an incomplete but potentially valid character,
24424 and all *n* bytes have been processed (no value is stored). When *n* has at least
24425 the value of the {MB_CUR_MAX} macro, this case can only occur if *s* points at
24426 a sequence of redundant shift sequences (for implementations with state-
24427 dependent encodings).

24428 (**size_t**)–1 If an encoding error occurs, in which case the next *n* or fewer bytes do not
24429 contribute to a complete and valid character (no value is stored). In this case,
24430 [EILSEQ] shall be stored in *errno* and the conversion state is undefined.

24431 **ERRORS**

24432 The *mbrtowc*( ) function may fail if:

24433 CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

[EILSEQ] Invalid character sequence is detected.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbsinit( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The *mbrtowc*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

mbsinit — determine conversion object status

SYNOPSIS

```
#include <wchar.h>

int mbsinit(const mbstate_t *ps);
```

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.

If *ps* is not a null pointer, the *mbsinit*( ) function shall determine whether the object pointed to by *ps* describes an initial conversion state.

RETURN VALUE

The *mbsinit*( ) function shall return non-zero if *ps* is a null pointer, or if the pointed-to object describes an initial conversion state; otherwise, it shall return zero.

If an **mbstate_t** object is altered by any of the functions described as ''restartable'', and is then used with a different character sequence, or in the other conversion direction, or with a different *LC_CTYPE* category setting than on earlier function calls, the behavior is undefined.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The **mbstate_t** object is used to describe the current conversion state from a particular character sequence to a wide-character sequence (or *vice versa*) under the rules of a particular setting of the *LC_CTYPE* category of the current locale.

The initial conversion state corresponds, for a conversion in either direction, to the beginning of a new character sequence in the initial shift state. A zero valued **mbstate_t** object is at least one way to describe an initial conversion state. A zero valued **mbstate_t** object can be used to initiate conversion involving any character sequence, in any *LC_CTYPE* category setting.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbrlen( ), *mbrtowc*( ), *wcrtomb*( ), *mbsrtowcs*( ), *wcsrtombs*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

24489 **NAME**

24490 mbsrtowcs — convert a character string to a wide-character string (restartable)

24491 **SYNOPSIS**

24492 ```
#include <wchar.h>
```

24493 ```
size_t mbsrtowcs(wchar_t *restrict dst, const char **restrict src,
```

24494 ```
    size_t len, mbstate_t *restrict ps);
```

24495 **DESCRIPTION**

24496 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24497 conflict between the requirements described here and the ISO C standard is unintentional. This
24498 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

24499 The *mbsrtowcs*( ) function shall convert a sequence of characters, beginning in the conversion
24500 state described by the object pointed to by *ps*, from the array indirectly pointed to by *src* into a
24501 sequence of corresponding wide characters. If *dst* is not a null pointer, the converted characters
24502 are stored into the array pointed to by *dst*. Conversion continues up to and including a
24503 terminating null character, which is also stored. Conversion stops early in either of the following
24504 cases:

24505 - A sequence of bytes is encountered that does not form a valid character.

24506 - *len* codes have been stored into the array pointed to by *dst* (and *dst* is not a null pointer).

24507 Each conversion takes place as if by a call to the *mbrtowc*( ) function.

24508 If *dst* is not a null pointer, the pointer object pointed to by *src* is assigned either a null pointer (if
24509 conversion stopped due to reaching a terminating null character) or the address just past the last
24510 character converted (if any). If conversion stopped due to reaching a terminating null character,
24511 and if *dst* is not a null pointer, the resulting state described is the initial conversion state.

24512 If *ps* is a null pointer, the *mbsrtowcs*( ) function uses its own internal **mbstate_t** object, which is
24513 initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object
24514 pointed to by *ps* is used to completely describe the current conversion state of the associated
24515 character sequence. The implementation behaves as if no function defined in this volume of
24516 IEEE Std. 1003.1-200x calls *mbsrtowcs*( ).

24517 XSI The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

24518 **RETURN VALUE**

24519 If the input conversion encounters a sequence of bytes that do not form a valid character, an
24520 encoding error occurs. In this case, the *mbsrtowcs*( ) function stores the value of the macro
24521 [EILSEQ] in *errno* and shall return (**size_t**)–1; the conversion state is undefined. Otherwise, it
24522 shall return the number of characters successfully converted, not including the terminating null
24523 (if any).

24524 **ERRORS**

24525 The *mbsrtowcs*( ) function may fail if:

24526 [EINVAL] *ps* points to an object that contains an invalid conversion state.

24527 [EILSEQ] Invalid character sequence is detected.
```

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbsinit( ), *mbrtowc*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The *mbsrtowcs*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

24543 **NAME**

24544 mbstowcs — convert a character string to a wide-character string

24545 **SYNOPSIS**

24546
```
#include <stdlib.h>

size_t mbstowcs(wchar_t *restrict pwcs, const char *restrict s,
    size_t n);
```

24549 **DESCRIPTION**

24550 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24551 conflict between the requirements described here and the ISO C standard is unintentional. This
24552 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

24553 The *mbstowcs*( ) function shall convert a sequence of characters that begins in the initial shift
24554 state from the array pointed to by *s* into a sequence of corresponding wide-character codes and
24555 stores not more than *n* wide-character codes into the array pointed to by *pwcs*. No characters
24556 that follow a null byte (which is converted into a wide-character code with value 0) shall be
24557 examined or converted. Each character is converted as if by a call to *mbtowc*( ), except that the
24558 shift state of *mbtowc*( ) is not affected.

24559 No more than *n* elements shall be modified in the array pointed to by *pwcs*. If copying takes
24560 place between objects that overlap, the behavior is undefined.

24561 XSI The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale. If
24562 *pwcs* is a null pointer, *mbstowcs*( ) shall return the length required to convert the entire array
24563 regardless of the value of *n*, but no values are stored.

24564 **RETURN VALUE**

24565 CX If an invalid character is encountered, *mbstowcs*( ) shall return (**size_t**)–1 and may set *errno* to
24566 indicate the error. Otherwise, *mbstowcs*( ) shall return the number of the array elements modified
24567 (or required if *pwcs* is null), not including a terminating 0 code, if any. The array shall not be
24568 zero-terminated if the value returned is *n*.

24569 **ERRORS**

24570 The *mbstowcs*( ) function may fail if:

24571 XSI [EILSEQ] Invalid byte sequence is detected.

24572 **EXAMPLES**

24573 None.

24574 **APPLICATION USAGE**

24575 None.

24576 **RATIONALE**

24577 None.

24578 **FUTURE DIRECTIONS**

24579 None.

24580 **SEE ALSO**

24581 *mblen*( ), *mbtowc*( ), *wctomb*( ), *wcstombs*( ), the Base Definitions volume of IEEE Std. 1003.1-200x,
24582 **<stdlib.h>**

24583 **CHANGE HISTORY**

24584 First released in Issue 4. Aligned with the ISO C standard.

Issue 6

The *mbstowcs*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

mbtowc — convert a character to a wide-character code

SYNOPSIS

```
#include <stdlib.h>

int mbtowc(wchar_t *restrict pwc, const char *restrict s, 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.

If *s* is not a null pointer, *mbtowc*( ) shall determine the number of the bytes that constitute the character pointed to by *s*. It then determines the wide-character code for the value of type **wchar_t** that corresponds to that character. (The value of the wide-character code corresponding to the null byte is 0.) If the character is valid and *pwc* is not a null pointer, *mbtowc*( ) stores the wide-character code in the object pointed to by *pwc*.

The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a state-dependent encoding, this function is placed into its initial state by a call for which its character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null pointer cause the internal state of the function to be altered as necessary. A call with *s* as a null pointer causes this function to return a non-zero value if encodings have state dependency, and 0 otherwise. If the implementation employs special bytes to change the shift state, these bytes do not produce separate wide-character codes, but are grouped with an adjacent character. Changing the *LC_CTYPE* category causes the shift state of this function to be indeterminate. At most *n* bytes of the array pointed to by *s* shall be examined.

The implementation shall behave as if no function defined in this volume of IEEE Std. 1003.1-200x calls *mbtowc*( ).

RETURN VALUE

If *s* is a null pointer, *mbtowc*( ) shall return a non-zero or 0 value, if character encodings, respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *mbtowc*( ) shall either return 0 (if *s* points to the null byte), or return the number of bytes that constitute the CX converted character (if the next *n* or fewer bytes form a valid character), or return −1 and may set *errno* to indicate the error(if they do not form a valid character).

In no case shall the value returned be greater than *n* or the value of the {MB_CUR_MAX} macro.

ERRORS

The *mbtowc*( ) function may fail if:

XSI [EILSEQ] Invalid character sequence is detected.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mblen( ), *mbstowcs*( ), *wctomb*( ), *wcstombs*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4. Aligned with the ISO C standard.

Issue 6

The *mbtowc*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

memccpy — copy bytes in memory

SYNOPSIS

XSI
```
#include <string.h>

void *memccpy(void *restrict s1, const void *restrict s2,
    int c, size_t n);
```

DESCRIPTION

The *memccpy*( ) function shall copy bytes from memory area *s2* into *s1*, stopping after the first occurrence of byte *c* (converted to an **unsigned char**) is copied, or after *n* bytes are copied, whichever comes first. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *memccpy*( ) function shall return a pointer to the byte after the copy of *c* in *s1*, or a null pointer if *c* was not found in the first *n* bytes of *s2*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *memccpy*( ) function does not check for the overflow of the receiving memory area.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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 4

The type of argument *s2* is changed from **void*** to **const void***.

Reference to use of the **<memory.h>** header is removed from the APPLICATION USAGE section.

The FUTURE DIRECTIONS section is removed.

Issue 6

The **restrict** keyword is added to the *memccpy*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

memchr — find byte in memory

SYNOPSIS

```
#include <string.h>

void *memchr(const void *s, int c, 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 *memchr*( ) function shall locate the first occurrence of *c* (converted to an **unsigned char**) in the initial *n* bytes (each interpreted as **unsigned char**) of the object pointed to by *s*.

RETURN VALUE

The *memchr*( ) function shall return a pointer to the located byte, or a null pointer if the byte does not occur in the object.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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 4

The APPLICATION USAGE section is removed.

The following changes are incorporated for alignment with the ISO C standard:

- The function is no longer marked as an extension.
- The type of argument *s* is changed from **void*** to **const void***.

24707 **NAME**

24708 memcmp — compare bytes in memory

24709 **SYNOPSIS**

24710 `#include <string.h>`

24711 `int memcmp(const void *s1, const void *s2, size_t n);`

24712 **DESCRIPTION**

24713 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24714 conflict between the requirements described here and the ISO C standard is unintentional. This
24715 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

24716 The *memcmp*( ) function shall compare the first *n* bytes (each interpreted as **unsigned char**) of the
24717 object pointed to by *s1* to the first *n* bytes of the object pointed to by *s2*.

24718 The sign of a non-zero return value shall be determined by the sign of the difference between the
24719 values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the objects
24720 being compared.

24721 **RETURN VALUE**

24722 The *memcmp*( ) function shall return an integer greater than, equal to, or less than 0, if the object
24723 pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*, respectively.

24724 **ERRORS**

24725 No errors are defined.

24726 **EXAMPLES**

24727 None.

24728 **APPLICATION USAGE**

24729 None.

24730 **RATIONALE**

24731 None.

24732 **FUTURE DIRECTIONS**

24733 None.

24734 **SEE ALSO**

24735 The Base Definitions volume of IEEE Std. 1003.1-200x, **<string.h>**

24736 **CHANGE HISTORY**

24737 First released in Issue 1. Derived from Issue 1 of the SVID.

24738 **Issue 4**

24739 The RETURN VALUE section is clarified.

24740 The APPLICATION USAGE section is removed.

24741 The following changes are incorporated for alignment with the ISO C standard:

24742 - The function is no longer marked as an extension.

24743 - The type of arguments *s1* and *s2* are changed from **void*** to **const void***.

NAME

memcpy — copy bytes in memory

SYNOPSIS

```
#include <string.h>

void *memcpy(void *restrict s1, const void *restrict s2, 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 *memcpy*( ) function shall copy *n* bytes from the object pointed to by *s2* into the object pointed to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *memcpy*( ) function shall return *s1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *memcpy*( ) function does not check for the overflowing of the receiving memory area.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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 4

Reference to use of the **<memory.h>** header is removed from the APPLICATION USAGE section, and a note about overflow checking has been added.

The FUTURE DIRECTIONS section is removed.

The following changes are incorporated for alignment with the ISO C standard:

- The function is no longer marked as an extension.
- The type of argument *s2* is changed from **void*** to **const void***.

Issue 6

The *memcpy*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

24780 **NAME**

24781 memmove — copy bytes in memory with overlapping areas

24782 **SYNOPSIS**

24783 `#include <string.h>`

24784 `void *memmove(void *s1, const void *s2, size_t n);`

24785 **DESCRIPTION**

24786 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24787 conflict between the requirements described here and the ISO C standard is unintentional. This
24788 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

24789 The *memmove*( ) function shall copy *n* bytes from the object pointed to by *s2* into the object
24790 pointed to by *s1*. Copying takes place as if the *n* bytes from the object pointed to by *s2* are first
24791 copied into a temporary array of *n* bytes that does not overlap the objects pointed to by *s1* and
24792 *s2*, and then the *n* bytes from the temporary array are copied into the object pointed to by *s1*.

24793 **RETURN VALUE**

24794 The *memmove*( ) function shall return *s1*; no return value is reserved to indicate an error.

24795 **ERRORS**

24796 No errors are defined.

24797 **EXAMPLES**

24798 None.

24799 **APPLICATION USAGE**

24800 None.

24801 **RATIONALE**

24802 None.

24803 **FUTURE DIRECTIONS**

24804 None.

24805 **SEE ALSO**

24806 The Base Definitions volume of IEEE Std. 1003.1-200x, **<string.h>**

24807 **CHANGE HISTORY**

24808 First released in Issue 4. Derived from the ANSI C standard.

NAME

memset — set bytes in memory

SYNOPSIS

```
#include <string.h>

void *memset(void *s, int c, 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 *memset*( ) function shall copy *c* (converted to an **unsigned char**) into each of the first *n* bytes of the object pointed to by *s*.

RETURN VALUE

The *memset*( ) function shall return *s*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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 4

The APPLICATION USAGE section is removed.

The following change is incorporated for alignment with the ISO C standard:

- The function is no longer marked as an extension.

24840 **NAME**

24841 mkdir — make a directory

24842 **SYNOPSIS**

24843 `#include <sys/stat.h>`

24844 `int mkdir(const char *`*`path`*`, mode_t `*`mode`*`);`

24845 **DESCRIPTION**

24846 The *mkdir*( ) function creates a new directory with name *path*. The file permission bits of the new
24847 directory are initialized from *mode*. These file permission bits of the *mode* argument are modified
24848 by the process' file creation mask.

24849 When bits in *mode* other than the file permission bits are set, the meaning of these additional bits |
24850 is implementation-defined. |

24851 The directory's user ID is set to the process' effective user ID. The directory's group ID shall be
24852 set to the group ID of the parent directory or to the effective group ID of the process.

24853 The newly created directory shall be an empty directory.

24854 If *path* names a symbolic link, *mkdir*( ) shall fail and set *errno* to [EEXIST].

24855 Upon successful completion, *mkdir*( ) shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
24856 fields of the directory. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the
24857 new entry shall be marked for update.

24858 **RETURN VALUE**

24859 Upon successful completion, *mkdir*( ) shall return 0. Otherwise, –1 shall be returned, no directory
24860 shall be created, and *errno* shall be set to indicate the error.

24861 **ERRORS**

24862 The *mkdir*( ) function shall fail if:

24863 [EACCES] Search permission is denied on a component of the path prefix, or write |
24864 permission is denied on the parent directory of the directory to be created.

24865 [EEXIST] The named file exists. |

24866 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* |
24867 argument. |

24868 [EMLINK] The link count of the parent directory would exceed {LINK_MAX}. |

24869 [ENAMETOOLONG] |
24870 The length of the *path* argument exceeds {PATH_MAX} or a path name
24871 component is longer than {NAME_MAX}. |

24872 [ENOENT] A component of the path prefix specified by *path* does not name an existing |
24873 directory or *path* is an empty string.

24874 [ENOSPC] The file system does not contain enough space to hold the contents of the new |
24875 directory or to extend the parent directory of the new directory.

24876 [ENOTDIR] A component of the path prefix is not a directory. |

24877 [EROFS] The parent directory resides on a read-only file system. |

24878 The *mkdir*( ) function may fail if:

24879 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
24880 resolution of the *path* argument. |

[ENAMETOOLONG] |
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted path name string exceeded {PATH_MAX}. |

EXAMPLES

Creating a Directory

The following example shows how to create a directory named **/home/cnd/mod1**, with read/write/search permissions for owner and group, and with read/search permissions for others.

```
#include <sys/types.h>
#include <sys/stat.h>

int status;
...
status = mkdir("/home/cnd/mod1", S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
```

APPLICATION USAGE

None.

RATIONALE

The *mkdir*( ) function originated in 4.2 BSD and was added to System V in Release 3.0.

4.3 BSD detects [ENAMETOOLONG]. |

See *getgroups*( ) about the group of a newly created directory.

FUTURE DIRECTIONS

None.

SEE ALSO

umask( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/stat.h>**, **<sys/types.h>** |

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument *path* is changed from **char*** to **const char***.

The following changes are incorporated for alignment with the FIPS requirements:

- In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path name component is larger than {NAME_MAX} is now defined as mandatory and marked as an extension.

Issue 4, Version 2

The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- It states that [ELOOP] is returned if too many symbolic links are encountered during path name resolution.
- A second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of path name resolution of a symbolic link.

Issue 6

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

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 [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.

24936 **NAME**

24937 mkfifo — make a FIFO special file

24938 **SYNOPSIS**

24939 `#include <sys/stat.h>`

24940 `int mkfifo(const char *`*`path`*`, mode_t `*`mode`*`);`

24941 **DESCRIPTION**

24942 The *mkfifo*( ) function shall create a new FIFO special file named by the path name pointed to by
24943 *path*. The file permission bits of the new FIFO are initialized from *mode*. The file permission bits
24944 of the *mode* argument are modified by the process' file creation mask.

24945 When bits in *mode* other than the file permission bits are set, the effect is implementation-
24946 defined.

24947 If *path* names a symbolic link, *mkfifo*( ) shall fail and set *errno* to [EEXIST].

24948 The FIFO's user ID shall be set to the process' effective user ID. The FIFO's group ID shall be set
24949 to the group ID of the parent directory or to the effective group ID of the process.

24950 Upon successful completion, *mkfifo*( ) shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
24951 fields of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new
24952 entry shall be marked for update.

24953 **RETURN VALUE**

24954 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned, no FIFO shall
24955 be created, and *errno* shall be set to indicate the error.

24956 **ERRORS**

24957 The *mkfifo*( ) function shall fail if:

24958 [EACCES] A component of the path prefix denies search permission, or write permission
24959 is denied on the parent directory of the FIFO to be created.

24960 [EEXIST] The named file already exists.

24961 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
24962 argument.

24963 [ENAMETOOLONG]
24964 The length of the *path* argument exceeds {PATH_MAX} or a path name
24965 component is longer than {NAME_MAX}.

24966 [ENOENT] A component of the path prefix specified by *path* does not name an existing
24967 directory or *path* is an empty string.

24968 [ENOSPC] The directory that would contain the new file cannot be extended or the file
24969 system is out of file-allocation resources.

24970 [ENOTDIR] A component of the path prefix is not a directory.

24971 [EROFS] The named file resides on a read-only file system.

24972 The *mkfifo*( ) function may fail if:

24973 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
24974 resolution of the *path* argument.

24975 [ENAMETOOLONG]
24976 As a result of encountering a symbolic link in resolution of the *path* argument,
24977 the length of the substituted path name string exceeded {PATH_MAX}.

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.

```
#include <sys/types.h>
#include <sys/stat.h>

int status;
...
status = mkfifo("/home/cnd/mod_done", S_IWUSR | S_IRUSR |
    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.

See *getgroups*( ) about the group of a newly created FIFO.

FUTURE DIRECTIONS

None.

SEE ALSO

umask( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument *path* is changed from **char*** to **const char***.
- The description of [EACCES] is updated to indicate that this error is also returned if write permission is denied to the parent directory.

The following changes are incorporated for alignment with the FIPS requirements:

- In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path name component is larger that {NAME_MAX} is now defined as mandatory and marked as an extension.

Issue 4, Version 2

The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- It states that [ELOOP] is returned if too many symbolic links are encountered during path name resolution.

- A second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of path name resolution of a symbolic link.

Issue 6

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

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 [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.

25036 **NAME**

25037 mknod — make a directory, a special or regular file

25038 **SYNOPSIS**

25039 XSI
```
#include <sys/stat.h>

int mknod(const char *path, mode_t mode, dev_t dev);
```
25040

25041

25042 **DESCRIPTION**

25043 The *mknod*( ) function shall create a new file named by the path name to which the argument
25044 *path* points.

25045 The file type for *path* is OR'ed into the *mode* argument, and the application shall select one of the
25046 following symbolic constants:

25047

25048

Name	Description
25049 S_IFIFO	FIFO-special
25050 S_IFCHR	Character-special (non-portable)
25051 S_IFDIR	Directory (non-portable)
25052 S_IFBLK	Block-special (non-portable)
25053 S_IFREG	Regular (non-portable)

25054 The only portable use of *mknod*( ) is to create a FIFO-special file. If *mode* is not S_IFIFO or *dev* is
25055 not 0, the behavior of *mknod*( ) is unspecified.

25056 The permissions for the new file are OR'ed into the *mode* argument, and may be selected from
25057 any combination of the following symbolic constants:

25058

25059

Name	Description
25060 S_ISUID	Set user ID on execution.
25061 S_ISGID	Set group ID on execution.
25062 S_IRWXU	Read, write, or execute (search) by owner.
25063 S_IRUSR	Read by owner.
25064 S_IWUSR	Write by owner.
25065 S_IXUSR	Execute (search) by owner.
25066 S_IRWXG	Read, write, or execute (search) by group.
25067 S_IRGRP	Read by group.
25068 S_IWGRP	Write by group.
25069 S_IXGRP	Execute (search) by group.
25070 S_IRWXO	Read, write, or execute (search) by others.
25071 S_IROTH	Read by others.
25072 S_IWOTH	Write by others.
25073 S_IXOTH	Execute (search) by others.
25074 S_ISVTX	On directories, restricted deletion flag.

25075 The user ID of the file is initialized to the effective user ID of the process. The group ID of the file
25076 is initialized to either the effective group ID of the process or the group ID of the parent
25077 directory.

25078 The owner, group, and other permission bits of *mode* are modified by the file mode creation
25079 mask of the process. The *mknod*( ) function clears each bit whose corresponding bit in the file
25080 mode creation mask of the process is set.

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.

RETURN VALUE

Upon successful completion, *mknod*( ) shall return 0. Otherwise, it shall return –1, the new file shall not be created, and *errno* shall be set to indicate the error.

ERRORS

The *mknod*( ) function 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 *path* argument.

[ENAMETOOLONG] The length of a path name exceeds {PATH_MAX} or a path name component is longer than {NAME_MAX}.

[ENOENT] A component of the path prefix specified by *path* does not name an existing directory or *path* is an empty string.

[ENOSPC] The directory that would contain the new file cannot be extended or the file system is out of file allocation resources.

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The invoking process does not have appropriate privileges and the file type is not FIFO-special.

[EROFS] The directory in which the file is to be created is located on a read-only file system.

The *mknod*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG] Path name resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

EXAMPLES

Creating a FIFO Special File

The following example shows how to create a FIFO special file named **/home/cnd/mod_done**, with read/write permissions for owner, and with read permissions for group and others.

```
#include <sys/types.h>
#include <sys/stat.h>

dev_t dev;
int   status;
...
status  = mknod("/home/cnd/mod_done", S_IFIFO | S_IWUSR |
    S_IRUSR | S_IRGRP | S_IROTH, dev);
```

APPLICATION USAGE

mkfifo( ) is preferred over this function for making FIFO special files.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *creat*( ), *exec*, *mkdir*( ), *mkfifo*( ), *open*( ), *stat*( ), *umask*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/stat.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The DESCRIPTION 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.

NAME

mkstemp — make a unique file name

SYNOPSIS

XSI
```
#include <stdlib.h>

int mkstemp(char *template);
```

DESCRIPTION

The *mkstemp*( ) function shall replace the contents of the string pointed to by *template* by a unique file name, and return a file descriptor for the file open for reading and writing. The function thus prevents any possible race condition between testing whether the file exists and opening it for use. The string in *template* should look like a file name with six trailing *X*s; *mkstemp*( ) replaces each *X* with a character from the portable file name character set. The characters are chosen such that the resulting name does not duplicate the name of an existing file at the time of a call to *mkstemp*( ).

RETURN VALUE

Upon successful completion, *mkstemp*( ) shall return an open file descriptor. Otherwise, –1 shall be returned if no suitable file could be created.

ERRORS

No errors are defined.

EXAMPLES

Generating a File Name

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 *mkstemp*( ) function need not check to determine whether the file name part of *template* exceeds the maximum allowable file name length.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getpid( ), *open*( ), *tmpfile*( ), *tmpnam*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>**

25185 **CHANGE HISTORY**

25186 First released in Issue 4, Version 2.

25187 **Issue 5**

25188 Moved from X/OPEN UNIX extension to BASE.

NAME

mktemp — make a unique file name (**LEGACY**)

SYNOPSIS

XSI
```
#include <stdlib.h>

char *mktemp(char *template);
```

DESCRIPTION

The *mktemp*( ) function shall replace the contents of the string pointed to by *template* by a unique file name and return *template*. The application shall initialize *template* to be a file name with six trailing *X*s; *mktemp*( ) shall replace each *X* with a single byte character from the portable file name character set.

RETURN VALUE

The *mktemp*( ) function shall return the pointer *template*. If a unique name cannot be created, *template* shall point to a null string.

ERRORS

No errors are defined.

EXAMPLES

Generating a File Name

The following example replaces the contents of the `"template"` string with a 10-character file name beginning with the characters `"file"` and returns a pointer to the `"template"` string that contains the new file name.

```
#include <stdlib.h>
...
char *template = "/tmp/fileXXXXXX";
char *ptr;

ptr = mktemp(template);
```

APPLICATION USAGE

Between the time a path name is created and the file opened, it is possible for some other process to create a file with the same name. The *mkstemp*( ) function avoids this problem and is preferred over this function.

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

mkstemp( ), *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 6

This function is marked LEGACY.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

25232 **NAME**

25233 mktime — convert broken-down time into time since the Epoch

25234 **SYNOPSIS**

25235
```
#include <time.h>
```

25236
```
time_t mktime(struct tm *timeptr);
```

25237 **DESCRIPTION**

25238 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25239 conflict between the requirements described here and the ISO C standard is unintentional. This
25240 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

25241 The *mktime*( ) function shall convert the broken-down time, expressed as local time, in the
25242 structure pointed to by *timeptr*, into a time since the Epoch value with the same encoding as that
25243 of the values returned by *time*( ). The original values of the *tm_wday* and *tm_yday* components of
25244 the structure are ignored, and the original values of the other components are not restricted to
25245 the ranges described in **<time.h>**.

25246 CX A positive or 0 value for *tm_isdst* shall cause *mktime*( ) to presume initially that Daylight Savings
25247 Time, respectively, is or is not in effect for the specified time. A negative value for *tm_isdst* shall
25248 cause *mktime*( ) to attempt to determine whether Daylight Saving Time is in effect for the
25249 specified time.

25250 Local timezone information shall be set as though *mktime*( ) called *tzset*( ).

25251 Upon successful completion, the values of the *tm_wday* and *tm_yday* components of the structure
25252 shall be set appropriately, and the other components are set to represent the specified time since
25253 the Epoch, but with their values forced to the ranges indicated in the **<time.h>** entry; the final
25254 value of *tm_mday* shall not be set until *tm_mon* and *tm_year* are determined.

25255 **RETURN VALUE**

25256 The *mktime*( ) function shall return the specified time since the Epoch encoded as a value of type
25257 **time_t**. If the time since the Epoch cannot be represented, the function shall return the value
25258 (**time_t**)–1.

25259 **ERRORS**

25260 No errors are defined.

25261 **EXAMPLES**

25262 What day of the week is July 4, 2001?

25263
```
#include <stdio.h>
```
25264
```
#include <time.h>
```

25265
```
struct tm time_str;
```

25266
```
char daybuf[20];
```

25267
```
int main(void)
```
25268
```
{
```
25269
```
    time_str.tm_year = 2001 – 1900;
```
25270
```
    time_str.tm_mon = 7 – 1;
```
25271
```
    time_str.tm_mday = 4;
```
25272
```
    time_str.tm_hour = 0;
```
25273
```
    time_str.tm_min = 0;
```
25274
```
    time_str.tm_sec = 1;
```
25275
```
    time_str.tm_isdst = –1;
```
25276
```
    if (mktime(&time_str) == -1)
```

```
25277                (void)puts("-unknown-");
25278            else {
25279                (void)strftime(daybuf, sizeof(daybuf), "%A", &time_str);
25280                (void)puts(daybuf);
25281            }
25282            return 0;
25283        }
```

25284 **APPLICATION USAGE**

25285 None.

25286 **RATIONALE**

25287 None.

25288 **FUTURE DIRECTIONS**

25289 None.

25290 **SEE ALSO**

25291 *asctime*( ), *clock*( ), *ctime*( ), *difftime*( ), *gmtime*( ), *localtime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ), |
25292 the Base Definitions volume of IEEE Std. 1003.1-200x, **<time.h>** |

25293 **CHANGE HISTORY**

25294 First released in Issue 3.

25295 Entry included for alignment with the POSIX.1-1988 standard and the ANSI C standard.

25296 **Issue 4**

25297 In the DESCRIPTION, a paragraph is added indicating the possible settings of *tm_isdst*, and
25298 reference to setting of *tm_sec* for leap seconds or double leap seconds is removed (although this
25299 functionality is still supported).

25300 In the EXAMPLES section, the sample code is updated to use ISO C standard syntax.

25301 **Issue 6**

25302 Extensions beyond the ISO C standard are now marked.

25303 **NAME**

25304 mlock, munlock — lock or unlock a range of process address space (**REALTIME**)

25305 **SYNOPSIS**

25306 MLR
```
#include <sys/mman.h>

int mlock(const void * addr, size_t len);
int munlock(const void * addr, size_t len);
```
25307
25308
25309

25310 **DESCRIPTION**

25311 The *mlock*( ) function shall cause those whole pages containing any part of the address space of
25312 the process starting at address *addr* and continuing for *len* bytes to be memory-resident until
25313 unlocked or until the process exits or *exec*s another process image. The implementation may
25314 require that *addr* be a multiple of {PAGESIZE}.

25315 The *munlock*( ) function shall unlock those whole pages containing any part of the address space
25316 of the process starting at address *addr* and continuing for *len* bytes, regardless of how many
25317 times *mlock*( ) has been called by the process for any of the pages in the specified range. The
25318 implementation may require that *addr* be a multiple of the {PAGESIZE}.

25319 If any of the pages in the range specified to a call to *munlock*( ) are also mapped into the address
25320 spaces of other processes, any locks established on those pages by another process are
25321 unaffected by the call of this process to *munlock*( ). If any of the pages in the range specified by a
25322 call to *munlock*( ) are also mapped into other portions of the address space of the calling process
25323 outside the range specified, any locks established on those pages via the other mappings are also
25324 unaffected by this call.

25325 Upon successful return from *mlock*( ), pages in the specified range shall be locked and memory-
25326 resident. Upon successful return from *munlock*( ), pages in the specified range shall be unlocked
25327 with respect to the address space of the process. Memory residency of unlocked pages is
25328 unspecified.

25329 The appropriate privilege is required to lock process memory with *mlock*( ).

25330 **RETURN VALUE**

25331 Upon successful completion, the *mlock*( ) and *munlock*( ) functions shall return a value of zero.
25332 Otherwise, no change is made to any locks in the address space of the process, and the function
25333 shall return a value of –1 and set *errno* to indicate the error.

25334 **ERRORS**

25335 The *mlock*( ) and *munlock*( ) functions shall fail if:

25336 [ENOMEM] Some or all of the address range specified by the *addr* and *len* arguments does |
25337 not correspond to valid mapped pages in the address space of the process.

25338 The *mlock*( ) function shall fail if:

25339 [EAGAIN] Some or all of the memory identified by the operation could not be locked |
25340 when the call was made.

25341 The *mlock*( ) and *munlock*( ) functions may fail if:

25342 [EINVAL] The *addr* argument is not a multiple of {PAGESIZE}. |

25343 The *mlock*( ) function may fail if:

25344 [ENOMEM] Locking the pages mapped by the specified range would exceed an |
25345 implementation-defined limit on the amount of memory that the process may |
25346 lock.

25347 [EPERM] The calling process does not have the appropriate privilege to perform the |
25348 requested operation.

25349 **EXAMPLES**

25350 None.

25351 **APPLICATION USAGE**

25352 None.

25353 **RATIONALE**

25354 None.

25355 **FUTURE DIRECTIONS**

25356 None.

25357 **SEE ALSO**

25358 *exec*, *exit*( ), *fork*( ), *mlockall*( ), *munmap*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
25359 **<sys/mman.h>**

25360 **CHANGE HISTORY**

25361 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

25362 **Issue 6**

25363 The *mlock*( ) and *munlock*( ) functions are marked as part of the Range Memory Locking option. |

25364 The [ENOSYS] error condition has been removed as stubs need not be provided if an
25365 implementation does not support the Range Memory Locking option. |

NAME

mlockall, munlockall — lock/unlock the address space of a process (**REALTIME**)

SYNOPSIS

ML
```
#include <sys/mman.h>

int mlockall(int flags);
int munlockall(void);
```

DESCRIPTION

The *mlockall*() function shall cause all of the pages mapped by the address space of a process to be memory-resident until unlocked or until the process exits or *exec*s another process image. The *flags* argument determines whether the pages to be locked are those currently mapped by the address space of the process, those that are mapped in the future, or both. The *flags* argument is constructed from the bitwise-inclusive OR of one or more of the following symbolic constants, defined in **<sys/mman.h>**:

MCL_CURRENT Lock all of the pages currently mapped into the address space of the process.

MCL_FUTURE Lock all of the pages that become mapped into the address space of the process in the future, when those mappings are established.

If MCL_FUTURE is specified, and the automatic locking of future mappings eventually causes the amount of locked memory to exceed the amount of available physical memory or any other implementation-defined limit, the behavior is implementation-defined. The manner in which the implementation informs the application of these situations is also implementation-defined.

The *munlockall*() function unlocks all currently mapped pages of the address space of the process. Any pages that become mapped into the address space of the process after a call to *munlockall*() shall not be locked, unless there is an intervening call to *mlockall*() specifying MCL_FUTURE or a subsequent call to *mlockall*() specifying MCL_CURRENT. If pages mapped into the address space of the process are also mapped into the address spaces of other processes and are locked by those processes, the locks established by the other processes are unaffected by a call by this process to *munlockall*().

Upon successful return from the *mlockall*() function that specifies MCL_CURRENT, all currently mapped pages of the process' address space shall be memory-resident and locked. Upon return from the *munlockall*() function, all currently mapped pages of the process' address space shall be unlocked with respect to the process' address space. The memory residency of unlocked pages is unspecified.

The appropriate privilege is required to lock process memory with *mlockall*().

RETURN VALUE

Upon successful completion, the *mlockall*() function shall return a value of zero. Otherwise, no additional memory shall be locked, and the function shall return a value of −1 and set *errno* to indicate the error. The effect of failure of *mlockall*() on previously existing locks in the address space is unspecified.

If it is supported by the implementation, the *munlockall*() function shall always return a value of zero. Otherwise, the function shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *mlockall*() function shall fail if:

[EAGAIN] Some or all of the memory identified by the operation could not be locked when the call was made.

[EINVAL] The *flags* argument is zero, or includes unimplemented flags.

The *mlockall*( ) function may fail if:

[ENOMEM] Locking all of the pages currently mapped into the address space of the process would exceed an implementation-defined limit on the amount of memory that the process may lock.

[EPERM] The calling process does not have the appropriate privilege to perform the 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.

NAME

mmap — map pages of memory

SYNOPSIS

MF|SHM
```
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flags,
    int fildes, off_t off);
```

DESCRIPTION

The *mmap*( ) function shall establish a mapping between a process' address space and a file, shared memory object, or typed memory object. The format of the call is as follows:

```
pa=mmap(addr, len, prot, flags, fildes, off);
```

The *mmap*( ) function establishes a mapping between the address space of the process at an address *pa* for *len* bytes to the memory object represented by the file descriptor *fildes* at offset *off* for *len* bytes. The value of *pa* is an implementation-defined function of the parameter *addr* and the values of *flags*, further described below. A successful *mmap*( ) call shall return *pa* as its result. The address range starting at *pa* and continuing for *len* bytes shall be legitimate for the possible (not necessarily current) address space of the process. The range of bytes starting at *off* and continuing for *len* bytes shall be legitimate for the possible (not necessarily current) offsets in the file, shared memory object, or typed memory object represented by *fildes*.

TYM If *fildes* represents a typed memory object opened with either the POSIX_TYPED_MEM_ALLOCATE flag or the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, the memory object to be mapped shall be that portion of the typed memory object allocated by the implementation as specified below. In this case, if *off* is non-zero, the behavior of *mmap*( ) is undefined. If *fildes* refers to a valid typed memory object that is not accessible from the calling process, *mmap*( ) shall fail.

The mapping established by *mmap*( ) replaces any previous mappings for those whole pages containing any part of the address space of the process starting at *pa* and continuing for *len* bytes.

If the size of the mapped file changes after the call to *mmap*( ) as a result of some other operation on the mapped file, the effect of references to portions of the mapped region that correspond to added or removed portions of the file is unspecified.

TYM The *mmap*( ) function is supported for regular files, shared memory objects, and typed memory objects. Support for any other type of file is unspecified.

The parameter *prot* determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. The *prot* should be either PROT_NONE or the bitwise-inclusive OR of one or more of the other flags in the following table, defined in the header **<sys/mman.h>**.

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.

If an implementation cannot support the combination of access types specified by *prot*, the call to *mmap*( ) fails. An implementation may permit accesses other than those specified by *prot*; however, if the Memory Protection option is supported, the implementation shall not permit a

write to succeed where PROT_WRITE has not been set or 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. If the Memory Protection option is not supported, the result of any access that conflicts with the specified protection is undefined. The file descriptor *fildes* 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 *fildes* with write permission unless MAP_PRIVATE is specified in the *flags* parameter as described below.

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>**:

Symbolic Constant	Description
MAP_SHARED	Changes are shared.
MAP_PRIVATE	Changes are private.
MAP_FIXED	Interpret *addr* exactly.

Implementations that do not support the Memory Mapped Files option are not required to
XSI support MAP_PRIVATE. It is implementation-defined whether MAP_FIXED shall be supported.
MAP_FIXED shall be supported on XSI-conformant systems.

MAP_SHARED and MAP_PRIVATE describe the disposition of write references to the memory object. If MAP_SHARED is specified, write references 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*( ).

TYM When *fildes* 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 *fildes* 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 *fildes* 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 *fildes* 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 port), both processes shall map the same region of storage.

When MAP_FIXED is set in the *flags* argument, the implementation is informed that the value of *pa* shall be *addr*, exactly. If MAP_FIXED is set, *mmap*( ) may return MAP_FAILED and set *errno* to [EINVAL]. If a MAP_FIXED request is successful, the mapping established by *mmap*( ) replaces any previous mappings for the process' pages in the range [*pa*,*pa*+*len*).

When MAP_FIXED is not set, the implementation uses *addr* in an implementation-defined manner to arrive at *pa*. The *pa* so chosen shall be an area of the address space that the

implementation deems suitable for a mapping of *len* bytes to the file. All implementations interpret an *addr* value of 0 as granting the implementation complete freedom in selecting *pa*, subject to constraints described below. A non-zero value of *addr* is taken to be a suggestion of a process address near which the mapping should be placed. When the implementation selects a value for *pa*, it never places a mapping at address 0, nor does it replace any extant mapping.

The *off* argument is constrained to be aligned and sized according to the value returned by *sysconf*( ) when passed _SC_PAGESIZE or _SC_PAGE_SIZE. When MAP_FIXED is specified, the application shall ensure that the argument *addr* also meets these constraints. The implementation performs mapping operations over whole pages. Thus, while the argument *len* need not meet a size or alignment constraint, the implementation shall include, in any mapping operation, any partial page specified by the range [*pa*,*pa*+*len*).

The system shall always zero-fill any partial page at the end of an object. Further, the system shall never write out any modified portions of the last page of an object which are beyond its
MPR end. References within the address range starting at *pa* and continuing for *len* bytes to whole pages following the end of an object shall result in delivery of a SIGBUS signal.

An implementation may generate SIGBUS signals when a reference would cause an error in the mapped object, such as out-of-space condition.

The *mmap*( ) function adds an extra reference to the file associated with the file descriptor *fildes* which is not removed by a subsequent *close*( ) on that file descriptor. This reference is removed when there are no more mappings to the file.

The *st_atime* field of the mapped file may be marked for update at any time between the *mmap*( ) call and the corresponding *munmap*( ) call. The initial read or write reference to a mapped region shall cause the file's *st_atime* field to be marked for update if it has not already been marked for update.

The *st_ctime* and *st_mtime* fields of a file that is mapped with MAP_SHARED and PROT_WRITE shall be marked for update at some point in the interval between a write reference to the mapped region and the next call to *msync*( ) with MS_ASYNC or MS_SYNC for that portion of the file by any process. If there is no such call and if the underlying file is modified as a result of a write reference, then these fields shall be marked for update at some time after the write reference.

There may be implementation-defined limits on the number of memory regions that can be mapped (per process or per system).

XSI If such a limit is imposed, whether the number of memory regions that can be mapped by a process is decreased by the use of *shmat*( ) is implementation-defined.

If *mmap*( ) fails for reasons other than [EBADF], [EINVAL], or [ENOTSUP], some of the mappings in the address range starting at *addr* and continuing for *len* bytes may have been unmapped.

RETURN VALUE

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 header **<sys/mman.h>**. No successful return from *mmap*( ) shall return the value MAP_FAILED.

ERRORS

The *mmap*( ) function shall fail if:

[EACCES] The *fildes* argument is not open for read, regardless of the protection specified, or *fildes* is not open for write and PROT_WRITE was specified for a

MAP_SHARED type mapping.

ML [EAGAIN] The mapping could not be locked in memory, if required by *mlockall*(), due to a lack of resources.

[EBADF] The *fildes* argument is not a valid open file descriptor.

[EINVAL] The *addr* argument (if MAP_FIXED was specified) or *off* is not a multiple of the page size as returned by *sysconf*(), or are considered invalid by the implementation.

[EINVAL] The value of *flags* is invalid (neither MAP_PRIVATE nor MAP_SHARED is set).

[EMFILE] The number of mapped regions would exceed an implementation-defined limit (per process or per system).

[ENODEV] The *fildes* argument refers to a file whose type is not supported by *mmap*().

[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.

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.

MAP_FIXED or MAP_PRIVATE was specified in the *flags* argument and the implementation does not support this functionality.

TYM [ENOMEM] Not enough unallocated memory resources remain in the typed memory object designated by *fildes* to allocate *len* bytes.

[ENOTSUP] The implementation does not support the combination of accesses requested in the *prot* argument.

[ENXIO] Addresses in the range [*off*,*off+len*) are invalid for the object specified by *fildes*.

[ENXIO] MAP_FIXED was specified in *flags* and the combination of *addr*, *len*, and *off* is invalid for the object specified by *fildes*.

TYM [ENXIO] The *fildes* argument refers to a typed memory object that is not accessible from the calling process.

[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 *fildes*.

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.

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. */
```

The [EINVAL] error above is marked EX because it is defined as an optional error in the POSIX Realtime Extension.

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, when the Memory Protection option is supported, that all hardware implementations are able to support all combinations of permissions at all addresses. When this option is supported, implementations are required to disallow write access to mappings without write permission and to disallow access to mappings without any access permission. Other than these restrictions, implementations may allow access types other than those requested by the application. For example, if the application requests only PROT_WRITE, the implementation may also allow read access. A call to *mmap*( ) fails if the implementation cannot support allowing all the access requested by the application. For example, some implementations cannot support a request for both write access and execute access simultaneously. All implementations supporting the Memory Protection option must support requests for no access, read access, write access, and both read and write access. Strictly conforming code must only rely on the required checks. These restrictions allow for portability across a wide range of hardware.

The MAP_FIXED address treatment is likely to fail for non-page-aligned values and for certain architecture-dependent address ranges. Conforming implementations cannot count on being able to choose address values for MAP_FIXED without utilizing non-portable, implementation-defined knowledge. Nonetheless, MAP_FIXED is provided as a standard interface conforming to existing practice for utilizing such knowledge when it is available.

Similarly, in order to allow implementations that do not support virtual addresses, support for directly specifying any mapping addresses via MAP_FIXED is not required and thus a conforming application may not count on it.

The MAP_PRIVATE function can be implemented efficiently when memory protection hardware is available. When such hardware is not available, implementations can implement such ''mappings'' by simply making a real copy of the relevant data into process private memory, though this tends to behave similarly to *read*( ).

The function has been defined to allow for many different models of using shared memory. However, all uses are not equally portable across all machine architectures. In particular, the *mmap*( ) function allows the system as well as the application to specify the address at which to map a specific region of a memory object. The most portable way to use the function is always to let the system choose the address, specifying NULL as the value for the argument *addr* and not to specify MAP_FIXED.

If it is intended that a particular region of a memory object be mapped at the same address in a group of processes (on machines where this is even possible), then MAP_FIXED can be used to pass in the desired mapping address. The system can still be used to choose the desired address if the first such mapping is made without specifying MAP_FIXED, and then the resulting mapping address can be passed to subsequent processes for them to pass in via MAP_FIXED. The availability of a specific address range cannot be guaranteed, in general.

The *mmap*( ) function can be used to map a region of memory that is larger than the current size of the object. Memory access within the mapping but beyond the current end of the underlying objects may result in SIGBUS signals being sent to the process. The reason for this is that the size of the object can be manipulated by other processes and can change at any moment. The implementation should tell the application that a memory reference is outside the object where this can be detected; otherwise, written data may be lost and read data may not reflect actual data in the object.

Note that references beyond the end of the object do not extend the object as the new end cannot be determined precisely by most virtual memory hardware. Instead, the size can be directly manipulated by *ftruncate*( ).

Process memory locking does apply to shared memory regions, and the MEMLOCK_FUTURE argument to *memlockall*( ) can be relied upon to cause new shared memory regions to be automatically locked.

Existing implementations of *mmap*( ) return the value –1 when unsuccessful. Since the casting of this value to type **void*** cannot be guaranteed by the ISO C standard to be distinct from a successful value, this volume of IEEE Std. 1003.1-200x defines the symbol MAP_FAILED, which a conforming implementation does not return as the result of a successful call.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fcntl*( ), *fork*( ), *lockf*( ), *msync*( ), *munmap*( ), *mprotect*( ), *posix_typed_mem_open*( ), *shmat*( ), *sysconf*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Aligned with *mmap*( ) in the POSIX Realtime Extension as follows:

- The DESCRIPTION is extensively reworded.
- The [EAGAIN] and [ENOTSUP] mandatory error conditions are added.
- New cases of [ENOMEM] and [ENXIO] are added as mandatory error conditions.
- The value returned on failure is the value of the constant MAP_FAILED; this was previously defined as –1.

Large File Summit extensions are added.

Issue 6

The *mmap*( ) function is marked as part of the Memory Mapped Files option.

The Open Group corrigenda item U028/6 has been applied, changing (**void** *)–1 to MAP_FAILED.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to described the use of MAP_FIXED.
- The DESCRIPTION is updated to describe the addition of an extra reference to the file associated with the file descriptor passed to *mmap*( ).
- The DESCRIPTION is updated to state that there may be implementation-defined limits on the number of memory regions that can be mapped.
- The DESCRIPTION is updated to describe constraints on the alignment and size of the *off* argument.
- The [EINVAL] and [EMFILE] error conditions are added.
- The [EOVERFLOW] error condition is added. This change is to support large files.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The DESCRIPTION is updated to describe the cases when MAP_PRIVATE and MAP_FIXED need not be supported.

The following changes are made for alignment with IEEE Std. 1003.1j-2000:

- Semantics for typed memory objects are added to the DESCRIPTION.
- New [ENOMEM] and [ENXIO] errors are added to the ERRORS section.
- The *posix_typed_mem_open*( ) function is added to the SEE ALSO section.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

25746 **NAME**

25747 modf, modff, modfl — decompose a floating-point number

25748 **SYNOPSIS**

25749
```
#include <math.h>
```

25750
```
double modf(double x, double *iptr);
```
25751
```
float modff(float value, float *iptr);
```
25752
```
long double modfl(long double value, long double *iptr);
```

25753 **DESCRIPTION**

25754 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25755 conflict between the requirements described here and the ISO C standard is unintentional. This
25756 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

25757 These functions shall break the argument *x* into integral and fractional parts, each of which has
25758 the same sign as the argument. It stores the integral part as a double in the object pointed to by
25759 *iptr*.

25760 An application wishing to check for error situations should set *errno* to 0 before calling *modf*( ). If
25761 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

25762 **RETURN VALUE**

25763 Upon successful completion, these functions shall return the signed fractional part of *x*.

25764 XSI If *x* is NaN, NaN shall be returned, *errno* may be set to [EDOM], and **iptr* shall be set to NaN.

25765 If the correct value would cause underflow, 0 shall be returned and *errno* may be set to
25766 [ERANGE].

25767 **ERRORS**

25768 These functions may fail if:

25769 XSI [EDOM] The value of *x* is NaN.

25770 [ERANGE] The result underflows.

25771 XSI No other errors shall occur.

25772 **EXAMPLES**

25773 None.

25774 **APPLICATION USAGE**

25775 The *modf*( ) function computes the function result and **iptr* such that:

25776
```
a = modf(x, &iptr)
```
25777
```
x == a+iptr
```

25778 allowing for the usual floating point inaccuracies.

25779 **RATIONALE**

25780 None.

25781 **FUTURE DIRECTIONS**

25782 None.

25783 **SEE ALSO**

25784 *frexp*( ), *isnan*( ), *ldexp*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

25785 **CHANGE HISTORY**

25786 First released in Issue 1. Derived from Issue 1 of the SVID.

25787 **Issue 4**

25788 References to *matherr*( ) are removed.

25789 The name of the first argument is changed from *value* to *x*.

25790 The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the
25791 ISO C standard and to rationalize error handling in the mathematics functions.

25792 The return value specified for [EDOM] is marked as an extension.

25793 **Issue 5**

25794 The DESCRIPTION is updated to indicate how an application should check for an error. This
25795 text was previously published in the APPLICATION USAGE section.

25796 **Issue 6**

25797 The *modff*( ) and *modfl*( ) functions are added for alignment with the ISO/IEC 9899:1999
25798 standard.

25799 **NAME**

25800 mprotect — set protection of memory mapping

25801 **SYNOPSIS**

25802 MPR
```
#include <sys/mman.h>

int mprotect(void *addr, size_t len, int prot);
```
25803

25804

25805 **DESCRIPTION**

25806 The *mprotect*( ) function shall change the access protections to be that specified by *prot* for those
25807 whole pages containing any part of the address space of the process starting at address *addr* and
25808 continuing for *len* bytes. The parameter *prot* determines whether read, write, execute, or some
25809 combination of accesses are permitted to the data being mapped. The *prot* argument should be
25810 either PROT_NONE or the bitwise-inclusive OR of one or more of PROT_READ, PROT_WRITE,
25811 and PROT_EXEC.

25812 If an implementation cannot support the combination of access types specified by *prot*, the call
25813 to *mprotect*( ) shall fail.

25814 An implementation may permit accesses other than those specified by *prot*; however, no |
25815 implementation shall permit a write to succeed where PROT_WRITE has not been set or shall |
25816 permit any access where PROT_NONE alone has been set. Implementations shall support at |
25817 least the following values of *prot*: PROT_NONE, PROT_READ, PROT_WRITE, and the bitwise- |
25818 inclusive OR of PROT_READ and PROT_WRITE. If PROT_WRITE is specified, the application
25819 shall ensure that it has opened the mapped objects in the specified address range with write
25820 permission, unless MAP_PRIVATE was specified in the original mapping, regardless of whether
25821 the file descriptors used to map the objects have since been closed.

25822 The implementation shall require that *addr* be a multiple of the page size as returned by |
25823 *sysconf*( ).

25824 The behavior of this function is unspecified if the mapping was not established by a call to
25825 *mmap*( ).

25826 When *mprotect*( ) fails for reasons other than [EINVAL], the protections on some of the pages in
25827 the range [*addr*,*addr*+*len*) may have been changed.

25828 **RETURN VALUE**

25829 Upon successful completion, *mprotect*( ) shall return 0; otherwise, it shall return –1 and set *errno*
25830 to indicate the error.

25831 **ERRORS**

25832 The *mprotect*( ) function shall fail if:

25833 [EACCES] The *prot* argument specifies a protection that violates the access permission |
25834 the process has to the underlying memory object.

25835 [EAGAIN] The *prot* argument specifies PROT_WRITE over a MAP_PRIVATE mapping |
25836 and there are insufficient memory resources to reserve for locking the private
25837 page. |

25838 [EINVAL] The *addr* argument is not a multiple of the page size as returned by *sysconf*( ). |

25839 [ENOMEM] Addresses in the range [*addr*,*addr*+*len*) are invalid for the address space of a |
25840 process, or specify one or more pages which are not mapped.

25841 [ENOMEM] The *prot* argument specifies PROT_WRITE on a MAP_PRIVATE mapping, and
25842 it would require more space than the system is able to supply for locking the
25843 private pages, if required.

[ENOTSUP] The implementation does not support the combination of accesses requested in the *prot* argument.

EXAMPLES

None.

APPLICATION USAGE

The [EINVAL] error above is marked EX because it is defined as an optional error in the POSIX Realtime Extension.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mmap( ), *sysconf*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Aligned with *mprotect*( ) in the POSIX Realtime Extension as follows:

- The DESCRIPTION is largely reworded.
- [ENOTSUP] and a second form of [ENOMEM] are added as mandatory error conditions.
- [EAGAIN] is moved from the optional to the mandatory error conditions.

Issue 6

The *mprotect*( ) function is marked as part of the Memory Protection 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 as returned by *sysconf*( ).
- The [EINVAL] error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

25873 **NAME**

25874 mq_close — close a message queue (**REALTIME**)

25875 **SYNOPSIS**

25876 MSG
```
#include <mqueue.h>
```

25877
```
int mq_close(mqd_t mqdes);
```

25878

25879 **DESCRIPTION**

25880 The *mq_close*( ) function shall remove the association between the message queue descriptor,
25881 *mqdes*, and its message queue. The results of using this message queue descriptor after
25882 successful return from this *mq_close*( ), and until the return of this message queue descriptor
25883 from a subsequent *mq_open*( ), are undefined.

25884 If the process has successfully attached a notification request to the message queue via this
25885 *mqdes*, this attachment shall be removed, and the message queue is available for another process
25886 to attach for notification.

25887 **RETURN VALUE**

25888 Upon successful completion, the *mq_close*( ) function shall return a value of zero; otherwise, the
25889 function shall return a value of –1 and set *errno* to indicate the error.

25890 **ERRORS**

25891 The *mq_close*( ) function shall fail if:

25892 [EBADF] The *mqdes* argument is not a valid message queue descriptor. |

25893 **EXAMPLES**

25894 None.

25895 **APPLICATION USAGE**

25896 None.

25897 **RATIONALE**

25898 None.

25899 **FUTURE DIRECTIONS**

25900 None.

25901 **SEE ALSO**

25902 *mq_open*( ), *mq_unlink*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of |
25903 IEEE Std. 1003.1-200x, **<mqueue.h>** |

25904 **CHANGE HISTORY**

25905 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

25906 **Issue 6**

25907 The *mq_close*( ) function is marked as part of the Message Passing option. |

25908 The [ENOSYS] error condition has been removed as stubs need not be provided if an
25909 implementation does not support the Message Passing option. |

25910 **NAME**

25911 mq_getattr — get message queue attributes (**REALTIME**)

25912 **SYNOPSIS**

25913 MSG
```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);
```
25914
25915

25916 **DESCRIPTION**

25917 The *mqdes* argument specifies a message queue descriptor.

25918 The *mq_getattr*( ) function is used to get status information and attributes of the message queue
25919 and the open message queue description associated with the message queue descriptor.

25920 The results are returned in the **mq_attr** structure referenced by the *mqstat* argument.

25921 Upon return, the following members have the values associated with the open message queue
25922 description as set when the message queue was opened and as modified by subsequent
25923 *mq_setattr*( ) calls: *mq_flags*.

25924 The following attributes of the message queue shall be returned as set at message queue
25925 creation: *mq_maxmsg*, *mq_msgsize*.

25926 Upon return, the following members within the **mq_attr** structure referenced by the *mqstat*
25927 argument are set to the current state of the message queue:

25928 *mq_curmsgs* The number of messages currently on the queue.

25929 **RETURN VALUE**

25930 Upon successful completion, the *mq_getattr*( ) function shall return zero. Otherwise, the function
25931 shall return –1 and set *errno* to indicate the error.

25932 **ERRORS**

25933 The *mq_getattr*( ) function shall fail if:

25934 [EBADF] The *mqdes* argument is not a valid message queue descriptor. |

25935 **EXAMPLES**

25936 None.

25937 **APPLICATION USAGE**

25938 None.

25939 **RATIONALE**

25940 None.

25941 **FUTURE DIRECTIONS**

25942 None.

25943 **SEE ALSO**

25944 *mq_open*( ), *mq_send*( ), *mq_setattr*( ), *mq_timedsend*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the |
25945 Base Definitions volume of IEEE Std. 1003.1-200x, **<mqueue.h>** |

25946 **CHANGE HISTORY**

25947 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

25948 **Issue 6**

25949 The *mq_getattr*( ) function is marked as part of the Message Passing option. |

25950 The [ENOSYS] error condition has been removed as stubs need not be provided if an
25951 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.

25954 **NAME**

25955 mq_notify — notify process that a message is available (**REALTIME**)

25956 **SYNOPSIS**

25957 MSG
```
#include <mqueue.h>

int mq_notify(mqd_t mqdes, const struct sigevent *notification);
```
25958

25959

25960 **DESCRIPTION**

25961 If the argument *notification* is not NULL, this function registers the calling process to be notified
25962 of message arrival at an empty message queue associated with the specified message queue
25963 descriptor, *mqdes*. The notification specified by the *notification* argument shall be sent to the
25964 process when the message queue transitions from empty to non-empty. At any time, only one
25965 process may be registered for notification by a message queue. If the calling process or any other
25966 process has already registered for notification of message arrival at the specified message queue,
25967 subsequent attempts to register for that message queue fail.

25968 If *notification* is NULL and the process is currently registered for notification by the specified
25969 message queue, the existing registration is removed.

25970 When the notification is sent to the registered process, its registration shall be removed. The
25971 message queue shall then be available for registration.

25972 If a process has registered for notification of message arrival at a message queue and some
25973 thread is blocked in *mq_receive*( ) waiting to receive a message when a message arrives at the
25974 queue, the arriving message satisfies the appropriate *mq_receive*( ). The resulting behavior is as if
25975 the message queue remains empty, and no notification is sent.

25976 **RETURN VALUE**

25977 Upon successful completion, the *mq_notify*( ) function shall return a value of zero; otherwise, the
25978 function shall return a value of –1 and set *errno* to indicate the error.

25979 **ERRORS**

25980 The *mq_notify*( ) function shall fail if:

25981 [EBADF] The *mqdes* argument is not a valid message queue descriptor. |

25982 [EBUSY] A process is already registered for notification by the message queue. |

25983 **EXAMPLES**

25984 None.

25985 **APPLICATION USAGE**

25986 None.

25987 **RATIONALE**

25988 None.

25989 **FUTURE DIRECTIONS**

25990 None.

25991 **SEE ALSO**

25992 *mq_open*( ), *mq_send*( ), *mq_timedsend*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base |
25993 Definitions volume of IEEE Std. 1003.1-200x, **<mqueue.h>** |

25994 **CHANGE HISTORY**

25995 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.

26002 **NAME**

26003 mq_open — open a message queue (**REALTIME**)

26004 **SYNOPSIS**

26005 MSG
```
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag, ...);
```
26006

26007

26008 **DESCRIPTION**

26009 The *mq_open*( ) function shall establish the connection between a process and a message queue
26010 with a message queue descriptor. It creates an open message queue description that refers to the
26011 message queue, and a message queue descriptor that refers to that open message queue
26012 description. The message queue descriptor is used by other functions to refer to that message
26013 queue. The *name* argument points to a string naming a message queue. It is unspecified whether
26014 the name appears in the file system and is visible to other functions that take path names as
26015 arguments. The *name* argument conforms to the construction rules for a path name. If *name*
26016 begins with the slash character, then processes calling *mq_open*( ) with the same value of *name*
26017 refer to the same message queue object, as long as that name has not been removed. If *name* does
26018 not begin with the slash character, the effect is implementation-defined. The interpretation of |
26019 slash characters other than the leading slash character in *name* is implementation-defined. If the |
26020 *name* argument is not the name of an existing message queue and creation is not requested,
26021 *mq_open*( ) shall fail and return an error.

26022 A message queue descriptor may be implemented using a file descriptor, in which case |
26023 applications can open up to at least {OPEN_MAX} file and message queues. |

26024 The *oflag* argument requests the desired receive and/or send access to the message queue. The |
26025 requested access permission to receive messages or send messages is granted if the calling
26026 process would be granted read or write access, respectively, to an equivalently protected file.

26027 The value of *oflag* is the bitwise-inclusive OR of values from the following list. Applications
26028 specify exactly one of the first three values (access modes) below in the value of *oflag*:

26029 O_RDONLY Open the message queue for receiving messages. The process can use the
26030 returned message queue descriptor with *mq_receive*( ), but not *mq_send*( ). A
26031 message queue may be open multiple times in the same or different processes
26032 for receiving messages.

26033 O_WRONLY Open the queue for sending messages. The process can use the returned
26034 message queue descriptor with *mq_send*( ) but not *mq_receive*( ). A message
26035 queue may be open multiple times in the same or different processes for
26036 sending messages.

26037 O_RDWR Open the queue for both receiving and sending messages. The process can use
26038 any of the functions allowed for O_RDONLY and O_WRONLY. A message
26039 queue may be open multiple times in the same or different processes for
26040 sending messages.

26041 Any combination of the remaining flags may be specified in the value of *oflag*:

26042 O_CREAT This option is used to create a message queue, and it requires two additional
26043 arguments: *mode*, which is of type **mode_t**, and *attr*, which is a pointer to a
26044 **mq_attr** structure. If the path name *name* has already been used to create a
26045 message queue that still exists, then this flag has no effect, except as noted
26046 under O_EXCL. Otherwise, a message queue is created without any messages
26047 in it. The user ID of the message queue is set to the effective user ID of the
26048 process, and the group ID of the message queue is set to the effective group ID

of the process. The file permission bits are set to the value of *mode*. When bits in *mode* other than file permission bits are set, the effect is implementation-defined. If *attr* is NULL, the message queue is created with implementation-defined default message queue attributes. If *attr* is non-NULL and the calling process has the appropriate privilege on *name*, the message queue *mq_maxmsg* and *mq_msgsize* attributes are set to the values of the corresponding members in the **mq_attr** structure referred to by *attr*. If *attr* is non-NULL, but the calling process does not have the appropriate privilege on *name*, the *mq_open*( ) function shall fail and return an error without creating the message queue.

O_EXCL — If O_EXCL and O_CREAT are set, *mq_open*( ) fails if the message queue *name* exists. The check for the existence of the message queue and the creation of the message queue if it does not exist shall be atomic with respect to other threads executing *mq_open*( ) naming the same *name* with O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the result is undefined.

O_NONBLOCK — The setting of this flag is associated with the open message queue description and determines whether a *mq_send*( ) or *mq_receive*( ) waits for resources or messages that are not currently available, or fails with *errno* set to [EAGAIN]; see *mq_send*( ) and *mq_receive*( ) for details.

The *mq_open*( ) function does not add or remove messages from the queue.

RETURN VALUE

Upon successful completion, the function shall return a message queue descriptor; otherwise, the function shall return (**mqd_t**)–1 and set *errno* to indicate the error.

ERRORS

The *mq_open*( ) function shall fail if:

[EACCES] — The message queue exists and the permissions specified by *oflag* are denied, or the message queue does not exist and permission to create the message queue is denied.

[EEXIST] — O_CREAT and O_EXCL are set and the named message queue already exists.

[EINTR] — The *mq_open*( ) function was interrupted by a signal.

[EINVAL] — The *mq_open*( ) function is not supported for the given name.

[EINVAL] — O_CREAT was specified in *oflag*, the value of *attr* is not NULL, and either *mq_maxmsg* or *mq_msgsize* was less than or equal to zero.

[EMFILE] — Too many message queue descriptors or file descriptors are currently in use by this process.

[ENAMETOOLONG] — The length of the *name* argument exceeds {PATH_MAX} or a path name component is longer than {NAME_MAX}.

[ENFILE] — Too many message queues are currently open in the system.

[ENOENT] — O_CREAT is not set and the named message queue does not exist.

[ENOSPC] — There is insufficient space for the creation of the new message queue.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_close( ), *mq_getattr*( ), *mq_receive*( ), *mq_send*( ), *mq_setattr*( ), *mq_timedreceive*( ), *mq_timedsend*( ), *mq_unlink*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<mqueue.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mq_open*( ) 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_timedreceive*( ) and *mq_timedsend*( ) functions are added to the SEE ALSO section for alignment with IEEE Std. 1003.1d-1999.

The DESCRIPTION of O_EXCL is updated in response to IEEE PASC Interpretation 1003.1c #48.

NAME

mq_receive, mq_timedreceive — receive a message from a message queue (**REALTIME**)

SYNOPSIS

MSG
```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
    unsigned *msg_prio);
```

MSG TMO
```
#include <mqueue.h>
#include <time.h>

ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr,
    size_t msg_len, unsigned *restrict msg_prio,
    const struct timespec *restrict abs_timeout);
```

DESCRIPTION

The *mq_receive*( ) function is used to receive the oldest of the highest priority message(s) from the message queue specified by *mqdes*. If the size of the buffer in bytes, specified by the *msg_len* argument, is less than the *mq_msgsize* attribute of the message queue, the function shall fail and return an error. Otherwise, the selected message is removed from the queue and copied to the buffer pointed to by the *msg_ptr* argument.

If the value of *msg_len* is greater than {SSIZE_MAX}, the result is implementation-defined.

If the argument *msg_prio* is not NULL, the priority of the selected message is stored in the location referenced by *msg_prio*.

If the specified message queue is empty and O_NONBLOCK is not set in the message queue description associated with *mqdes*, *mq_receive*( ) blocks until a message is enqueued on the message queue or until *mq_receive*( ) is interrupted by a signal. If more than one thread is waiting to receive a message when a message arrives at an empty queue and the Priority Scheduling option is supported, then the thread of highest priority that has been waiting the longest shall be selected to receive the message. Otherwise, it is unspecified which waiting thread receives the message. If the specified message queue is empty and O_NONBLOCK is set in the message queue description associated with *mqdes*, no message is removed from the queue, and *mq_receive*( ) shall return an error.

TMO The *mq_timedreceive*( ) function is used to receive the oldest of the highest priority messages from the message queue specified by *mqdes* as in the *mq_receive*( ) function. However, if O_NONBLOCK was not specified when the message queue was opened via the *mq_open*( ) function, and no message exists on the queue to satisfy the receive, the wait for such a message will be terminated when the specified timeout expires. If O_NONBLOCK is set, this function shall behave identically to *mq_receive*( ).

The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call. If the Timers option is supported, the timeout is based on the CLOCK_REALTIME clock; if the Timers option is not supported, the timeout is based on the system clock as returned by the *time*( ) function. The resolution of the timeout is the resolution of the clock on which it is based. The *timespec* argument is defined as a structure in the **<time.h>** header.

Under no circumstance shall the operation fail with a timeout if a message can be removed from the message queue immediately. The validity of the *abs_timeout* parameter need not be checked if a message can be removed from the message queue immediately.

RETURN VALUE

TMO Upon successful completion, the *mq_receive*( ) and *mq_timedreceive*( ) functions shall return the length of the selected message in bytes and the message shall be removed from the queue. Otherwise, no message shall be removed from the queue, the functions shall return a value of –1, and set *errno* to indicate the error.

ERRORS

TMO The *mq_receive*( ) and *mq_timedreceive*( ) functions shall fail if:

[EAGAIN] O_NONBLOCK was set in the message description associated with *mqdes*, and the specified message queue is empty.

[EBADF] The *mqdes* argument is not a valid message queue descriptor open for reading.

[EMSGSIZE] The specified message buffer size, *msg_len*, is less than the message size attribute of the message queue.

TMO [EINTR] The *mq_receive*( ) or *mq_timedreceive*( ) operation was interrupted by a signal.

TMO [EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1 000 million.

TMO [ETIMEDOUT] The O_NONBLOCK flag was not set when the message queue was opened, but no message arrived on the queue before the specified timeout expired.

TMO The *mq_receive*( ) and *mq_timedreceive*( ) functions may fail if:

[EBADMSG] The implementation has detected a data corruption problem with the message.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_open( ), *mq_send*( ), *mq_timedsend*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), *time*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<mqueue.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mq_receive*( ) function is marked as part of the Message Passing option.

The Open Group corrigenda item U021/4 has been applied. The DESCRIPTION is changed to refer to *msg_len* rather than *maxsize*.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- 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**.

NAME

mq_send, mq_timedsend — send a message to a message queue (**REALTIME**)

SYNOPSIS

MSG
```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
    unsigned msg_prio);
```

MSG TMO
```
#include <mqueue.h>
#include <time.h>

int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
    unsigned msg_prio, const struct timespec *abs_timeout);
```

DESCRIPTION

The *mq_send*( ) function shall add the message pointed to by the argument *msg_ptr* to the message queue specified by *mqdes*. The *msg_len* argument specifies the length of the message in bytes pointed to by *msg_ptr*. The value of *msg_len* is less than or equal to the *mq_msgsize* attribute of the message queue, or *mq_send*( ) shall fail.

If the specified message queue is not full, *mq_send*( ) behaves as if the message shall be inserted into the message queue at the position indicated by the *msg_prio* argument. A message with a larger numeric value of *msg_prio* shall be inserted before messages with lower values of *msg_prio*. A message shall be inserted after other messages in the queue, if any, with equal *msg_prio*. The value of *msg_prio* shall be less than {MQ_PRIO_MAX}.

If the specified message queue is full and O_NONBLOCK is not set in the message queue description associated with *mqdes*, *mq_send*( ) shall block until space becomes available to enqueue the message, or until *mq_send*( ) is interrupted by a signal. If more than one thread is waiting to send when space becomes available in the message queue and the Priority Scheduling option is supported, then the thread of the highest priority that has been waiting the longest shall be unblocked to send its message. Otherwise, it is unspecified which waiting thread is unblocked. If the specified message queue is full and O_NONBLOCK is set in the message queue description associated with *mqdes*, the message shall not be queued and *mq_send*( ) shall return an error.

TMO The *mq_timedsend*( ) function adds a message to the message queue specified by *mqdes* in the manner defined for the *mq_send*( ) function. However, if the specified message queue is full and O_NONBLOCK is not set in the message queue description associated with *mqdes*, the wait for sufficient room in the queue shall be terminated when the specified timeout expires. If O_NONBLOCK is set in the message queue description, this function shall behave identically to *mq_send*( ).

The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call. If the Timers option is supported, the timeout is based on the CLOCK_REALTIME clock; if the Timers option is not supported, the timeout is based on the system clock as returned by the *time*( ) function. The resolution of the timeout is the resolution of the clock on which it is based. The *timespec* argument is defined as a structure in the **<time.h>** header.

Under no circumstance shall the operation fail with a timeout if there is sufficient room in the queue to add the message immediately. The validity of the *abs_timeout* parameter need not be checked when there is sufficient room in the queue.

26255 **RETURN VALUE**

26256 TMO Upon successful completion, the *mq_send*( ) and *mq_timedsend*( ) functions shall return a value of
26257 zero. Otherwise, no message shall be enqueued, the functions shall return –1, and *errno* shall be
26258 set to indicate the error.

26259 **ERRORS**

26260 TMO The *mq_send*( ) and *mq_timedsend*( ) functions shall fail if:

26261 [EAGAIN] The O_NONBLOCK flag is set in the message queue description associated with *mqdes*, and the specified message queue is full.

26263 [EBADF] The *mqdes* argument is not a valid message queue descriptor open for writing.

26264 TMO [EINTR] A signal interrupted the call to *mq_send*( ) or *mq_timedsend*( ).

26265 [EINVAL] The value of *msg_prio* was outside the valid range.

26266 TMO [EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1 000 million.

26269 [EMSGSIZE] The specified message length, *msg_len*, exceeds the message size attribute of the message queue.

26271 TMO [ETIMEDOUT] The O_NONBLOCK flag was not set when the message queue was opened, but the timeout expired before the message could be added to the queue.

26273 **EXAMPLES**

26274 None.

26275 **APPLICATION USAGE**

26276 The value of the symbol {MQ_PRIO_MAX} limits the number of priority levels supported by the
26277 application. Message priorities range from 0 to {MQ_PRIO_MAX}–1.

26278 **RATIONALE**

26279 None.

26280 **FUTURE DIRECTIONS**

26281 None.

26282 **SEE ALSO**

26283 *mq_open*( ), *mq_receive*( ), *mq_setattr*( ), *mq_timedreceive*( ), *time*( ), the Base Definitions volume of
26284 IEEE Std. 1003.1-200x, **<mqueue.h>**, **<time.h>**

26285 **CHANGE HISTORY**

26286 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

26287 **Issue 6**

26288 The *mq_send*( ) function is marked as part of the Message Passing option.

26289 The [ENOSYS] error condition has been removed as stubs need not be provided if an
26290 implementation does not support the Message Passing option.

26291 The *mq_timedsend*( ) function is added for alignment with IEEE Std. 1003.1d-1999.

NAME

mq_setattr — set message queue attributes (**REALTIME**)

SYNOPSIS

MSG
```
#include <mqueue.h>

int mq_setattr(mqd_t mqdes, const struct mq_attr *restrict mqstat,
    struct mq_attr *restrict omqstat);
```

DESCRIPTION

The *mq_setattr*( ) function is used to set attributes associated with the open message queue description referenced by the message queue descriptor specified by *mqdes*.

The message queue attributes corresponding to the following members defined in the **mq_attr** structure are set to the specified values upon successful completion of *mq_setattr*( ):

mq_flags The value of this member is the bitwise-logical OR of zero or more of O_NONBLOCK and any implementation-defined flags.

The values of the *mq_maxmsg*, *mq_msgsize*, and *mq_curmsgs* members of the **mq_attr** structure are ignored by *mq_setattr*( ).

If *omqstat* is non-NULL, the *mq_setattr*( ) function stores, in the location referenced by *omqstat*, the previous message queue attributes and the current queue status. These values are the same as would be returned by a call to *mq_getattr*( ) at that point.

RETURN VALUE

Upon successful completion, the function shall return a value of zero and the attributes of the message queue shall have been changed as specified.

Otherwise, the message queue attributes shall be unchanged, and the function shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *mq_setattr*( ) function shall fail if:

[EBADF] The *mqdes* argument is not a valid message queue descriptor.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_open( ), *mq_send*( ), *mq_timedsend*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<mqueue.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mq_setattr*( ) 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.

The **restrict** keyword is added to the *mq_setattr*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

26340 **NAME**

26341 mq_timedreceive — receive a message from a message queue (**REALTIME**)

26342 **SYNOPSIS**

26343 MSG TMO
```
#include <mqueue.h>
#include <time.h>

int mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr,
    size_t msg_len, unsigned *restrict msg_prio,
    const struct timespec *restrict abs_timeout);
```
26348

26349 **DESCRIPTION**

26350 Refer to *mq_receive*( ).

26351 **NAME**

26352 mq_timedsend — send a message to a message queue (**REALTIME**)

26353 **SYNOPSIS**

26354 MSG TMO
```
#include <mqueue.h>
#include <time.h>

int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
    unsigned msg_prio, const struct timespec *abs_timeout);
```
26358

26359 **DESCRIPTION**

26360 Refer to *mq_send*( ).

26361 **NAME**

26362 mq_unlink — remove a message queue (**REALTIME**)

26363 **SYNOPSIS**

26364 MSG
```
#include <mqueue.h>
```

26365
```
int mq_unlink(const char *name);
```

26366

26367 **DESCRIPTION**

26368 The *mq_unlink*( ) function shall remove the message queue named by the path name *name*. After
26369 a successful call to *mq_unlink*( ) with *name*, a call to *mq_open*( ) with *name* fails if the flag
26370 O_CREAT is not set in *flags*. If one or more processes have the message queue open when
26371 *mq_unlink*( ) is called, destruction of the message queue is postponed until all references to the |
26372 message queue have been closed. |

26373 Calls to *mq_open*( ) to recreate the message queue may fail until the message queue is actually |
26374 removed. However, the *mq_unlink*( ) call need not block until all references have been closed; it
26375 may return immediately.

26376 **RETURN VALUE**

26377 Upon successful completion, the function shall return a value of zero. Otherwise, the named
26378 message queue shall be unchanged by this function call, and the function shall return a value of
26379 –1 and set *errno* to indicate the error.

26380 **ERRORS**

26381 The *mq_unlink*( ) function shall fail if:

26382 [EACCES] Permission is denied to unlink the named message queue. |

26383 [ENAMETOOLONG] |
26384 The length of the *name* argument exceeds {PATH_MAX} or a path name |
26385 component is longer than {NAME_MAX}. |

26386 [ENOENT] The named message queue does not exist. |

26387 **EXAMPLES**

26388 None.

26389 **APPLICATION USAGE**

26390 None.

26391 **RATIONALE**

26392 None.

26393 **FUTURE DIRECTIONS**

26394 None.

26395 **SEE ALSO**

26396 *mq_close*( ), *mq_open*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of |
26397 IEEE Std. 1003.1-200x, **<mqueue.h>** |

26398 **CHANGE HISTORY**

26399 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

26400 **Issue 6**

26401 The *mq_unlink*( ) function is marked as part of the Message Passing option. |

26402 The Open Group corrigenda item U021/5 has been applied, clarifying that upon unsuccessful
26403 completion, the named message queue is unchanged by this function.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

26406 **NAME**

26407 mrand48 — generate uniformly distributed pseudo-random signed long integers

26408 **SYNOPSIS**

26409 XSI
```
#include <stdlib.h>

long mrand48(void);
```
26410 |

26411 |

26412 **DESCRIPTION**

26413 Refer to *drand48*( ).

NAME

msgctl — XSI message control operations

SYNOPSIS

XSI
```
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

DESCRIPTION

The *msgctl*( ) function operates on XSI message queues (see the Base Definitions volume of IEEE Std. 1003.1-200x, Section 3.226, Message Queue). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 543).

The *msgctl*( ) function shall provide message control operations as specified by *cmd*. The following values for *cmd*, and the message control operations they specify, are:

IPC_STAT
: Place the current value of each member of the **msqid_ds** data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in **<sys/msg.h>**.

IPC_SET
: Set the value of the following members of the **msqid_ds** data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

  ```
  msg_perm.uid
  msg_perm.gid
  msg_perm.mode
  msg_qbytes
  ```

  IPC_SET can only be executed by a process with appropriate privileges or that has an effective user ID equal to the value of **msg_perm.cuid** or **msg_perm.uid** in the **msqid_ds** data structure associated with *msqid*. Only a process with appropriate privileges can raise the value of *msg_qbytes*.

IPC_RMID
: Remove the message queue identifier specified by *msqid* from the system and destroy the message queue and **msqid_ds** data structure associated with it. IPC_RMD can only be executed by a process with appropriate privileges or one that has an effective user ID equal to the value of **msg_perm.cuid** or **msg_perm.uid** in the **msqid_ds** data structure associated with *msqid*.

RETURN VALUE

Upon successful completion, *msgctl*( ) shall return 0; otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *msgctl*( ) function shall fail if:

[EACCES]
: The argument *cmd* is IPC_STAT and the calling process does not have read permission; see Section 2.7 (on page 541).

[EINVAL]
: The value of *msqid* is not a valid message queue identifier; or the value of *cmd* is not a valid command.

[EPERM]
: The argument *cmd* is 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 **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*.

[EPERM] The argument *cmd* is IPC_SET, an attempt is being made to increase to the value of *msg_qbytes*, and the effective user ID of the calling process does not have appropriate privileges.

EXAMPLES

None.

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 (on page 541) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_close( ), *mq_getattr*( ), *mq_notify*( ), *mq_open*( ), *mq_receive*( ), *mq_send*( ), *mq_setattr*( ), *mq_unlink*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/msg.h>**, Section 2.7 (on page 541)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 4

The function is no longer marked as OPTIONAL FUNCTIONALITY.

Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the SYNOPSIS section.

A FUTURE DIRECTIONS section is added warning application developers about migration to IEEE 1003.4 interfaces for interprocess communication.

The [ENOSYS] error is removed from the ERRORS section.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

NAME

msgget — get the XSI message queue identifier

SYNOPSIS

XSI
```
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

DESCRIPTION

The *msgget*( ) function operates on XSI message queues (see the Base Definitions volume of IEEE Std. 1003.1-200x, Section 3.226, Message Queue). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 543).

The *msgget*( ) function shall return the message queue identifier associated with the argument *key*.

A message queue identifier, associated message queue, and data structure (see **<sys/msg.h>**), are created for the argument *key* if one of the following is true:

- The argument *key* is equal to IPC_PRIVATE.
- The argument *key* does not already have a message queue identifier associated with it, and (*msgflg* & IPC_CREAT) is non-zero.

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

- **msg_perm.cuid**, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of **msg_perm.mode** are set equal to the low-order 9 bits of *msgflg*.
- **msg_qnum**, **msg_lspid**, **msg_lrpid**, **msg_stime**, and **msg_rtime** are set equal to 0.
- **msg_ctime** is set equal to the current time.
- *msg_qbytes* is set equal to the system limit.

RETURN VALUE

Upon successful completion, *msgget*( ) shall return a non-negative integer, namely a message queue identifier. Otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *msgget*( ) function shall fail if:

[EACCES] A message queue identifier exists for the argument *key*, but operation permission as specified by the low-order 9 bits of *msgflg* would not be granted; see Section 2.7 (on page 541).

[EEXIST] A message queue identifier exists for the argument *key* but ((*msgflg* & IPC_CREAT) && (*msgflg* & IPC_EXCL)) is non-zero.

[ENOENT] A message queue identifier does not exist for the argument *key* and (*msgflg* & IPC_CREAT) is 0.

[ENOSPC] A message queue identifier is to be created but the system-imposed limit on the maximum number of allowed message queue identifiers system-wide would be exceeded.

EXAMPLES

None.

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 (on page 541) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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>**, Section 2.7 (on page 541)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 4

The function is no longer marked as OPTIONAL FUNCTIONALITY.

Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the SYNOPSIS section.

The [ENOSYS] error is removed from the ERRORS section.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

NAME

msgrcv — XSI message receive operation

SYNOPSIS

XSI
```
#include <sys/msg.h>

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
    int msgflg);
```

DESCRIPTION

The *msgrcv*( ) function operates on XSI message queues (see the Base Definitions volume of IEEE Std. 1003.1-200x, Section 3.226, Message Queue). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 543).

The *msgrcv*( ) function shall read a message from the queue associated with the message queue identifier specified by *msqid* and place it in the user-defined buffer pointed to by *msgp*.

The application shall ensure that the argument *msgp* points to a user-defined buffer that contains first a field of type **long** specifying the type of the message, and then a data portion that holds the data bytes of the message. The structure below is an example of what this user-defined buffer might look like:

```
struct mymsg {
    long    mtype;     /* Message type. */
    char    mtext[1];  /* Message text. */
}
```

The structure member *mtype* is the received message's type as specified by the sending process.

The structure member *mtext* is the text of the message.

The argument *msgsz* specifies the size in bytes of *mtext*. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & MSG_NOERROR) is non-zero. The truncated part of the message is lost and no indication of the truncation is given to the calling process.

If the value of *msgsz* is greater than {SSIZE_MAX}, the result is implementation-defined.

The argument *msgtyp* specifies the type of message requested as follows:

- If *msgtyp* is 0, the first message on the queue is received.
- If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.
- If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

The argument *msgflg* specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

- If (*msgflg* & IPC_NOWAIT) is non-zero, the calling thread shall return immediately with a return value of –1 and *errno* set to [ENOMSG].
- If (*msgflg* & IPC_NOWAIT) is 0, the calling thread shall suspend execution until one of the following occurs:
  - — A message of the desired type is placed on the queue.
  - — The message queue identifier *msqid* is removed from the system; when this occurs, *errno* shall be set equal to [EIDRM] and –1 shall be returned.

— The calling thread receives a signal that is to be caught; in this case a message is not received and the calling thread resumes execution in the manner prescribed in *sigaction*( ).

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*:

- **msg_qnum** is decremented by 1.
- **msg_lrpid** is set equal to the process ID of the calling process.
- **msg_rtime** is set equal to the current time.

RETURN VALUE

Upon successful completion, *msgrcv*( ) shall return a value equal to the number of bytes actually placed into the buffer *mtext*. Otherwise, no message shall be received, *msgrcv*( ) shall return (**ssize_t**)–1, and *errno* shall be set to indicate the error.

ERRORS

The *msgrcv*( ) function shall fail if:

[E2BIG]	The value of *mtext* is greater than *msgsz* and (*msgflg* & MSG_NOERROR) is 0.
[EACCES]	Operation permission is denied to the calling process; see Section 2.7 (on page 541).
[EIDRM]	The message queue identifier *msqid* is removed from the system.
[EINTR]	The *msgrcv*( ) function was interrupted by a signal.
[EINVAL]	*msqid* is not a valid message queue identifier.
[ENOMSG]	The queue does not contain a message of the desired type and (*msgflg* & IPC_NOWAIT) is non-zero.

EXAMPLES

Receiving a Message

The following example receives the first message on the queue (based on the value of the *msgtype* argument, 0). The queue is identified by the *msqid* argument (assuming that the value has previously been set). This call specifies that an error should be reported if no message is available, but not if the message is too large. The message size is calculated directly using the *sizeof* operator.

```
#include <sys/msg.h>
...
int result;
int msqid;
struct message {
    long type;
    char text[20];
} msg;
long msgtyp = 0;
...
result = msgrcv(msqid, (void *) &msg, sizeof(msg.text),
         msgtyp, MSG_NOERROR | IPC_NOWAIT);
```

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 (on page 541) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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>**, Section 2.7 (on page 541)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 4

The function is no longer marked as OPTIONAL FUNCTIONALITY.

Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the SYNOPSIS section.

The [ENOSYS] error is removed from the ERRORS section.

A FUTURE DIRECTIONS section is added warning application developers about migration to IEEE 1003.4 interfaces for interprocess communication.

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 the {SSIZE_MAX}.

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to the APPLICATION USAGE section.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

msgsnd — XSI message send operation

SYNOPSIS

XSI
```
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

DESCRIPTION

The *msgsnd*( ) function operates on XSI message queues (see the Base Definitions volume of IEEE Std. 1003.1-200x, Section 3.226, Message Queue). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 543).

The *msgsnd*( ) function is used to send a message to the queue associated with the message queue identifier specified by *msqid*.

The application shall ensure that the argument *msgp* points to a user-defined buffer that contains first a field of type **long** specifying the type of the message, and then a data portion that holds the data bytes of the message. The structure below is an example of what this user-defined buffer might look like:

```
struct mymsg {
    long   mtype;       /* Message type. */
    char   mtext[1];    /* Message text. */
}
```

The structure member *mtype* is a non-zero positive type **long** that can be used by the receiving process for message selection.

The structure member *mtext* is any text of length *msgsz* bytes. The argument *msgsz* can range from 0 to a system-imposed maximum.

The argument *msgflg* specifies the action to be taken if one or more of the following are true:

- The number of bytes already on the queue is equal to *msg_qbytes*; see **<sys/msg.h>**.
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

- If (*msgflg* & IPC_NOWAIT) is non-zero, the message shall not be sent and the calling thread shall return immediately.
- If (*msgflg* & IPC_NOWAIT) is 0, the calling thread shall suspend execution until one of the following occurs:
  - — The condition responsible for the suspension no longer exists, in which case the message is sent.
  - — The message queue identifier *msqid* is removed from the system; when this occurs, *errno* shall be set equal to [EIDRM] and –1 shall be returned.
  - — The calling thread receives a signal that is to be caught; in this case the message is not sent and the calling thread resumes execution in the manner prescribed in *sigaction*( ).

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*; see **<sys/msg.h>**:

- **msg_qnum** is incremented by 1.
- **msg_lspid** is set equal to the process ID of the calling process.
- **msg_stime** is set equal to the current time.

RETURN VALUE

Upon successful completion, *msgsnd*( ) shall return 0; otherwise, no message shall be sent, *msgsnd*( ) shall return –1, and *errno* shall be set to indicate the error.

ERRORS

The *msgsnd*( ) function shall fail if:

[EACCES] Operation permission is denied to the calling process; see Section 2.7 (on page 541).

[EAGAIN] The message cannot be sent for one of the reasons cited above and (*msgflg* & IPC_NOWAIT) is non-zero.

[EIDRM] The message queue identifier *msgid* is removed from the system.

[EINTR] The *msgsnd*( ) function was interrupted by a signal.

[EINVAL] The value of *msqid* is not a valid message queue identifier, or the value of *mtype* is less than 1; or the value of *msgsz* is less than 0 or greater than the system-imposed limit.

EXAMPLES

Sending a Message

The following example sends a message to the queue identified by the *msqid* argument (assuming that value has previously been set). This call specifies that an error should be reported if no message is available. The message size is calculated directly using the *sizeof* operator.

```
#include <sys/msg.h>
...
int result;
int msqid;
struct message {
    long type;
    char text[20];
} msg;

msg.type = 1;
strcpy(msg.text, "This is message 1");
...
result = msgsnd(msqid, (void *) &msg, sizeof(msg.text), IPC_NOWAIT);
```

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 (on page 541) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_close(), *mq_getattr*(), *mq_notify*(), *mq_open*(), *mq_receive*(), *mq_send*(), *mq_setattr*(), *mq_unlink*(), *msgctl*(), *msgget*(), *msgrcv*(), *sigaction*(), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/msg.h>**, Section 2.7 (on page 541)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 4

The function is no longer marked as OPTIONAL FUNCTIONALITY.

Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the SYNOPSIS section. Also the type of argument *msgp* is changed from **void***** to **const void***.

In the DESCRIPTION, the example of a message buffer is changed:

- Explicitly to define the first member as being of type **long**
- To define the size of the message array *mtext*

The [ENOSYS] error is removed from the ERRORS section.

A FUTURE DIRECTIONS section is added warning application developers about migration to IEEE 1003.4 interfaces for interprocess communication.

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

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

msync — synchronize memory with physical storage

SYNOPSIS

MF SIO
```
#include <sys/mman.h>

int msync(void *addr, size_t len, int flags);
```

DESCRIPTION

The *msync*( ) function shall write all modified data to permanent storage locations, if any, in those whole pages containing any part of the address space of the process starting at address *addr* and continuing for *len* bytes. If no such storage exists, *msync*( ) need not have any effect. If requested, the *msync*( ) function then invalidates cached copies of data.

The implementation shall require that *addr* be a multiple of the page size as returned by *sysconf*( ).

For mappings to files, the *msync*( ) function ensures that all write operations are completed as defined for synchronized I/O data integrity completion. It is unspecified whether the implementation also writes out other file attributes. When the *msync*( ) function is called on MAP_PRIVATE mappings, any modified data shall not be written to the underlying object and shall not cause such data to be made visible to other processes. It is unspecified whether data in MAP_PRIVATE mappings has any permanent storage locations. The effect of *msync*( ) on a shared memory object or a typed memory object is unspecified. The behavior of this function is unspecified if the mapping was not established by a call to *mmap*( ).

SHM|TYM

The *flags* argument is constructed from the bitwise-inclusive OR of one or more of the following flags defined in the header **<sys/mman.h>**:

Symbolic Constant	Description
MS_ASYNC	Perform asynchronous writes.
MS_SYNC	Perform synchronous writes.
MS_INVALIDATE	Invalidate cached data.

When MS_ASYNC is specified, *msync*( ) shall return immediately once all the write operations are initiated or queued for servicing; when MS_SYNC is specified, *msync*( ) shall not return until all write operations are completed as defined for synchronized I/O data integrity completion. Either MS_ASYNC or MS_SYNC is specified, but not both.

When MS_INVALIDATE is specified, *msync*( ) invalidates all cached copies of mapped data that are inconsistent with the permanent storage locations such that subsequent references shall obtain data that was consistent with the permanent storage locations sometime between the call to *msync*( ) and the first subsequent memory reference to the data.

If *msync*( ) causes any write to a file, the file's *st_ctime* and *st_mtime* fields shall be marked for update.

RETURN VALUE

Upon successful completion, *msync*( ) shall return 0; otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *msync*( ) function shall fail if:

[EBUSY] Some or all of the addresses in the range starting at *addr* and continuing for *len* bytes are locked, and MS_INVALIDATE is specified.

[EINVAL] The value of *flags* is invalid.

[EINVAL] The value of *addr* is not a multiple of the page size, {PAGESIZE}.

[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.

EXAMPLES

None.

APPLICATION USAGE

The *msync*( ) function is only supported if the Memory Mapped Files option and the Synchronized Input and Output option are supported, and thus need not be available on all implementations.

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.

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.

RATIONALE

The *msync*( ) function is used to write out data in a mapped region to the permanent storage for the underlying object. The call to *msync*( ) ensures data integrity of the file.

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.

FUTURE DIRECTIONS

None.

SEE ALSO

mmap( ), *sysconf*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Aligned with *msync*( ) in the POSIX Realtime Extension as follows:

- The DESCRIPTION is extensively reworded.
- [EBUSY] and a new form of [EINVAL] are added as mandatory error conditions.

Issue 6

The *msync*( ) function is marked as part of the Memory Mapped Files and Synchronized Input and Output options.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [EBUSY] mandatory error condition is added.

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 second [EINVAL] error condition is made mandatory.

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

26858 **NAME**

26859 munlock — unlock a range of process address space

26860 **SYNOPSIS**

26861 MLR
```
#include <sys/mman.h>
```

26862
```
int munlock(const void * addr, size_t len);
```

26863

26864 **DESCRIPTION**

26865 Refer to *mlock*( ).

26866 **NAME**

26867 munlockall — unlock the address space of a process

26868 **SYNOPSIS**

26869 ML
```
#include <sys/mman.h>
```

26870
```
int munlockall(void);
```

26871

26872 **DESCRIPTION**

26873 Refer to *mlockall*( ).

NAME

munmap — unmap pages of memory

SYNOPSIS

MF|SHM
```
#include <sys/mman.h>

int munmap(void *addr, size_t len);
```

DESCRIPTION

The *munmap*( ) function shall remove any mappings for those entire pages containing any part of the address space of the process starting at *addr* and continuing for *len* bytes. Further references to these pages result in the generation of a SIGSEGV signal to the process. If there are no mappings in the specified address range, then *munmap*( ) has no effect.

The implementation shall require that *addr* be a multiple of the page size {PAGESIZE}.

If a mapping to be removed was private, any modifications made in this address range shall be discarded.

ML|MLR Any memory locks (see *mlock*( ) and *mlockall*( )) associated with this address range shall be removed, as if by an appropriate call to *munlock*( ).

TYM If a mapping removed from a typed memory object causes the corresponding address range of the memory pool to be inaccessible by any process in the system except through allocatable mappings (that is, mappings of typed memory objects opened with the POSIX_TYPED_MEM_MAP_ALLOCATABLE flag), then that range of the memory pool shall become deallocated and may become available to satisfy future typed memory allocation requests.

A mapping removed from a typed memory object opened with the POSIX_TYPED_MEM_MAP_ALLOCATABLE flag shall not affect in any way the availability of that typed memory for allocation.

The behavior of this function is unspecified if the mapping was not established by a call to *mmap*( ).

RETURN VALUE

Upon successful completion, *munmap*( ) shall return 0; otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *munmap*( ) function shall fail if:

[EINVAL] Addresses in the range [*addr*,*addr*+*len*) are outside the valid range for the address space of a process.

[EINVAL] The *len* argument is 0.

[EINVAL] The *addr* argument is not a multiple of the page size as returned by *sysconf*( ).

EXAMPLES

None.

APPLICATION USAGE

The *munmap*( ) function is only supported if the Memory Mapped Files option or the Shared Memory Objects option is supported.

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.

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.

NAME

nan, nanf, nanl — return quiet NaN

SYNOPSIS

```
#include <math.h>

double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
```

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 function call *nan*("*n-char-sequence*") shall be equivalent to:

```
strtod("NAN(n-char-sequence)", (char **) NULL);
```

The function call *nan*("") shall be equivalent to:

```
strtod("NAN()", (char **) NULL)
```

If *tagp* does not point to an *n*-**char** sequence or an empty string, the function call shall be equivalent to:

```
strtod("NAN", (char **) NULL)
```

Function calls to *nanf*( ) and *nanl*( ) are equivalent to the corresponding function calls to *strtof*( ) and *strtold*( ).

RETURN VALUE

These functions shall return a quiet NaN, if available, with content indicated through *tagp*.

If the implementation does not support quiet NaNs, these functions shall return zero.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strtod( ), <REFERENCE UNDEFINED>(strtold), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

nanosleep — high resolution sleep (**REALTIME**)

SYNOPSIS

TMR
```
#include <time.h>

int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```

DESCRIPTION

The *nanosleep*( ) function shall cause the current thread to be suspended from execution until either the time interval specified by the *rqtp* argument has elapsed or a signal is delivered to the calling thread, and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time shall not be less than the time specified by *rqtp*, as measured by the system clock, CLOCK_REALTIME.

The use of the *nanosleep*( ) function has no effect on the action or blockage of any signal.

RETURN VALUE

If the *nanosleep*( ) function returns because the requested time has elapsed, its return value shall be zero.

If the *nanosleep*( ) function returns because it has been interrupted by a signal, the function shall return a value of –1 and set *errno* to indicate the interruption. If the *rmtp* argument is non-NULL, the **timespec** structure referenced by it is updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). If the *rmtp* argument is NULL, the remaining time is not returned.

If *nanosleep*( ) fails, it shall return a value of –1 and set *errno* to indicate the error.

ERRORS

The *nanosleep*( ) function shall fail if:

[EINTR]	The *nanosleep*( ) function was interrupted by a signal.
[EINVAL]	The *rqtp* argument specified a nanosecond value less than zero or greater than or equal to 1000 million.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

It is common to suspend execution of a process for an interval in order to poll the status of a non-interrupting function. A large number of actual needs can be met with a simple extension to *sleep*( ) that provides finer resolution.

In the POSIX.1-1990 standard and SVR4, it is possible to implement such a routine, but the frequency of wakeup is limited by the resolution of the *alarm*( ) and *sleep*( ) functions. In 4.3 BSD, it is possible to write such a routine using no static storage and reserving no system facilities. Although it is possible to write a function with similar functionality to *sleep*( ) using the remainder of the timers function, such a function requires the use of signals and the reservation of some signal number. This volume of IEEE Std. 1003.1-200x requires that *nanosleep*( ) be non-intrusive of the signals function.

The *nanosleep*( ) function shall return a value of 0 on success and –1 on failure or if interrupted. This latter case is different from *sleep*( ). This was done because the remaining time is returned via an argument structure pointer, *rmtp*, instead of as the return value.

FUTURE DIRECTIONS

None.

SEE ALSO

sleep( ), 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 *nanosleep*( ) function is marked as part of the Timers option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

27038 **NAME**

27039 nearbyint, nearbyintf, nearbyintl — floating-point rounding functions

27040 **SYNOPSIS**

27041 `#include <math.h>`

27042 `double nearbyint(double x);`
27043 `float nearbyintf(float x);`
27044 `long double nearbyintl(long double x);`

27045 **DESCRIPTION**

27046 CX The functionality described on this reference page is aligned with the ISO C standard. Any
27047 conflict between the requirements described here and the ISO C standard is unintentional. This
27048 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

27049 These functions shall round their argument to an integer value in floating-point format, using
27050 the current rounding direction and without raising the inexact floating-point exception.

27051 An application wishing to check for error situations should set *errno* to 0 before calling these
27052 functions. If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

27053 **RETURN VALUE**

27054 Upon successful completion, these functions shall return the rounded integer value.

27055 If *x* is ±Inf, these functions shall return *x*.

27056 If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

27057 **ERRORS**

27058 These functions may fail if:

27059 [EDOM] The value of *x* is NaN.

27060 **EXAMPLES**

27061 None.

27062 **APPLICATION USAGE**

27063 None.

27064 **RATIONALE**

27065 None.

27066 **FUTURE DIRECTIONS**

27067 None.

27068 **SEE ALSO**

27069 The Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

27070 **CHANGE HISTORY**

27071 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

nextafter, nextafterf, nextafterl, nexttoward, nexttowardf, nexttowardl — next representable double-precision floating-point number

SYNOPSIS

XSI
```
#include <math.h>

double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double 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 *nextafter*( ), *nextafterf*( ), and *nextafterl*( ) functions shall compute the next representable double-precision floating-point value following *x* in the direction of *y*. Thus, if *y* is less than *x*, *nextafter*( ) returns the largest representable floating-point number less than *x*. The *nextafter*( ), *nextafterf*( ), and *nextafterl*( ) functions shall return *y* if *x* equals *y*.

The *nexttoward*( ), *nexttowardf*( ), and *nexttowardl*( ) functions are equivalent to the corresponding *nextafter*( ) functions, except that the second parameter has type **long double** and the functions return *y* converted to the type of the function if *x* equals *y*.

An application wishing to check for error situations should set *errno* to 0 before calling *nextafter*( ). If *errno* is non-zero on return, or the value NaN is returned, an error has occurred.

RETURN VALUE

These functions shall return the next representable double-precision floating-point value following *x* in the direction of *y*. The *nextafter*( ), *nextafterf*( ), and *nextafterl*( ) functions shall return *y* if *x* equals *y*.

If *x* or *y* is NaN, then *nextafter*( ) shall return NaN and may set *errno* to [EDOM].

If *x* is finite and the correct function value would overflow, HUGE_VAL shall be returned and *errno* set to [ERANGE].

ERRORS

These functions shall fail if:

[ERANGE] The correct value would overflow.

These functions may fail if:

[EDOM] The *x* or *y* argument is NaN.

27109 **EXAMPLES**

27110 None.

27111 **APPLICATION USAGE**

27112 None.

27113 **RATIONALE**

27114 None.

27115 **FUTURE DIRECTIONS**

27116 None.

27117 **SEE ALSO**

27118 The Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

27119 **CHANGE HISTORY**

27120 First released in Issue 4, Version 2.

27121 **Issue 5**

27122 Moved from X/OPEN UNIX extension to BASE.

27123 **Issue 6**

27124 The *nextafterf*( ), *nextafterl*( ), *nexttoward*( ), *nexttowardf*( ), *nexttowardl*( ) functions are added for
27125 alignment with the ISO/IEC 9899: 1999 standard.

NAME

nftw — walk a file tree

SYNOPSIS

XSI

```
#include <ftw.h>

int nftw(const char *path, int (*fn)(const char *,
    const struct stat *, int, struct FTW *), int depth, int flags);
```

DESCRIPTION

The *nftw*() function shall recursively descend the directory hierarchy rooted in *path*. The *nftw*() function has a similar effect to *ftw*() except that it takes an additional argument *flags*, which is a bitwise-inclusive OR of zero or more of the following flags:

FTW_CHDIR — If set, *nftw*() shall change the current working directory to each directory as it reports files in that directory. If clear, *nftw*() shall not change the current working directory.

FTW_DEPTH — If set, *nftw*() shall report all files in a directory before reporting the directory itself. If clear, *nftw*() shall report any directory before reporting the files in that directory.

FTW_MOUNT — If set, *nftw*() shall only report files in the same file system as *path*. If clear, *nftw*() shall report all files encountered during the walk.

FTW_PHYS — If set, *nftw*() shall perform a physical walk and shall not follow symbolic links.

If FTW_PHYS is clear and FTW_DEPTH is set, *nftw*() shall follow links instead of reporting them, but shall not report any directory that would be a descendant of itself. If FTW_PHYS is clear and FTW_DEPTH is clear, *nftw*() shall follow links instead of reporting them, but shall not report the contents of any directory that would be a descendant of itself.

At each file it encounters, *nftw*() shall call the user-supplied function *fn* with four arguments:

- The first argument is the path name of the object.
- The second argument is a pointer to the **stat** buffer containing information on the object.
- The third argument is an integer giving additional information. Its value is one of the following:

  FTW_F — The object is a file.

  FTW_D — The object is a directory.

  FTW_DP — The object is a directory and subdirectories have been visited. (This condition shall only occur if the FTW_DEPTH flag is included in *flags*.)

  FTW_SL — The object is a symbolic link. (This condition shall only occur if the FTW_PHYS flag is included in *flags*.)

  FTW_SLN — The object is a symbolic link that does not name an existing file. (This condition shall only occur if the FTW_PHYS flag is not included in *flags*.)

  FTW_DNR — The object is a directory that cannot be read. The *fn* function shall not be called for any of its descendants.

  FTW_NS — The *stat*() function failed on the object because of lack of appropriate permission. The **stat** buffer passed to *fn* is undefined. Failure of *stat*() for any other reason is considered an error and *nftw*() shall return –1.

- The fourth argument is a pointer to an **FTW** structure. The value of **base** is the offset of the object's file name in the path name passed as the first argument to *fn*. The value of **level** indicates depth relative to the root of the walk, where the root level is 0.

The argument *depth* sets the maximum number of file descriptors that are shall be by *nftw*() while traversing the file tree. At most one file descriptor shall be used for each directory level.

RETURN VALUE

The *nftw*() function shall continue until the first of the following conditions occurs:

- An invocation of *fn* shall return a non-zero value, in which case *nftw*() shall return that value.
- The *nftw*() function detects an error other than [EACCES] (see FTW_DNR and FTW_NS above), in which case *nftw*() shall return –1 and set *errno* to indicate the error.
- The tree is exhausted, in which case *nftw*() shall return 0.

ERRORS

The *nftw*() function shall fail if:

[EACCES] Search permission is denied for any component of *path* or read permission is denied for *path*, or *fn* returns –1 and does not reset *errno*.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

[ENAMETOOLONG]
The length of the *path* argument exceeds {PATH_MAX} or a path name component is longer than {NAME_MAX}.

[ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

[ENOTDIR] A component of *path* is not a directory.

The *nftw*() function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

[ENAMETOOLONG]
Path name resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

[ENFILE] Too many files are currently open in the system.

In addition, *errno* may be set if the function pointed by *fn* causes *errno* to be set.

EXAMPLES

The following example walks the **/tmp** directory and its subdirectories, calling the *nftw*() function for every directory entry, to a maximum of 5 levels deep.

```
#include <ftw.h>
...
int nftwfunc(const char *, const struct stat *, int, struct FTW *);

int nftwfunc(const char *filename, const struct stat *statptr,
    int fileflags, struct FTW *pfwt)
{
    return 0;
}
```

```
        ...
        char *startpath = "/tmp";
        int depth = 5;
        int flags = FTW_CHDIR | FTW_DEPTH | FTW_MOUNT;
        int ret;

        ret = nftw(startpath, nftwfunc, depth, flags);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

lstat( ), *opendir*( ), *readdir*( ), *stat*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ftw.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

In the DESCRIPTION, the definition of the *depth* argument is clarified.

Issue 6

The Open Group Base Resolution bwg97-003 is applied.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

27233 **NAME**

27234 nice — change the nice value of a process

27235 **SYNOPSIS**

27236 XSI
```
#include <unistd.h>
```

27237
```
int nice(int incr);
```

27238

27239 **DESCRIPTION**

27240 The *nice*( ) function shall add the value of *incr* to the nice value of the calling process. A process'
27241 nice value is a non-negative number for which a more positive value results in less favorable
27242 scheduling.

27243 A maximum nice value of 2*{NZERO}–1 and a minimum nice value of 0 are imposed by the
27244 system. Requests for values above or below these limits result in the nice value being set to the
27245 corresponding limit. Only a process with appropriate privileges can lower the nice value.

27246 PS | TPS Calling the *nice*( ) function has no effect on the priority of processes or threads with policy
27247 SCHED_FIFO or SCHED_RR. The effect on processes or threads with other scheduling policies
27248 is implementation-defined. |

27249 The nice value set with *nice*( ) shall be applied to the process. If the process is multi-threaded, the
27250 nice value shall affect all system scope threads in the process.

27251 As –1 is a permissible return value in a successful situation, an application wishing to check for
27252 error situations should set *errno* to 0, then call *nice*( ), and if it returns –1, check to see if *errno* is
27253 non-zero.

27254 **RETURN VALUE**

27255 Upon successful completion, *nice*( ) shall return the new nice value –{NZERO}. Otherwise, –1
27256 shall be returned, the process' nice value shall not be changed, and *errno* shall be set to indicate
27257 the error.

27258 **ERRORS**

27259 The *nice*( ) function shall fail if:

27260 [EPERM] The *incr* argument is negative and the calling process does not have |
27261 appropriate privileges.

27262 **EXAMPLES**

27263 **Changing the Nice Value**

27264 The following example adds the value of the *incr* argument, –20, to the nice value of the calling
27265 process.

27266
```
#include <unistd.h>
```
27267
```
...
```
27268
```
int incr = -20;
```
27269
```
int ret;
```

27270
```
ret = nice(incr);
```

27271 **APPLICATION USAGE**

27272 None.

27273 **RATIONALE**

27274 None.

27275 **FUTURE DIRECTIONS**

27276 None.

27277 **SEE ALSO**

27278 The Base Definitions volume of IEEE Std. 1003.1-200x, **<limits.h>**, **<unistd.h>**

27279 **CHANGE HISTORY**

27280 First released in Issue 1. Derived from Issue 1 of the SVID.

27281 **Issue 4**

27282 The **<unistd.h>** header is added to the SYNOPSIS section.

27283 A statement is added to the DESCRIPTION indicating that the nice value can only be lowered by
27284 a process with appropriate privileges.

27285 **Issue 4, Version 2**

27286 The RETURN VALUE section is updated for X/OPEN UNIX conformance to define that the
27287 process' nice value is not changed if an error is detected.

27288 **Issue 5**

27289 A statement is added to the description indicating the effects of this function on the different
27290 scheduling policies and multi-threaded processes.

27291 **NAME**

27292 nl_langinfo — language information

27293 **SYNOPSIS**

27294 XSI
```
#include <langinfo.h>
```

27295
```
char *nl_langinfo(nl_item item);
```

27296

27297 **DESCRIPTION**

27298 The *nl_langinfo*( ) function shall return a pointer to a string containing information relevant to
27299 the particular language or cultural area defined in the program's locale (see **<langinfo.h>**). The
27300 manifest constant names and values of *item* are defined in **<langinfo.h>**. For example:

27301
```
nl_langinfo(ABDAY_1)
```

27302 would return a pointer to the string `"Dom"` if the identified language was Portuguese, and
27303 `"Sun"` if the identified language was English.

27304 Calls to *setlocale*( ) with a category corresponding to the category of *item* (see **<langinfo.h>**), or to
27305 the category *LC_ALL*, may overwrite the array pointed to by the return value.

27306 The *nl_langinfo*( ) function need not be reentrant. A function that is not required to be reentrant is
27307 not required to be thread-safe.

27308 **RETURN VALUE**

27309 In a locale where *langinfo* data is not defined, *nl_langinfo*( ) shall return a pointer to the
27310 corresponding string in the POSIX locale. In all locales, *nl_langinfo*( ) shall return a pointer to an
27311 empty string if *item* contains an invalid setting.

27312 This pointer may point to static data that may be overwritten on the next call.

27313 **ERRORS**

27314 No errors are defined.

27315 **EXAMPLES**

27316 **Getting Date and Time Formatting Information**

27317 The following example returns a pointer to a string containing date and time formatting
27318 information, as defined in the *LC_TIME* category of the current locale.

27319
```
#include <time.h>
```
27320
```
#include <langinfo.h>
```
27321
```
...
```
27322
```
strftime(datestring, sizeof(datestring), nl_langinfo(D_T_FMT), tm);
```
27323
```
...
```

27324 **APPLICATION USAGE**

27325 The array pointed to by the return value should not be modified by the program, but may be
27326 modified by further calls to *nl_langinfo*( ).

27327 **RATIONALE**

27328 None.

27329 **FUTURE DIRECTIONS**

27330 None.

27331 **SEE ALSO**

27332 *setlocale*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<langinfo.h>**, **<nl_types.h>**, the |
27333 Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale |

27334 **CHANGE HISTORY**

27335 First released in Issue 2.

27336 **Issue 4**

27337 The **<nl_types.h>** header is removed from the SYNOPSIS section.

27338 **Issue 5**

27339 The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section.

27340 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

27341 **NAME**

27342 nrand48 — generate uniformly distributed pseudo-random non-negative long integers

27343 **SYNOPSIS**

27344 XSI
```
#include <stdlib.h>

long nrand48(unsigned short xsubi[3]);
```
27345 |

27346 |

27347 **DESCRIPTION**

27348 Refer to *drand48*( ).

27349 **NAME**

27350 ntohl — convert values between host and network byte order

27351 **SYNOPSIS**

27352 `#include <arpa/inet.h>`

27353 `uint32_t ntohl(uint32_t` *`netlong`*`);`

27354 **DESCRIPTION**

27355 Refer to *htonl*( ).

27356 **NAME**

27357 ntohs — convert values between host and network byte order

27358 **SYNOPSIS**

27359 `#include <arpa/inet.h>`

27360 `uint16_t ntohs(uint16_t` *`netshort`*`);`

27361 **DESCRIPTION**

27362 Refer to *htonl*( ).

27363 **NAME**

27364 open — open a file

27365 **SYNOPSIS**

27366 OH
```
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int oflag, ... );
```

27369 **DESCRIPTION**

27370 The *open*( ) function establishes the connection between a file and a file descriptor. It creates an
27371 open file description that refers to a file and a file descriptor that refers to that open file
27372 description. The file descriptor is used by other I/O functions to refer to that file. The *path*
27373 argument points to a path name naming the file.

27374 The *open*( ) function shall return a file descriptor for the named file that is the lowest file
27375 descriptor not currently open for that process. The open file description is new, and therefore the
27376 file descriptor does not share it with any other process in the system. The FD_CLOEXEC file
27377 descriptor flag associated with the new file descriptor shall be cleared.

27378 The file offset used to mark the current position within the file shall be set to the beginning of the
27379 file.

27380 The file status flags and file access modes of the open file description shall be set according to
27381 the value of *oflag*.

27382 Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list,
27383 defined in **<fcntl.h>**. Applications shall specify exactly one of the first three values (file access
27384 modes) below in the value of *oflag*:

27385 O_RDONLY Open for reading only.

27386 O_WRONLY Open for writing only.

27387 O_RDWR Open for reading and writing. The result is undefined if this flag is applied to
27388 a FIFO.

27389 Any combination of the following may be used:

27390 O_APPEND If set, the file offset shall be set to the end of the file prior to each write.

27391 O_CREAT If the file exists, this flag has no effect except as noted under O_EXCL below.
27392 Otherwise, the file is created; the user ID of the file shall be set to the effective
27393 user ID of the process; the group ID of the file shall be set to the group ID of
27394 the file's parent directory or to the effective group ID of the process; and the
27395 access permission bits (see **<sys/stat.h>**) of the file mode shall be set to the
27396 value of the third argument taken as type **mode_t** modified as follows: a
27397 bitwise AND is performed on the file-mode bits and the corresponding bits in
27398 the complement of the process' file mode creation mask. Thus, all bits in the
27399 file mode whose corresponding bit in the file mode creation mask is set are
27400 cleared. When bits other than the file permission bits are set, the effect is
27401 unspecified. The third argument does not affect whether the file is open for
27402 reading, writing, or for both.

27403 SIO O_DSYNC Write I/O operations on the file descriptor complete as defined by
27404 synchronized I/O data integrity completion

27405 O_EXCL If O_CREAT and O_EXCL are set, *open*( ) shall fail if the file exists. The check
27406 for the existence of the file and the creation of the file if it does not exist shall
27407 be atomic with respect to other threads executing *open*( ) naming the same file |

name in the same directory with O_EXCL and O_CREAT set. If O_EXCL and O_CREAT are set, and *path* names a symbolic link, *open*( ) shall fail and set *errno* 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.

O_NOCTTY — If set and *path* identify a terminal device, *open*( ) shall not cause the terminal device to become the controlling terminal for the process.

O_NONBLOCK — When opening a FIFO with O_RDONLY or O_WRONLY set:

- If O_NONBLOCK is set, an *open*( ) for reading-only shall return without delay. An *open*( ) for writing-only shall return an error if no process currently has the file open for reading.
- If O_NONBLOCK is clear, an *open*( ) for reading-only shall block the calling thread until a thread opens the file for writing. An *open*( ) for writing-only shall block the calling thread until a thread opens the file for reading.

When opening a block special or character special file that supports non-blocking opens:

- If O_NONBLOCK is set, the *open*( ) function shall return without blocking for the device to be ready or available. Subsequent behavior of the device is device-specific.
- If O_NONBLOCK is clear, the *open*( ) function shall block the calling thread until the device is ready or available before returning.

Otherwise, the behavior of O_NONBLOCK is unspecified.

SIO O_RSYNC — Read I/O operations on the file descriptor 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 *oflag*, all I/O operations on the file descriptor 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 complete as defined by synchronized I/O file integrity completion.

SIO O_SYNC — Write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.

O_TRUNC — If the file exists and is a regular file, and the file is successfully opened O_RDWR or O_WRONLY, its length is truncated to 0, and the mode and owner are unchanged. It shall have no effect on FIFO special files or terminal device files. Its effect on other file types is implementation-defined. The result of using O_TRUNC with O_RDONLY is undefined.

If O_CREAT is set and the file did not previously exist, upon successful completion, *open*( ) shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file and the *st_ctime* and *st_mtime* fields of the parent directory.

If O_TRUNC is set and the file did previously exist, upon successful completion, *open*( ) shall mark for update the *st_ctime* and *st_mtime* fields of the file.

SIO If both the O_SYNC and O_DSYNC flags are set, the effect is as if only the O_SYNC flag was set.

XSR If *path* refers to a STREAMS file, *oflag* may be constructed from O_NONBLOCK OR'ed with either O_RDONLY, O_WRONLY, or O_RDWR. Other flag values are not applicable to STREAMS devices and have no effect on them. The value O_NONBLOCK affects the operation

of STREAMS drivers and certain functions applied to file descriptors associated with STREAMS files. For STREAMS drivers, the implementation of O_NONBLOCK is device-specific.

If *path* names the master side of a pseudo-terminal device, then it is unspecified whether *open*( ) locks the slave side so that it cannot be opened. Portable applications shall call *unlockpt*( ) before opening the slave side.

The largest value that can be represented correctly in an object of type **off_t** shall be established as the offset maximum in the open file description.

RETURN VALUE

Upon successful completion, the function shall open the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, –1 shall be returned and *errno* set to indicate the error. No files shall be created or modified if the function returns –1.

ERRORS

The *open*( ) function shall fail if:

[EACCES] Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *oflag* are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created, or O_TRUNC is specified and write permission is denied.

[EEXIST] O_CREAT and O_EXCL are set, and the named file exists.

[EINTR] A signal was caught during *open*( ).

SIO [EINVAL] The implementation does not support synchronized I/O for this file.

XSR [EIO] The *path* argument names a STREAMS file and a hangup or error occurred during the *open*( ).

[EISDIR] The named file is a directory and *oflag* includes O_WRONLY or O_RDWR.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

[EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

[ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a path name component is longer than {NAME_MAX}.

[ENFILE] The maximum allowable number of files is currently open in the system.

[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 *path* argument points to an empty string.

XSR [ENOSR] The *path* argument names a STREAMS-based file and the system is unable to allocate a STREAM.

[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.

[ENOTDIR] A component of the path prefix is not a directory.

[ENXIO] O_NONBLOCK is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.

[ENXIO] The named file is a character special or block special file, and the device associated with this special file does not exist.

[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**.

[EROFS] The named file resides on a read-only file system and either O_WRONLY, O_RDWR, O_CREAT (if file does not exist), or O_TRUNC is set in the *oflag* argument.

The *open*( ) function may fail if:

XSR [EAGAIN] The *path* argument names the slave side of a pseudo-terminal device that is locked.

[EINVAL] The value of the *oflag* argument is not valid.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted path name string exceeded {PATH_MAX}.

XSR [ENOMEM] The *path* argument names a STREAMS file and the system is unable to allocate resources.

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed and *oflag* is O_WRONLY or O_RDWR.

EXAMPLES

Opening a File for Writing by the Owner

The following example opens the file **/tmp/file**, either by creating it (if it does not already exist), or by truncating its length to 0 (if it does exist). In the former case, if the call creates a new file, the access permission bits in the file mode of the file are set to permit reading and writing by the owner, and to permit reading only by group members and others.

If the call to *open*( ) is successful, the file is opened for writing.

```
#include <fcntl.h>
...
int fd;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
char *filename = "/tmp/file";
...
fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, mode);
...
```

Opening a File Using an Existence Check

The following example uses the *open*( ) function to try to create the **LOCKFILE** file and open it for writing. Because the *open*( ) function specifies the O_EXCL flag, the call fails if the file already exists. In that case, the program assumes that someone else is updating the password file and exits.

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

```
#define LOCKFILE "/etc/ptmp"
...
int pfd; /* Integer for file descriptor returned by open() call. */
...
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);
}
...
```

Opening a File for Writing

The following example opens a file for writing, creating the file if it does not already exist. If the file does exist, the system truncates the file to zero bytes.

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

#define LOCKFILE "/etc/ptmp"
...
int pfd;
char filename[PATH_MAX+1];
...
if ((pfd = open(filename, O_WRONLY | O_CREAT | O_TRUNC,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    perror("Cannot open output file\n"); exit(1);
}
...
```

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). Because 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
```

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 location.

For example, a privileged application that must create a file with a predictable name in a user-writable directory, such as the user's home directory, could be compromised if the user creates a symbolic link with that name that refers to a nonexistent file in a system directory. If the user can influence the contents of a file, the user could compromise the system by creating a new system configuration or spool file that would then be interpreted by the system. The test for a symbolic link which refers to a nonexisting file must be atomic with the creation of a new file.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *close*( ), *creat*( ), *dup*( ), *fcntl*( ), *lseek*( ), *read*( ), *umask*( ), *unlockpt*( ), *write*( ), 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 4

The **<sys/types.h>** and **<sys/stat.h>** headers are now marked as optional (OH); these headers do not need to be included on XSI-conformant systems.

O_NDELAY is removed from the list of *oflag* values (this flag was marked WITHDRAWN in Issue 3).

The [ENXIO] error (for the condition where the file is a character or block special file and the associated device does not exist) and the [EINVAL] error are marked as extensions.

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument *path* is changed from **char*** to **const char***.
- Various wording changes are made to the DESCRIPTION to improve clarity and to align the text with the ISO POSIX-1 standard.

The following changes are incorporated for alignment with the FIPS requirements:

- In the DESCRIPTION, the description of O_CREAT is amended and the relevant part marked as an extension.
- In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path name component is larger that {NAME_MAX} is now defined as mandatory and marked as an extension.

Issue 4, Version 2

The following changes are incorporated for X/OPEN UNIX conformance:

- The DESCRIPTION is updated to define the use of open flags with STREAMS files, and to identify special considerations when opening the master side of a pseudo-terminal.
- In the ERRORS section, the [EIO], [ELOOP], and [ENOSR] mandatory error conditions are added, and the [EAGAIN], [ENAMETOOLONG], and [ENOMEM] optional error conditions are added.

Issue 5

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

Large File Summit extensions are added.

Issue 6

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

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, 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.
- 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 [ENXIO] mandatory error condition is added.
- The [EINVAL], [ENAMETOOLONG], and [ETXTBSY] optional error conditions are added.

The DESCRIPTION and ERRORS sections are updated so that items related to the optional XSI STREAMS Option Group are marked.

The following changes were made to align with the IEEE P1003.1a draft standard:

- An explanation is added of the effect of the O_CREAT and O_EXCL flags when the path refers to a symbolic link.
- The [ELOOP] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements. |

The DESCRIPTION of O_EXCL is updated in response to IEEE PASC Interpretation 1003.1c #48. |

NAME

opendir — open a directory

SYNOPSIS

```
#include <dirent.h>

DIR *opendir(const char *dirname);
```

DESCRIPTION

The *opendir*( ) function shall open a directory stream corresponding to the directory named by the *dirname* argument. The directory stream is positioned at the first entry. If the type **DIR** is implemented using a file descriptor, applications shall only be able to open up to a total of {OPEN_MAX} files and directories.

RETURN VALUE

Upon successful completion, *opendir*( ) shall return a pointer to an object of type **DIR**. Otherwise, a null pointer shall be returned and *errno* set to indicate the error.

ERRORS

The *opendir*( ) function shall fail if:

[EACCES] Search permission is denied for the component of the path prefix of *dirname* or read permission is denied for *dirname*.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *dirname* argument.

[ENAMETOOLONG] The length of the *dirname* argument exceeds {PATH_MAX} or a path name component is longer than {NAME_MAX}.

[ENOENT] A component of *dirname* does not name an existing directory or *dirname* is an empty string.

[ENOTDIR] A component of *dirname* is not a directory.

The *opendir*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *dirname* argument.

[EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

[ENAMETOOLONG] As a result of encountering a symbolic link in resolution of the *dirname* argument, the length of the substituted path name string exceeded {PATH_MAX}.

[ENFILE] Too many files are currently open in the system.

EXAMPLES

Open a Directory Stream

The following program fragment demonstrates how the *opendir*( ) function is used.

```
#include <sys/types.h>
#include <dirent.h>
#include <libgen.h>
...
    DIR *dir;
    struct dirent *dp;
...
    if ((dir = opendir (".")) == NULL) {
        perror ("Cannot open .");
        exit (1);
    }

    while ((dp = readdir (dir)) != NULL) {
...
```

APPLICATION USAGE

The *opendir*( ) function should be used in conjunction with *readdir*( ), *closedir*( ), and *rewinddir*( ) to examine the contents of the directory (see the EXAMPLES section in *readdir*( )). This method is recommended for portability.

RATIONALE

Based on historical implementations, the rules about file descriptors apply to directory streams as well. However, this volume of IEEE Std. 1003.1-200x does not mandate that the directory stream be implemented using file descriptors. The description of *closedir*( ) clarifies that if a file descriptor is used for the directory stream, it is mandatory that *closedir*( ) deallocate the file descriptor. When a file descriptor is used to implement the directory stream, it behaves as if the FD_CLOEXEC had been set for the file descriptor.

The directory entries for dot and dot-dot are optional. This volume of IEEE Std. 1003.1-200x does not provide a way to test *a priori* for their existence because an application that is portable must be written to look for (and usually ignore) those entries. Writing code that presumes that they are the first two entries does not always work, as many implementations permit them to be other than the first two entries, with a ‘‘normal’’ entry preceding them. There is negligible value in providing a way to determine what the implementation does because the code to deal with dot and dot-dot must be written in any case and because such a flag would add to the list of those flags (which has proven in itself to be objectionable) and might be abused.

Since the structure and buffer allocation, if any, for directory operations are defined by the implementation, this volume of IEEE Std. 1003.1-200x imposes no portability requirements for erroneous program constructs, erroneous data, or the use of indeterminate values such as the use or referencing of a *dirp* value or a **dirent** structure value after a directory stream has been closed or after a *fork*( ) or one of the *exec* function calls.

FUTURE DIRECTIONS

None.

SEE ALSO

closedir( ), *lstat*( ), *readdir*( ), *rewinddir*( ), *symlink*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<dirent.h>**, **<limits.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 2.

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

In the DESCRIPTION, the following sentence is moved to the Base Definitions volume of IEEE Std. 1003.1-200x: ''The type **DIR**, which is defined in **<dirent.h>**, represents a *directory stream*, which is an ordered sequence of all directory entries in a particular directory.''

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument *dirname* is changed from **char*** to **const char***.
- The generation of an [ENOENT] error when *dirname* points to an empty string is made mandatory.

The following change is incorporated for alignment with the FIPS requirements:

- In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path name component is larger that {NAME_MAX} is now defined as mandatory and marked as an extension.

Issue 4, Version 2

The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- It states that [ELOOP] is returned if too many symbolic links are encountered during path name resolution.
- A second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of path name resolution of a symbolic link.

Issue 6

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

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 [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.

27783 **NAME**

27784 openlog — open a connection to the logging facility

27785 **SYNOPSIS**

27786 XSI
```
#include <syslog.h>

void openlog(const char *ident, int logopt, int facility);
```
27787

27788

27789 **DESCRIPTION**

27790 Refer to *closelog*( ).

27791 **NAME**

27792 optarg, opterr, optind, optopt — options parsing variables

27793 **SYNOPSIS**

27794 `#include <unistd.h>`

27795 `extern char *optarg;`
27796 `extern int opterr, optind, optopt;`

27797 **DESCRIPTION**

27798 Refer to *getopt*( ).

27799 **NAME**

27800 pathconf — get configurable path name variables

27801 **SYNOPSIS**

27802 `#include <unistd.h>`

27803 `long pathconf(const char *`*`path`*`, int `*`name`*`);` |

27804 **DESCRIPTION** |

27805 Refer to *fpathconf*( ).

NAME

pause — suspend the thread until a signal is received

SYNOPSIS

```
#include <unistd.h>

int pause(void);
```

DESCRIPTION

The *pause*( ) function shall suspend the calling thread until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

If the action is to terminate the process, *pause*( ) shall not return.

If the action is to execute a signal-catching function, *pause*( ) shall return after the signal-catching function returns.

RETURN VALUE

Since *pause*( ) suspends thread execution indefinitely unless interrupted by a signal, there is no successful completion return value. A value of –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *pause*( ) function shall fail if:

[EINTR] A signal is caught by the calling process and control is returned from the signal-catching function.

EXAMPLES

None.

APPLICATION USAGE

Many common uses of *pause*( ) have timing windows. The scenario involves checking a condition related to a signal and, if the signal has not occurred, calling *pause*( ). When the signal occurs between the check and the call to *pause*( ), the process often blocks indefinitely. The *sigprocmask*( ) and *sigsuspend*( ) functions can be used to avoid this type of problem.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sigsuspend( ), 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 4

The **<unistd.h>** header is added to the SYNOPSIS section.

In the RETURN VALUE section, the text is expanded to indicate that process execution is suspended indefinitely ‘‘unless interrupted by a signal’’.

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The argument list is explicitly defined as **void**.

Issue 5

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

Issue 6

The APPLICATION USAGE section is added.

27850 **NAME**

27851 pclose — close a pipe stream to or from a process

27852 **SYNOPSIS**

27853 `#include <stdio.h>`

27854 `int pclose(FILE *stream);`

27855 **DESCRIPTION**

27856 The *pclose*( ) function shall close a stream that was opened by *popen*( ), wait for the command to
27857 terminate, and return the termination status of the process that was running the command
27858 language interpreter. However, if a call caused the termination status to be unavailable to
27859 *pclose*( ), then *pclose*( ) shall return –1 with *errno* set to [ECHILD] to report this situation. This can
27860 happen if the application calls one of the following functions:

27861 • *wait*( )

27862 • *waitpid*( ) with a *pid* argument less than or equal to 0 or equal to the process ID of the
27863 command line interpreter

27864 • Any other function not defined in this volume of IEEE Std. 1003.1-200x that could do one of
27865 the above

27866 In any case, *pclose*( ) shall not return before the child process created by *popen*( ) has terminated.

27867 If the command language interpreter cannot be executed, the child termination status returned
27868 by *pclose*( ) is as if the command language interpreter terminated using *exit*(127) or *_exit*(127).

27869 The *pclose*( ) function shall not affect the termination status of any child of the calling process
27870 other than the one created by *popen*( ) for the associated stream.

27871 If the argument *stream* to *pclose*( ) is not a pointer to a stream created by *popen*( ), the result of
27872 *pclose*( ) is undefined.

27873 **RETURN VALUE**

27874 Upon successful return, *pclose*( ) shall return the termination status of the command language
27875 interpreter. Otherwise, *pclose*( ) shall return –1 and set *errno* to indicate the error.

27876 **ERRORS**

27877 The *pclose*( ) function shall fail if:

27878 [ECHILD] The status of the child process could not be obtained, as described above. |

27879 **EXAMPLES**

27880 None.

27881 **APPLICATION USAGE**

27882 None.

27883 **RATIONALE**

27884 There is a requirement that *pclose*( ) not return before the child process terminates. This is
27885 intended to disallow implementations that return [EINTR] if a signal is received while waiting.
27886 If *pclose*( ) returned before the child terminated, there would be no way for the application to
27887 discover which child used to be associated with the stream, and it could not do the cleanup
27888 itself.

27889 If the stream pointed to by *stream* was not created by *popen*( ), historical implementations of
27890 *pclose*( ) return –1 without setting *errno*. To avoid requiring *pclose*( ) to set *errno* in this case,
27891 IEEE Std. 1003.1-200x makes the behavior unspecified. An application should not use *pclose*( ) to
27892 close any stream that was not created by *popen*( ).

Some historical implementations of *pclose*( ) either block or ignore the signals SIGINT, SIGQUIT, and SIGHUP while waiting for the child process to terminate. Since this behavior is not described for the *pclose*( ) function in IEEE Std. 1003.1-200x, such implementations are not conforming. Also, some historical implementations return [EINTR] if a signal is received, even though the child process has not terminated. Such implementations are also considered non-conforming.

Consider, for example, an application that uses:

```
popen("command", "r")
```

to start *command*, which is part of the same application. The parent writes a prompt to its standard output (presumably the terminal) and then reads from the stream. The child reads the response from the user, does some transformation on the response (path name expansion, perhaps) and writes the result to its standard output. The parent process reads the result from the pipe, does something with it, and prints another prompt. The cycle repeats. Assuming that both processes do appropriate buffer flushing, this would be expected to work.

To conform to IEEE Std. 1003.1-200x, *pclose*( ) must use *waitpid*( ), or some similar function, instead of *wait*( ).

The code sample below illustrates how the *pclose*( ) function might be implemented on a system conforming to IEEE Std. 1003.1-200x.

```
int pclose(FILE *stream)
{
    int stat;
    pid_t pid;

    pid = <pid for process created for stream by popen()>
    (void) fclose(stream);
    while (waitpid(pid, &stat, 0) == -1) {
        if (errno != EINTR){
            stat = -1;
            break;
        }
    }
    return(stat);
}
```

FUTURE DIRECTIONS

None.

SEE ALSO

fork( ), *popen*( ), *waitpid*( ), 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 4

The following changes are incorporated for alignment with the ISO POSIX-2 standard:

- The function is no longer marked as an extension.
- The simple DESCRIPTION given in Issue 3 is replaced with a more complete description in this issue. In particular, interactions between this function and *wait*( ) and *waitpid*( ) are defined.

NAME

perror — write error messages to standard error

SYNOPSIS

```
#include <stdio.h>

void perror(const char *s);
```

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 *perror*( ) function maps the error number accessed through the symbol *errno* to a language-dependent error message, which shall be written to the standard error stream as follows:

- First (if *s* is not a null pointer and the character pointed to by *s* is not the null byte), the string pointed to by *s* followed by a colon and a <space> character.
- Then an error message string followed by a <newline> character.

The contents of the error message strings are the same as those returned by *strerror*( ) with argument *errno*.

CX The *perror*( ) function shall mark the file associated with the standard error stream as having been written (*st_ctime*, *st_mtime* marked for update) at some time between its successful completion and *exit*( ), *abort*( ), or the completion of *fflush*( ) or *fclose*( ) on *stderr*.

The *perror*( ) function shall not change the orientation of the standard error stream.

RETURN VALUE

The *perror*( ) function shall return no value.

ERRORS

No errors are defined.

EXAMPLES

Printing an Error Message for a Function

The following example replaces *bufptr* with a buffer that is the necessary size. If an error occurs, the *perror*( ) function prints a message and the program exits.

```
#include <stdio.h>
#include <stdlib.h>
...
char *bufptr;
size_t szbuf;
...
if ((bufptr = malloc(szbuf)) == NULL) {
    perror("malloc"); exit(2);
}
...
```

APPLICATION USAGE

None.

27977 **RATIONALE**

27978 None.

27979 **FUTURE DIRECTIONS**

27980 None.

27981 **SEE ALSO**

27982 *strerror*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**

27983 **CHANGE HISTORY**

27984 First released in Issue 1. Derived from Issue 1 of the SVID.

27985 **Issue 4**

27986 The language for error message strings was given as implementation-defined in Issue 3. In this
27987 issue, they are defined as language-dependent.

27988 The following change is incorporated for alignment with the ISO POSIX-1 standard:

27989 • A paragraph is added to the DESCRIPTION defining the effects of this function on the
27990 *st_ctime* and *st_mtime* fields of the standard error stream.

27991 The following change is incorporated for alignment with the ISO C standard:

27992 • The type of argument *s* is changed from **char*** to **const char***.

27993 **Issue 5**

27994 A paragraph is added to the DESCRIPTION indicating that *perror*( ) does not change the
27995 orientation of the standard error stream.

27996 **Issue 6**

27997 Extensions beyond the ISO C standard are now marked.

NAME

pipe — create an interprocess channel

SYNOPSIS

```
#include <unistd.h>

int pipe(int fildes[2]);
```

DESCRIPTION

The *pipe*( ) function shall create a pipe and place two file descriptors, one each into the arguments *fildes*[0] and *fildes*[1], that refer to the open file descriptions for the read and write ends of the pipe. Their integer values shall be the two lowest available at the time of the *pipe*( ) call. The O_NONBLOCK and FD_CLOEXEC flags shall be clear on both file descriptors. (The *fcntl*( ) function can be used to set both these flags.)

Data can be written to the file descriptor *fildes*[1] and read from the file descriptor *fildes*[0]. A read on the file descriptor *fildes*[0] shall access data written to the file descriptor *fildes*[1] on a first-in-first-out basis. It is unspecified whether *fildes*[0] is also open for writing and whether *fildes*[1] is also open for reading.

A process has the pipe open for reading (correspondingly writing) if it has a file descriptor open that refers to the read end, *fildes*[0] (write end, *fildes*[1]).

Upon successful completion, *pipe*( ) shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the pipe.

RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *pipe*( ) function shall fail if:

[EMFILE] More than {OPEN_MAX} minus two file descriptors are already in use by this process.

[ENFILE] The number of simultaneously open files in the system would exceed a system-imposed limit.

EXAMPLES

None.

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 4

The **<unistd.h>** header is added to the SYNOPSIS section.

Issue 4, Version 2

The DESCRIPTION is updated for X/OPEN UNIX conformance to indicate that certain dispositions of *fildes*[0] and *fildes*[1] are unspecified.

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.

28050 **NAME**

28051 poll — input/output multiplexing

28052 **SYNOPSIS**

28053 XSI
```
#include <poll.h>

int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```
28054

28055

28056 **DESCRIPTION**

28057 The *poll*( ) function provides applications with a mechanism for multiplexing input/output over
28058 a set of file descriptors. For each member of the array pointed to by *fds*, *poll*( ) examines the given
28059 file descriptor for the event(s) specified in *events*. The number of **pollfd** structures in the *fds*
28060 array is specified by *nfds*. The *poll*( ) function identifies those file descriptors on which an
28061 application can read or write data, or on which certain events have occurred.

28062 The *fds* argument specifies the file descriptors to be examined and the events of interest for each
28063 file descriptor. It is a pointer to an array with one member for each open file descriptor of
28064 interest. The array's members are **pollfd** structures within which *fd* specifies an open file
28065 descriptor and *events* and *revents* are bitmasks constructed by OR'ing a combination of the
28066 following event flags:

28067 XSR POLLIN Data other than high-priority data may be read without blocking. For
28068 STREAMS, this flag is set in *revents* even if the message is of zero length.

28069 XSR POLLRDNORM Normal data (priority band equals 0) may be read without blocking. For
28070 STREAMS, this flag is set in *revents* even if the message is of zero length.

28071 XSR POLLRDBAND Data from a non-zero priority band may be read without blocking. For
28072 STREAMS, this flag is set in *revents* even if the message is of zero length.

28073 XSR POLLPRI High-priority data may be received without blocking. For STREAMS, this flag
28074 is set in *revents* even if the message is of zero length.

28075 POLLOUT Normal data (priority band equals 0) may be written without blocking.

28076 POLLWRNORM Same as POLLOUT.

28077 POLLWRBAND Priority data (priority band >0) may be written. This event only examines
28078 bands that have been written to at least once.

28079 POLLERR An error has occurred on the device or stream. This flag is only valid in the
28080 *revents* bitmask; it is ignored in the *events* member.

28081 POLLHUP The device has been disconnected. This event and POLLOUT are mutually-
28082 exclusive; a stream can never be writable if a hangup has occurred. However,
28083 this event and POLLIN, POLLRDNORM, POLLRDBAND, or POLLPRI are not
28084 mutually-exclusive. This flag is only valid in the *revents* bitmask; it is ignored
28085 in the *events* member.

28086 POLLNVAL The specified *fd* value is invalid. This flag is only valid in the *revents* member;
28087 it is ignored in the *events* member.

28088 If the value of *fd* is less than 0, *events* is ignored, and *revents* is set to 0 in that entry on return from
28089 *poll*( ).

28090 In each **pollfd** structure, *poll*( ) clears the *revents* member, except that where the application
28091 requested a report on a condition by setting one of the bits of *events* listed above, *poll*( ) sets the
28092 corresponding bit in *revents* if the requested condition is true. In addition, *poll*( ) sets the
28093 POLLHUP, POLLERR, and POLLNVAL flag in *revents* if the condition is true, even if the

application did not set the corresponding bit in *events*.

If none of the defined events have occurred on any selected file descriptor, *poll*( ) waits at least *timeout* milliseconds for an event to occur on any of the selected file descriptors. If the value of *timeout* is 0, *poll*( ) shall return immediately. If the value of *timeout* is –1, *poll*( ) shall block until a requested event occurs or until the call is interrupted.

Implementations may place limitations on the granularity of timeout intervals. If the requested timeout interval requires a finer granularity than the implementation supports, the actual timeout interval shall be rounded up to the next supported value.

The *poll*( ) function is not affected by the O_NONBLOCK flag.

XSR The *poll*( ) function supports regular files, terminal and pseudo-terminal devices, STREAMS-based files, FIFOs, pipes, and sockets. The behavior of *poll*( ) on elements of *fds* that refer to other types of file is unspecified.

Regular files always poll TRUE for reading and writing.

The *poll*( ) function supports sockets.

A file descriptor for a socket that is listening for connections shall indicate that it is ready for reading, once connections are available. A file descriptor for a socket that is connecting asynchronously shall indicate that it is ready for writing, once a connection has been established.

RETURN VALUE

Upon successful completion, *poll*( ) shall return a non-negative value. A positive value indicates the total number of file descriptors that have been selected (that is, file descriptors for which the *revents* member is non-zero). A value of 0 indicates that the call timed out and no file descriptors have been selected. Upon failure, *poll*( ) shall return –1 and set *errno* to indicate the error.

ERRORS

The *poll*( ) function shall fail if:

[EAGAIN] The allocation of internal data structures failed but a subsequent request may succeed.

[EINTR] A signal was caught during *poll*( ).

XSR [EINVAL] The *nfds* argument is greater than {OPEN_MAX}, or one of the *fd* members refers to a STREAM or multiplexer that is linked (directly or indirectly) 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.

```
#include <stropts.h>
#include <poll.h>
...
struct pollfd fds[2];
int timeout_msecs = 500;
int ret;
    int i;

/* Open STREAMS device. */
fds[0].fd = open("/dev/dev0", ...);
fds[1].fd = open("/dev/dev1", ...);
    fds[0].events = POLLOUT | POLLWRBAND;
    fds[1].events = POLLOUT | POLLWRBAND;

ret = poll(fds, 2, timeout_msecs);

if (ret > 0) {
    /* An event on one of the fds has occurred. */
    for (i=0; i<2; i++) {
        if (fds[i].revents & POLLWRBAND) {
        /* Priority data may be written on device number i. */
...
        }
        if (fds[i].revents & POLLOUT) {
        /* Data may be written on device number i. */
...
        }
        if (fds[i].revents & POLLHUP) {
        /* A hangup has occurred on device number i. */
...
        }
    }
}
```

APPLICATION USAGE

None.

28179 **RATIONALE**

28180 None.

28181 **FUTURE DIRECTIONS**

28182 None.

28183 **SEE ALSO**

28184 *getmsg*( ), *putmsg*( ), *read*( ), *select*( ), *write*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
28185 **<poll.h>**, **<stropts.h>**, Section 2.6 (on page 539)

28186 **CHANGE HISTORY**

28187 First released in Issue 4, Version 2.

28188 **Issue 5**

28189 Moved from X/OPEN UNIX extension to BASE.

28190 The description of POLLWRBAND is updated.

28191 **Issue 6**

28192 Text referring to sockets is added to the DESCRIPTION.

28193 Text relating to the XSI STREAMS Option Group is marked.

NAME

popen — initiate pipe streams to or from a process

SYNOPSIS

```
#include <stdio.h>

FILE *popen(const char *command, const char *mode);
```

DESCRIPTION

The *popen*( ) function shall execute the command specified by the string *command*. It creates a pipe between the calling program and the executed command, and returns a pointer to a stream that can be used to either read from or write to the pipe.

The environment of the executed command shall be as if a child process were created within the *popen*( ) call using the *fork*( ) function, and the child used *execl*( ) to invoke a command line interpreter.

If the implementation supports the Shell and Utilities volume of IEEE Std. 1003.1-200x, the environment of the executed command is as if a child process were created within the *popen*( ) call using *fork*( ), and the child invoked the *sh* utility using the call:

```
execl(shell path, "sh", "-c", command, (char *)0);
```

where *shell path* is an unspecified path name for the *sh* utility.

The *popen*( ) function ensures that any streams from previous *popen*( ) calls that remain open in the parent process are closed in the new child process.

The *mode* argument to *popen*( ) is a string that specifies I/O mode:

1. If *mode* is *r*, when the child process is started, its file descriptor STDOUT_FILENO shall be the writable end of the pipe, and the file descriptor *fileno*(*stream*) in the calling process, where *stream* is the stream pointer returned by *popen*( ), shall be the readable end of the pipe.
2. If *mode* is *w*, when the child process is started its file descriptor STDIN_FILENO shall be the readable end of the pipe, and the file descriptor *fileno*(*stream*) in the calling process, where *stream* is the stream pointer returned by *popen*( ), shall be the writable end of the pipe.
3. If *mode* is any other value, the result is undefined.

After *popen*( ), both the parent and the child process shall be capable of executing independently before either terminates.

Pipe streams are byte-oriented.

RETURN VALUE

Upon successful completion, *popen*( ) shall return a pointer to an open stream that can be used to read or write to the pipe. Otherwise, it shall return a null pointer and may set *errno* to indicate the error.

ERRORS

The *popen*( ) function may fail if:

[EMFILE] {FOPEN_MAX} or {STREAM_MAX} streams are currently open in the calling process.

[EINVAL] The *mode* argument is invalid.

The *popen*( ) function may also set *errno* values as described by *fork*( ) or *pipe*( ).

28236 **EXAMPLES**

28237 None.

28238 **APPLICATION USAGE**

28239 Because open files are shared, a mode *r* command can be used as an input filter and a mode *w*
28240 command as an output filter.

28241 Buffered reading before opening an input filter may leave the standard input of that filter
28242 mispositioned. Similar problems with an output filter may be prevented by careful buffer
28243 flushing; for example, with *fflush*( ).

28244 A stream opened by *popen*( ) should be closed by *pclose*( ).

28245 The behavior of *popen*( ) is specified for values of *mode* of *r* and *w*. Other modes such as *rb* and
28246 *wb* might be supported by specific implementations, but these would not be portable features.
28247 Note that historical implementations of *popen*( ) only check to see if the first character of *mode* is
28248 *r*. Thus, a *mode* of *robert the robot* would be treated as *mode r*, and a *mode* of *anything else* would be
28249 treated as *mode w*.

28250 If the application calls *waitpid*( ) or *waitid*( ) with a *pid* argument greater than 0, and it still has a
28251 stream that was called with *popen*( ) open, it must ensure that *pid* does not refer to the process
28252 started by *popen*( ).

28253 To determine whether or not the environment specified in the Shell and Utilities volume of |
28254 IEEE Std. 1003.1-200x is present, use the function call: |

28255 `sysconf(_SC_2_VERSION)`

28256 (See *sysconf*( )).

28257 **RATIONALE**

28258 The *popen*( ) function should not be used by programs that have set user (or group) ID privileges.
28259 The *fork*( ) and *exec* family of functions (except *execlp*( ) and *execvp*( )), should be used instead.
28260 This prevents any unforeseen manipulation of the environment of the user that could cause
28261 execution of commands not anticipated by the calling program.

28262 If the original and *popen*( )ed processes both intend to read or write or read and write a common |
28263 file, and either will be using FILE-type C functions (*fread*( ), *fwrite*( ), and so on), the rules for |
28264 sharing file handles must be observed (see Section 2.5.1 (on page 535)). |

28265 Because open files are shared, a mode *r* argument can be used as an input filter and a mode *w* |
28266 argument as an output filter. |

28267 The behavior of *popen*( ) is specified for modes of *r* and *w*. Other modes such as *rb* and *wb* might |
28268 be supported by specific implementations, but these would not be portable features. Note that
28269 historical implementations of *popen*( ) only check to see if the first character of mode is `'r'`.
28270 Thus, a mode of *robert the robot* would be treated as mode *r*, and a mode of *anything else* would be |
28271 treated as mode *w*. |

28272 If the application calls *waitpid*( ) with a *pid* argument greater than zero, and it still has a
28273 *popen*( )ed stream open, it must ensure that *pid* does not refer to the process started by *popen*( ).

28274 **FUTURE DIRECTIONS**

28275 None.

28276 **SEE ALSO**

28277 *pclose*( ), *pipe*( ), *sysconf*( ), *system*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
28278 **<stdio.h>**, the Shell and Utilities volume of IEEE Std. 1003.1-200x |

CHANGE HISTORY

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

Issue 4

The APPLICATION USAGE section is extended. Only notes about buffer flushing are retained from Issue 3.

The following changes are incorporated for alignment with the ISO POSIX-2 standard:

- The function is no longer marked as an extension.
- The type of arguments *command* and *mode* are changed from **char*** to **const char***.
- The DESCRIPTION is completely rewritten for alignment with the ISO POSIX-2 standard, although it describes essentially the same functionality as Issue 3.
- The *sh* utility defined in the Shell and Utilities volume of IEEE Std. 1003.1-200x is no longer required in all circumstances.
- The ERRORS section is added.

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.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The DESCRIPTION is adjusted to reflect the behavior on systems that do not support the Shell option.

28301 **NAME**

28302 posix_fadvise — file advisory information

28303 **SYNOPSIS**

28304 ADV
```
#include <fcntl.h>
```

28305
```
int posix_fadvise(int fd, off_t offset, size_t len, int advice);
```

28306

28307 **DESCRIPTION**

28308 The *posix_fadvise*( ) function provides advice to the implementation on the expected behavior of
28309 the application with respect to the data in the file associated with the open file descriptor, *fd*,
28310 starting at *offset* and continuing for *len* bytes. The specified range need not currently exist in the
28311 file. If *len* is zero, all data following *offset* is specified. The implementation may use this
28312 information to optimize handling of the specified data. The *posix_fadvise*( ) function has no effect
28313 on the semantics of other operations on the specified data, although it may affect the |
28314 performance of other operations. |

28315 The advice to be applied to the data is specified by the *advice* parameter and may be one of the
28316 following values:

28317 POSIX_FADV_NORMAL
28318 Specifies that the application has no advice to give on its behavior with respect to the
28319 specified data. It is the default characteristic if no advice is given for an open file.

28320 POSIX_FADV_SEQUENTIAL
28321 Specifies that the application expects to access the specified data sequentially from lower
28322 offsets to higher offsets.

28323 POSIX_FADV_RANDOM
28324 Specifies that the application expects to access the specified data in a random order.

28325 POSIX_FADV_WILLNEED
28326 Specifies that the application expects to access the specified data in the near future.

28327 POSIX_FADV_DONTNEED
28328 Specifies that the application expects that it will not access the specified data in the near
28329 future.

28330 POSIX_FADV_NOREUSE
28331 Specifies that the application expects to access the specified data once and then not reuse it
28332 thereafter.

28333 These values shall be defined in **<fcntl.h>**. |

28334 **RETURN VALUE**

28335 Upon successful completion, *posix_fadvise*( ) shall return zero; otherwise, an error number shall
28336 be returned to indicate the error.

28337 **ERRORS**

28338 The *posix_fadvise*( ) function shall fail if:

28339 [EBADF] The *fd* argument is not a valid file descriptor.

28340 [EINVAL] The value of *advice* is invalid. |

28341 [ESPIPE] The *fd* argument is associated with a pipe or FIFO.

28342 **EXAMPLES**

28343 None.

28344 **APPLICATION USAGE**

28345 The *posix_fadvise*( ) function is part of the Advisory Information option and need not be provided |
28346 on all implementations.

28347 **RATIONALE**

28348 None.

28349 **FUTURE DIRECTIONS**

28350 None.

28351 **SEE ALSO**

28352 *posix_madvise*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<fcntl.h>** |

28353 **CHANGE HISTORY**

28354 First released in Issue 6. Derived from IEEE Std. 1003.1d-1999.

28355 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

28356 **NAME**

28357 posix_fallocate — file space control

28358 **SYNOPSIS**

28359 ADV
```
#include <fcntl.h>
```

28360
```
int posix_fallocate(int fd, off_t offset, size_t len);
```

28361

28362 **DESCRIPTION**

28363 The *posix_fallocate*( ) function ensures that any required storage for regular file data starting at
28364 *offset* and continuing for *len* bytes is allocated on the file system storage media. If *posix_fallocate*( )
28365 returns successfully, subsequent writes to the specified file data shall not fail due to the lack of
28366 free space on the file system storage media.

28367 If the *offset*+*len* is beyond the current file size, then *posix_fallocate*( ) shall adjust the file size to
28368 *offset*+*len*. Otherwise, the file size shall not be changed.

28369 It is implementation-defined whether a previous *posix_fadvise*( ) call influences allocation |
28370 strategy.

28371 Space allocated via *posix_fallocate*( ) shall be freed by a successful call to *creat*( ) or *open*( ) that
28372 truncates the size of the file. Space allocated via *posix_fallocate*( ) may be freed by a successful call
28373 to *ftruncate*( ) that reduces the file size to a size smaller than *offset*+*len*.

28374 **RETURN VALUE**

28375 Upon successful completion, *posix_fallocate*( ) shall return zero; otherwise, an error number shall
28376 be returned to indicate the error.

28377 **ERRORS**

28378 The *posix_fallocate*( ) function shall fail if:

28379 [EBADF] The *fd* argument is not a valid file descriptor.

28380 [EBADF] The *fd* argument references a file that was opened without write permission.

28381 [EFBIG] The value of *offset*+*len* is greater than the maximum file size.

28382 [EINTR] A signal was caught during execution.

28383 [EINVAL] The *len* argument was zero or the *offset* argument was less than zero.

28384 [EIO] An I/O error occurred while reading from or writing to a file system.

28385 [ENODEV] The *fd* argument does not refer to a regular file.

28386 [ENOSPC] There is insufficient free space remaining on the file system storage media.

28387 [ESPIPE] The *fd* argument is associated with a pipe or FIFO.

28388 **EXAMPLES**

28389 None.

28390 **APPLICATION USAGE**

28391 The *posix_fallocate*( ) function is part of the Advisory Information option and need not be |
28392 provided on all implementations.

28393 **RATIONALE**

28394 None.

28395 **FUTURE DIRECTIONS**

28396 None.

28397 **SEE ALSO**

28398 *creat*( ), *ftruncate*( ), *open*( ), *unlink*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
28399 **<fcntl.h>**

28400 **CHANGE HISTORY**

28401 First released in Issue 6. Derived from IEEE Std. 1003.1d-1999.

28402 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

28403 **NAME**

28404 posix_madvise — memory advisory information and alignment control

28405 **SYNOPSIS**

28406 ADV
```
#include <sys/mman.h>
```

28407
```
int posix_madvise(void *addr, size_t len, int advice);
```

28408

28409 **DESCRIPTION**

28410 MF|SHM The *posix_madvise*( ) function need only be supported if either the Memory Mapped Files or the
28411 Shared Memory Objects options are supported.

28412 The *posix_madvise*( ) function provides advice to the implementation on the expected behavior of
28413 the application with respect to the data in the memory starting at address *addr*, and continuing
28414 for *len* bytes. The implementation may use this information to optimize handling of the specified
28415 data. The *posix_madvise*( ) function has no effect on the semantics of access to memory in the
28416 specified range, although it may affect the performance of access.

28417 The implementation may require that *addr* be a multiple of the page size, which is the value
28418 returned by *sysconf*( ) when the name value _SC_PAGESIZE is used.

28419 The advice to be applied to the memory range is specified by the *advice* parameter and may be
28420 one of the following values:

28421 POSIX_MADV_NORMAL
28422 Specifies that the application has no advice to give on its behavior with respect to the
28423 specified range. It is the default characteristic if no advice is given for a range of memory.

28424 POSIX_MADV_SEQUENTIAL
28425 Specifies that the application expects to access the specified range sequentially from lower
28426 addresses to higher addresses.

28427 POSIX_MADV_RANDOM
28428 Specifies that the application expects to access the specified range in a random order.

28429 POSIX_MADV_WILLNEED
28430 Specifies that the application expects to access the specified range in the near future.

28431 POSIX_MADV_DONTNEED
28432 Specifies that the application expects that it will not access the specified range in the near
28433 future.

28434 These values shall be defined in **<sys/mman.h>**.

28435 **RETURN VALUE**

28436 Upon successful completion, *posix_madvise*( ) shall return zero; otherwise, an error number shall
28437 be returned to indicate the error.

28438 **ERRORS**

28439 The *posix_madvise*( ) function shall fail if:

28440 [EINVAL] The value of *advice* is invalid.

28441 [ENOMEM] Addresses in the range starting at *addr* and continuing for *len* bytes are partly
28442 or completely outside the range allowed for the address space of the calling
28443 process.

28444 The *posix_madvise*( ) function may fail if:

[EINVAL] The value of *addr* is not a multiple of the value returned by *sysconf*( ) when the name value _SC_PAGESIZE is used.

[EINVAL] The value of *len* is zero.

EXAMPLES

None.

APPLICATION USAGE

The *posix_madvise*( ) function is part of the Advisory Information option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mmap( ), *posix_fadvise*( ), *sysconf*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/mmap.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std. 1003.1d-1999.

IEEE PASC Interpretation 1003.1 #102 is included.

28463 **NAME**

28464 posix_mem_offset — find offset and length of a mapped typed memory block

28465 **SYNOPSIS**

28466 TYM
```
#include <sys/mman.h>

int posix_mem_offset(const void *restrict addr, size_t len,
    off_t *restrict off, size_t *restrict contig_len,
    int *restrict fildes);
```

28470

28471 **DESCRIPTION**

28472 The *posix_mem_offset*( ) function returns in the variable pointed to by *off* a value that identifies
28473 the offset (or location), within a memory object, of the memory block currently mapped at *addr*.
28474 The function shall return in the variable pointed to by *fildes*, the descriptor used (via *mmap*( )) to
28475 establish the mapping which contains *addr*. If that descriptor was closed since the mapping was
28476 established, the returned value of *fildes* shall be –1. The *len* argument specifies the length of the
28477 block of the memory object the user wishes the offset for; upon return, the value pointed to by
28478 *contig_len* shall equal either *len*, or the length of the largest contiguous block of the memory
28479 object that is currently mapped to the calling process starting at *addr*, whichever is smaller.

28480 If the memory object mapped at *addr* is a typed memory object, then if the *off* and *contig_len*
28481 values obtained by calling *posix_mem_offset*( ) are used in a call to *mmap*( ) with a file descriptor
28482 that refers to the same memory pool as *fildes* (either through the same port or through a different
28483 port), and that was opened with neither the POSIX_TYPED_MEM_ALLOCATE nor the
28484 POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, the typed memory area that is mapped shall
28485 be exactly the same area that was mapped at *addr* in the address space of the process that called
28486 *posix_mem_offset*( ).

28487 If the memory object specified by *fildes* is not a typed memory object, then the behavior of this
28488 function is implementation-defined.

28489 **RETURN VALUE**

28490 Upon successful completion, the *posix_mem_offset*( ) function shall return zero; otherwise, the
28491 corresponding error status value shall be returned.

28492 **ERRORS**

28493 The *posix_mem_offset*( ) function shall fail if:

28494 [EACCES] The process has not mapped a memory object supported by this function at
28495 the given address *addr*.

28496 This function shall not return an error code of [EINTR].

28497 **EXAMPLES**

28498 None.

28499 **APPLICATION USAGE**

28500 None.

28501 **RATIONALE**

28502 None.

28503 **FUTURE DIRECTIONS**

28504 None.

28505 **SEE ALSO**

28506 *mmap*( ), *posix_typed_mem_open*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
28507 **<sys/mman.h>**

28508 **CHANGE HISTORY**

28509 First released in Issue 6. Derived from IEEE Std. 1003.1j-2000.

NAME

posix_memalign — aligned memory allocation

SYNOPSIS

ADV
```
#include <stdlib.h>

int posix_memalign(void **memptr, size_t alignment, size_t size);
```

DESCRIPTION

The *posix_memalign*( ) function allocates *size* bytes aligned on a boundary specified by *alignment*, and returns a pointer to the allocated memory in *memptr*. The value of *alignment* shall be a multiple of *sizeof*(**void***), that is also a power of two. Upon successful completion, the value pointed to by *memptr* shall be a multiple of *alignment*.

CX The *free*( ) function shall deallocate memory that has previously been allocated by *posix_memalign*( ).

RETURN VALUE

Upon successful completion, *posix_memalign*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *posix_memalign*( ) function shall fail if:

[EINVAL] The value of the alignment parameter is not a power of two multiple of *sizeof*(**void***).

[ENOMEM] There is insufficient memory available with the requested alignment.

EXAMPLES

None.

APPLICATION USAGE

The *posix_memalign*( ) function is part of the Advisory Information option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

free( ), *malloc*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.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.

NAME

posix_spawn, posix_spawnp — spawn a process (**REALTIME**)

SYNOPSIS

SPN

```
#include <spawn.h>

int posix_spawn(pid_t *restrict pid, const char *restrict path,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *restrict attrp,
    char *restrict const argv[restrict],
    char *restrict const envp[restrict]);
int posix_spawnp(pid_t *restrict pid, const char *restrict file,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *restrict attrp, char *const argv[restrict],
    char * const envp[restrict]);
```

DESCRIPTION

The *posix_spawn*( ) and *posix_spawnp*( ) functions shall create a new process (child process) from the specified process image. The new process image is constructed from a regular executable file called the new process image file.

When a C program is executed as the result of this call, it shall be entered as a C language function call as follows:

```
int main(int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings.

The argument *argv* is an array of character pointers to null-terminated strings. The last member of this array shall be a null pointer and is not counted in *argc*. These strings constitute the argument list available to the new process image. The value in *argv*[0] should point to a file name that is associated with the process image being started by the *posix_spawn*( ) or *posix_spawnp*( ) function.

The argument *envp* is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The environment array is terminated by a null pointer.

The number of bytes available for the child process' combined argument and environment lists is {ARG_MAX}. The implementation shall specify in the system documentation (see the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 2, Conformance) whether any list overhead, such as length words, null terminators, pointers, or alignment bytes, is included in this total.

The *path* argument to *posix_spawn*( ) is a path name that identifies the new process image file to execute.

The *file* parameter to *posix_spawnp*( ) shall be used to construct a path name that identifies the new process image file. If the *file* parameter contains a slash character, the *file* parameter shall be used as the path name for the new process image file. Otherwise, the path prefix for this file shall be obtained by a search of the directories passed as the environment variable *PATH* (see the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 8, Environment Variables). If this environment variable is not defined, the results of the search are implementation-defined.

If *file_actions* is a null pointer, then file descriptors open in the calling process shall remain open in the child process, except for those whose close-on-*exec* flag FD_CLOEXEC is set (see *fcntl*( )). For those file descriptors that remain open, all attributes of the corresponding open file descriptions, including file locks (see *fcntl*( )), shall remain unchanged.

If *file_actions* is not NULL, then the file descriptors open in the child process shall be those open in the calling process as modified by the spawn file actions object pointed to by *file_actions* and the FD_CLOEXEC flag of each remaining open file descriptor after the spawn file actions have been processed. The effective order of processing the spawn file actions shall be:

1. The set of open file descriptors for the child process shall initially be the same set as is open for the calling process. All attributes of the corresponding open file descriptions, including file locks (see *fcntl*( )), shall remain unchanged.
2. The signal mask, signal default actions, and the effective user and group IDs for the child process shall be changed as specified in the attributes object referenced by *attrp*.
3. The file actions specified by the spawn file actions object shall be performed in the order in which they were added to the spawn file actions object.
4. Any file descriptor that has its FD_CLOEXEC flag set (see *fcntl*( )) shall be closed.

The **posix_spawnattr_t** spawn attributes object type is defined in **<spawn.h>**. It shall contain at least the attributes defined below.

If the POSIX_SPAWN_SETPGROUP flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, and the *spawn-pgroup* attribute of the same object is non-zero, then the child's process group shall be as specified in the *spawn-pgroup* attribute of the object referenced by *attrp*.

As a special case, if the POSIX_SPAWN_SETPGROUP flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, and the *spawn-pgroup* attribute of the same object is set to zero, then the child shall be in a new process group with a process group ID equal to its process ID.

If the POSIX_SPAWN_SETPGROUP flag is not set in the *spawn-flags* attribute of the object referenced by *attrp*, the new child process shall inherit the parent's process group.

PS If the POSIX_SPAWN_SETSCHEDPARAM flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, but POSIX_SPAWN_SETSCHEDULER is not set, the new process image shall initially have the scheduling policy of the calling process with the scheduling parameters specified in the *spawn-schedparam* attribute of the object referenced by *attrp*.

If the POSIX_SPAWN_SETSCHEDULER flag is set in *spawn-flags* attribute of the object referenced by *attrp* (regardless of the setting of the POSIX_SPAWN_SETSCHEDPARAM flag), the new process image shall initially have the scheduling policy specified in the *spawn-schedpolicy* attribute of the object referenced by *attrp* and the scheduling parameters specified in the *spawn-schedparam* attribute of the same object.

The POSIX_SPAWN_RESETIDS flag in the *spawn-flags* attribute of the object referenced by *attrp* governs the effective user ID of the child process. If this flag is not set, the child process inherits the parent process' effective user ID. If this flag is set, the child process' effective user ID is reset to the parent's real user ID. In either case, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the child process will become that file's owner ID before the new process image begins execution.

The POSIX_SPAWN_RESETIDS flag in the *spawn-flags* attribute of the object referenced by *attrp* also governs the effective group ID of the child process. If this flag is not set, the child process inherits the parent process' effective group ID. If this flag is set, the child process' effective group ID is reset to the parent's real group ID. In either case, if the set-group-ID mode bit of the new

process image file is set, the effective group ID of the child process will become that file's group ID before the new process image begins execution.

If the POSIX_SPAWN_SETSIGMASK flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, the child process shall initially have the signal mask specified in the *spawn-sigmask* attribute of the object referenced by *attrp*.

If the POSIX_SPAWN_SETSIGDEF flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, the signals specified in the *spawn-sigdefault* attribute of the same object shall be set to their default actions in the child process. Signals set to the default action in the parent process shall be set to the default action in the child process.

Signals set to be caught by the calling process shall be set to the default action in the child process.

Signals set to be ignored by the calling process image shall be set to be ignored by the child process, unless otherwise specified by the POSIX_SPAWN_SETSIGDEF flag being set in the *spawn-flags* attribute of the object referenced by *attrp* and the signals being indicated in the *spawn-sigdefault* attribute of the object referenced by *attrp*.

If the value of the *attrp* pointer is NULL, then the default values are used.

All process attributes, other than those influenced by the attributes set in the object referenced by *attrp* as specified above or by the file descriptor manipulations specified in *file_actions*, shall appear in the new process image as though *fork*( ) had been called to create a child process and then a member of the *exec* family of functions had been called by the child process to execute the new process image.

THR It is implementation-defined whether the fork handlers are run when *posix_spawn*( ) or *posix_spawnp*( ) is called.

RETURN VALUE

Upon successful completion, *posix_spawn*( ) and *posix_spawnp*( ) shall return the process ID of the child process to the parent process, in the variable pointed to by a non-NULL *pid* argument, and shall return zero as the function return value. Otherwise, no child process shall be created, the value stored into the variable pointed to by a non-NULL *pid* is unspecified, and an error number shall be returned as the function return value to indicate the error. If the *pid* argument is a null pointer, the process ID of the child is not returned to the caller.

ERRORS

The *posix_spawn*( ) and *posix_spawnp*( ) functions may fail if:

[EINVAL] The value specified by *file_actions* or *attrp* is invalid.

If this error occurs after the calling process successfully returns from the *posix_spawn*( ) or *posix_spawnp*( ) function, the child process may exit with exit status 127.

If *posix_spawn*( ) or *posix_spawnp*( ) fail for any of the reasons that would cause *fork*( ) or one of the *exec* family of functions to fail, an error value shall be returned as described by *fork*( ) and *exec*, respectively (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

If POSIX_SPAWN_SETPGROUP is set in the *spawn-flags* attribute of the object referenced by *attrp*, and *posix_spawn*( ) or *posix_spawnp*( ) fails while changing the child's process group, an error value shall be returned as described by *setpgid*( ) (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

PS If POSIX_SPAWN_SETSCHEDPARAM is set and POSIX_SPAWN_SETSCHEDULER is not set in the *spawn-flags* attribute of the object referenced by *attrp*, then if *posix_spawn*( ) or

posix_spawnp( ) fails for any of the reasons that would cause *sched_setparam*( ) to fail, an error value shall be returned as described by *sched_setparam*( ) (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

If POSIX_SPAWN_SETSCHEDULER is set in the *spawn-flags* attribute of the object referenced by *attrp*, and if *posix_spawn*( ) or *posix_spawnp*( ) fails for any of the reasons that would cause *sched_setscheduler*( ) to fail, an error value shall be returned as described by *sched_setscheduler*( ) (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127)>

If the *file_actions* argument is not NULL, and specifies any *close*, *dup2*, or *open* actions to be performed, and if *posix_spawn*( ) or *posix_spawnp*( ) fails for any of the reasons that would cause *close*( ), *dup2*( ), or *open*( ) to fail, an error value shall be returned as described by *close*( ), *dup2*( ), and *open*( ), respectively (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127). An open file action may, by itself, result in any of the errors described by *close*( ) or *dup2*( ), in addition to those described by *open*( ).

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE

The POSIX *fork*( ) function is difficult or impossible to implement without swapping or dynamic address translation for the following reasons:

- Swapping is generally too slow for a realtime environment.
- Dynamic address translation is not available everywhere POSIX might be useful.
- Processes are too useful to simply option out of POSIX whenever it must run without address translation or other MMU services,

POSIX needs process creation and file execution primitives that can be efficiently implemented without address translation or other MMU services.

This function shall be called *posix_spawn*( ). A closely related function, *posix_spawnp*( ), is included for completeness.

The *posix_spawn*( ) function is implementable as a library routine, but both *posix_spawn*( ) and *posix_spawnp*( ) are designed as kernel operations. Also, although they may be an efficient replacement for many *fork*( )/*exec* pairs, their goal is to provide useful process creation primitives for systems that have difficulty with *fork*( ), not to provide drop-in replacements for *fork*( )/*exec*.

This view of the role of *posix_spawn*( ) and *posix_spawnp*( ) influenced the design of their API. It does not attempt to provide the full functionality of *fork*( )/*exec* in which arbitrary user-specified operations of any sort are permitted between the creation of the child process and the execution of the new process image; any attempt to reach that level would need to provide a programming language as parameters. Instead, *posix_spawn*( ) and *posix_spawnp*( ) are process creation primitives like the *Start_Process* and *Start_Process_Search* Ada language bindings package *POSIX_Process_Primitives* and also like those in many operating systems that are not UNIX systems, but with some POSIX-specific additions.

To achieve its coverage goals, *posix_spawn*( ) and *posix_spawnp*( ) have control of six types of inheritance: file descriptors, process group ID, user and group ID, signal mask, scheduling, and whether each signal ignored in the parent will remain ignored in the child, or be reset to its default action in the child.

Control of file descriptors is required to allow an independently written child process image to access data streams opened by and even generated or read by the parent process without being specifically coded to know which parent files and file descriptors are to be used. Control of the process group ID is required to control how the child process' job control relates to that of the parent.

Control of the signal mask and signal defaulting is sufficient to support the implementation of *system*(). Although support for *system*() is not explicitly one of the goals for *posix_spawn*() and *posix_spawnp*(), it is covered under the ''at least 50%'' coverage goal.

The intention is that the normal file descriptor inheritance across *fork*(), the subsequent effect of the specified spawn file actions, and the normal file descriptor inheritance across one of the *exec* family of functions should fully specify open file inheritance. The implementation need make no decisions regarding the set of open file descriptors when the child process image begins execution, those decisions having already been made by the caller and expressed as the set of open file descriptors and their FD_CLOEXEC flags at the time of the call and the spawn file actions object specified in the call. We have been assured that in cases where the POSIX *Start_Process* Ada primitives have been implemented in a library, this method of controlling file descriptor inheritance may be implemented very easily.

We can identify several problems with *posix_spawn*() and *posix_spawnp*(), but there does not appear to be a solution that introduces fewer problems. Environment modification for child process attributes not specifiable via the *attrp* or *file_actions* arguments must be done in the parent process, and since the parent generally wants to save its context, it is more costly than similar functionality with *fork*()/*exec*. It is also complicated to modify the environment of a multi-threaded process temporarily, since all threads must agree when it is safe for the environment to be changed. However, this cost is only borne by those invocations of *posix_spawn*() and *posix_spawnp*() that use the additional functionality. Since extensive modifications are not the usual case, and are particularly unlikely in time-critical code, keeping much of the environment control out of *posix_spawn*() and *posix_spawnp*() is appropriate design.

The *posix_spawn*() and *posix_spawnp*() functions do not have all the power of *fork*()/*exec*. This is to be expected. The *fork*() function is a wonderfully powerful operation. We do not expect to duplicate its functionality in a simple, fast function with no special hardware requirements. It is worth noting that *posix_spawn*() and *posix_spawnp*() are very similar to the process creation operations on many operating systems that are not UNIX systems.

Requirements

The requirements for *posix_spawn*() and *posix_spawnp*() are:

- They must be implementable without an MMU or unusual hardware.
- They must be compatible with existing POSIX standards.

Additional goals are:

- They should be efficiently implementable.
- They should be able to replace at least 50% of typical executions of *fork*().
- A system with *posix_spawn*() and *posix_spawnp*() and without *fork*() should be useful, at least for realtime applications.
- A system with *fork*() and the *exec* family should be able to implement *posix_spawn*() and *posix_spawnp*() as library routines.

Two-Syntax

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.

The parameter list does not differ between *posix_spawn*( ) and *posix_spawnp*( ); *posix_spawnp*( ) interprets the second parameter more elaborately than *posix_spawn*( ).

Compatibility with POSIX.5 (Ada)

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.

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*( ).

Process Group

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.

Threads

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.

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 continue to be AEPs for operating systems with only one process.

Asynchronous Error Notification

A library implementation of *posix_spawn*( ) or *posix_spawnp*( ) may not be able to detect all possible errors before it forks the child process. IEEE Std. 1003.1-200x provides for an error indication returned from a child process which could not successfully complete the spawn operation via a special exit status which may be detected using the status value returned by *wait*( ) and *waitpid*( ).

The *stat_val* interface and the macros used to interpret it are not well suited to the purpose of returning API errors, but they are the only path available to a library implementation. Thus, an implementation may cause the child process to exit with exit status 127 for any error detected during the spawn process after the *posix_spawn*( ) or *posix_spawnp*( ) function has successfully returned.

The standard developers had proposed using two additional macros to interpret *stat_val*. The first, WIFSPAWNFAIL, would have detected a status that indicated that the child exited because of an error detected during the *posix_spawn*( ) or *posix_spawnp*( ) operations rather than during actual execution of the child process image; the second, WSPAWNERRNO, would have extracted the error value if WIFSPAWNFAIL indicated a failure. Unfortunately, the ballot group strongly opposed this because it would make a library implementation of *posix_spawn*( ) or *posix_spawnp*( ) dependent on kernel modifications to *waitpid*( ) to be able to embed special information in *stat_val* to indicate a spawn failure.

The 8 bits of child process exit status that are guaranteed by IEEE Std. 1003.1-200x to be accessible to the waiting parent process are insufficient to disambiguate a spawn error from any other kind of error that may be returned by an arbitrary process image. No other bits of the exit status are required to be visible in *stat_val*, so these macros could not be strictly implemented at the library level. Reserving an exit status of 127 for such spawn errors is consistent with the use of this value by *system*( ) and *popen*( ) to signal failures in these operations that occur after the function has returned but before a shell is able to execute. The exit status of 127 does not uniquely identify this class of error, nor does it provide any detailed information on the nature of the failure. Note that a kernel implementation of *posix_spawn*( ) or *posix_spawnp*( ) is permitted (and encouraged) to return any possible error as the function value, thus providing more detailed failure information to the parent process.

Thus, no special macros are available to isolate asynchronous *posix_spawn*( ) or *posix_spawnp*( ) errors. Instead, errors detected by the *posix_spawn*( ) or *posix_spawnp*( ) operations in the context of the child process before the new process image executes are reported by setting the child’s exit status to 127. The calling process may use the WIFEXITED and WEXITSTATUS macros on the *stat_val* stored by the *wait*( ) or *waitpid*( ) functions to detect spawn failures to the extent that other status values with which the child process image may exit (before the parent can conclusively determine that the child process image has begun execution) are distinct from exit status 127.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *chmod*( ), *close*( ), *dup*( ), *exec*, *exit*( ), *fcntl*( ), *fork*( ), *kill*( ), *open*( ), *posix_spawn_file_actions_addclose*( ), *posix_spawn_file_actions_adddup2*( ), *posix_spawn_file_actions_addopen*( ), *posix_spawn_file_actions_destroy*( ), *posix_spawn_file_actions_init*( ), *posix_spawnattr_destroy*( ), *posix_spawnattr_init*( ),

28865 *posix_spawnattr_getsigdefault*( ), *posix_spawnattr_getflags*( ), *posix_spawnattr_getpgroup*( ), |
28866 *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_getsigmask*( ), |
28867 *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ), *posix_spawnattr_setpgroup*( ), |
28868 *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ),
28869 *sched_setparam*( ), *sched_setscheduler*( ), *setpgid*( ), *setuid*( ), *stat*( ), *times*( ), *wait*( ), the Base |
28870 Definitions volume of IEEE Std. 1003.1-200x, **<spawn.h>** |

28871 **CHANGE HISTORY**

28872 First released in Issue 6. Derived from IEEE Std. 1003.1d-1999.

28873 IEEE PASC Interpretation 1003.1 #103 is included, noting that the signal default actions are |
28874 changed as well as the signal mask in step 2. |

NAME

posix_spawn_file_actions_addclose, posix_spawn_file_actions_addopen — add close or open action to spawn file actions object (**REALTIME**)

SYNOPSIS

SPN

```
#include <spawn.h>

int posix_spawn_file_actions_addclose(posix_spawn_file_actions_t *
    file_actions, int fildes);
int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *restrict
    file_actions, int fildes, const char *restrict path,
    int oflag, mode_t mode);
```

DESCRIPTION

A spawn file actions object is of type **posix_spawn_file_actions_t** (defined in **<spawn.h>**) and is used to specify a series of actions to be performed by a *posix_spawn*() or *posix_spawnp*() operation in order to arrive at the set of open file descriptors for the child process given the set of open file descriptors of the parent. IEEE Std. 1003.1-200x does not define comparison or assignment operators for the type **posix_spawn_file_actions_t**.

A spawn file actions object, when passed to *posix_spawn*() or *posix_spawnp*(), shall specify how the set of open file descriptors in the calling process is transformed into a set of potentially open file descriptors for the spawned process. This transformation shall be as if the specified sequence of actions was performed exactly once, in the context of the spawned process (prior to execution of the new process image), in the order in which the actions were added to the object; additionally, when the new process image is executed, any file descriptor (from this new set) which has its FD_CLOEXEC flag set will be closed (see *posix_spawn*()).

The *posix_spawn_file_actions_addclose*() function adds a *close* action to the object referenced by *file_actions* that will cause the file descriptor *fildes* to be closed (as if *close*(*fildes*) had been called) when a new process is spawned using this file actions object.

The *posix_spawn_file_actions_addopen*() function adds an *open* action to the object referenced by *file_actions* that will cause the file named by *path* to be opened (as if *open*(*path*, *oflag*, *mode*) had been called, and the returned file descriptor, if not *fildes*, had been changed to *fildes*) when a new process is spawned using this file actions object. If *fildes* was already an open file descriptor, it shall be closed before the new file is opened.

Notes to Reviewers

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

D3, XSH, ERN 448 says the description of the posix_spawn_file_actions_addopen function does not say whether the function has to make a copy of the path parameter or whether it can store the pointer and assume the application does not destroy the copy of the string. Add to the description: "The string pointed to by path can become invalid so the function has to make a copy."

RETURN VALUE

Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions shall fail if:

[EBADF] The value specified by *fildes* is negative or greater than or equal to {OPEN_MAX}.

These functions may fail if:

[EINVAL] The value specified by *file_actions* is invalid.

[ENOMEM] Insufficient memory exists to add to the spawn file actions object.

It shall not be considered an error for the *fildes* argument passed to these functions to specify a file descriptor for which the specified operation could not be performed at the time of the call. Any such error will be detected when the associated file actions object is later used during a *posix_spawn*( ) or *posix_spawnp*( ) operation.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations. |

RATIONALE

A spawn file actions object may be initialized to contain an ordered sequence of *close*( ), *dup2*( ), and *open*( ) operations to be used by *posix_spawn*( ) or *posix_spawnp*( ) to arrive at the set of open file descriptors inherited by the spawned process from the set of open file descriptors in the parent at the time of the *posix_spawn*( ) or *posix_spawnp*( ) call. It had been suggested that the *close*( ) and *dup2*( ) operations alone are sufficient to rearrange file descriptors, and that files which need to be opened for use by the spawned process can be handled either by having the calling process open them before the *posix_spawn*( ) or *posix_spawnp*( ) call (and close them after), or by passing file names to the spawned process (in *argv*) so that it may open them itself. The standard developers recommend that applications use one of these two methods when practical, since detailed error status on a failed open operation is always available to the application this way. However, the standard developers feel that allowing a spawn file actions object to specify open operations is still appropriate because:

1. It is consistent with equivalent POSIX.5 (Ada) functionality.
2. It supports the I/O redirection paradigm commonly employed by POSIX programs designed to be invoked from a shell. When such a program is the child process, it may not be designed to open files on its own.
3. It allows file opens that might otherwise fail or violate file ownership/access rights if executed by the parent process.

Regarding 2. above, note that the spawn open file action provides to *posix_spawn*( ) and *posix_spawnp*( ) the same capability that the shell redirection operators provide to *system*( ), only without the intervening execution of a shell; for example:

```
system ("myprog <file1 3<file2");
```

Regarding 3. above, note that if the calling process needs to open one or more files for access by the spawned process, but has insufficient spare file descriptors, then the open action is necessary to allow the *open*( ) to occur in the context of the child process after other file descriptors have been closed (that must remain open in the parent).

Additionally, if a parent is executed from a file having a ‘‘set-user-id’’ mode bit set and the POSIX_SPAWN_RESETIDS flag is set in the spawn attributes, a file created within the parent process will (possibly incorrectly) have the parent’s effective user ID as its owner, whereas a file created via an *open*( ) action during *posix_spawn*( ) or *posix_spawnp*( ) will have the parent’s real ID as its owner; and an open by the parent process may successfully open a file to which the real user should not have access or fail to open a file to which the real user should have access.

File Descriptor Mapping

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_spawnp*( ) 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.

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.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *dup*( ), *open*( ), *posix_spawn*( ), *posix_spawn_file_actions_adddup2*( ), *posix_spawn_file_actions_destroy*( ), *posix_spawnp*( ), 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.

28988 **NAME**

28989 posix_spawn_file_actions_adddup2 — add dup2 action to spawn file actions object |
28990 (**REALTIME**) |

28991 **SYNOPSIS**

28992 SPN
```
#include <spawn.h>
```

28993
```
int posix_spawn_file_actions_adddup2(posix_spawn_file_actions_t *
    file_actions, int fildes, int newfildes);
```

28995

28996 **DESCRIPTION**

28997 A spawn file actions object is as defined in *posix_spawn_file_actions_addclose*( ).

28998 The *posix_spawn_file_actions_adddup2*( ) function adds a *dup2*( ) action to the object referenced by
28999 *file_actions* that will cause the file descriptor *fildes* to be duplicated as *newfildes* (as if *dup2*(*fildes*,
29000 *newfildes*) had been called) when a new process is spawned using this file actions object. |

29001 **RETURN VALUE** |

29002 Upon successful completion, the *posix_spawn_file_actions_adddup2*( ) function shall return zero; |
29003 otherwise, an error number shall be returned to indicate the error.

29004 **ERRORS**

29005 The *posix_spawn_file_actions_adddup2*( ) function shall fail if:

29006 [EBADF] The value specified by *fildes* or *newfildes* is negative or greater than or equal to |
29007 {OPEN_MAX}.

29008 [ENOMEM] Insufficient memory exists to add to the spawn file actions object.

29009 The *posix_spawn_file_actions_adddup2*( ) function may fail if:

29010 [EINVAL] The value specified by *file_actions* is invalid.

29011 It shall not be considered an error for the *fildes* argument passed to the
29012 *posix_spawn_file_actions_adddup2*( ) function to specify a file descriptor for which the specified
29013 operation could not be performed at the time of the call. Any such error will be detected when
29014 the associated file actions object is later used during a *posix_spawn*( ) or *posix_spawnp*( )
29015 operation.

29016 **EXAMPLES**

29017 None.

29018 **APPLICATION USAGE**

29019 The *posix_spawn_file_actions_adddup2*( ) function is part of the Spawn option and need not be |
29020 provided on all implementations.

29021 **RATIONALE**

29022 Refer to the RATIONALE in *posix_spawn_file_actions_addclose*( ).

29023 **FUTURE DIRECTIONS**

29024 None.

29025 **SEE ALSO**

29026 *dup*( ), *posix_spawn*( ), *posix_spawn_file_actions_addclose*( ), *posix_spawn_file_actions_destroy*( ),
29027 *posix_spawnp*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<spawn.h>** |

29028 **CHANGE HISTORY**

29029 First released in Issue 6. Derived from IEEE Std. 1003.1d-1999.

29030 IEEE PASC Interpretation 1003.1 #104 is included, noting that the [EBADF] error can apply to the |
29031 *newfildes* argument in addition to *fildes*. |

29032 **NAME**

29033 posix_spawn_file_actions_addopen — add open action to spawn file actions object |
29034 (**REALTIME**) |

29035 **SYNOPSIS**

29036 SPN
```
#include <spawn.h>
```

29037
```
int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *restrict|
    file_actions, int fildes, const char *restrict path,                  |
    int oflag, mode_t mode);                                              |
```
29038
29039
29040 |

29041 **DESCRIPTION**

29042 Refer to *posix_spawn_file_actions_addclose*( ).

29043 **NAME**

29044 posix_spawn_file_actions_destroy, posix_spawn_file_actions_init — destroy and initialize
29045 spawn file actions object (**REALTIME**)

29046 **SYNOPSIS**

29047 SPN
```
#include <spawn.h>

int posix_spawn_file_actions_destroy(posix_spawn_file_actions_t *
    file_actions);
int posix_spawn_file_actions_init(posix_spawn_file_actions_t *
    file_actions);
```
29052

29053 **DESCRIPTION**

29054 A spawn file actions object is as defined in *posix_spawn_file_actions_addclose*( ).

29055 The *posix_spawn_file_actions_destroy*( ) function destroys the object referenced by *file_actions*; the
29056 object becomes, in effect, uninitialized. An implementation may cause
29057 *posix_spawn_file_actions_destroy*( ) to set the object referenced by *file_actions* to an invalid value. A
29058 destroyed spawn file actions object can be reinitialized using *posix_spawn_file_actions_init*( ); the
29059 results of otherwise referencing the object after it has been destroyed are undefined.

29060 The *posix_spawn_file_actions_init*( ) function initializes the object referenced by *file_actions* to
29061 contain no file actions for *posix_spawn*( ) or *posix_spawnp*( ) to perform. |

29062 The effect of initializing an already initialized spawn file actions object is undefined. |

29063 **RETURN VALUE** |

29064 Upon successful completion, these functions shall return zero; otherwise, an error number shall |
29065 be returned to indicate the error.

29066 **ERRORS**

29067 The *posix_spawn_file_actions_init*( ) function shall fail if:

29068 [ENOMEM] Insufficient memory exists to initialize the spawn file actions object.

29069 The *posix_spawn_file_actions_destroy*( ) function may fail if:

29070 [EINVAL] The value specified by *file_actions* is invalid.

29071 **EXAMPLES**

29072 None.

29073 **APPLICATION USAGE**

29074 These functions are part of the Spawn option and need not be provided on all implementations. |

29075 **RATIONALE**

29076 Refer to the RATIONALE in *posix_spawn_file_actions_addclose*( ).

29077 **FUTURE DIRECTIONS**

29078 None.

29079 **SEE ALSO**

29080 *posix_spawn*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<spawn.h>** |

29081 **CHANGE HISTORY**

29082 First released in Issue 6. Derived from IEEE Std. 1003.1d-1999.

29083 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

29084 **NAME**

29085 posix_spawn_file_actions_init — initialize spawn file actions object (**REALTIME**)

29086 **SYNOPSIS**

29087 SPN
```
#include <spawn.h>
```

29088
```
int posix_spawn_file_actions_init(posix_spawn_file_actions_t *
```
29089
```
    file_actions);
```

29090

29091 **DESCRIPTION**

29092 Refer to *posix_spawn_file_actions_destroy*( ).

NAME

posix_spawnattr_destroy, posix_spawnattr_init — destroy and initialize spawn attributes object (**REALTIME**)

SYNOPSIS

SPN

```
#include <spawn.h>

int posix_spawnattr_destroy(posix_spawnattr_t *attr);
int posix_spawnattr_init(posix_spawnattr_t *attr);
```

DESCRIPTION

A spawn attributes object is of type **posix_spawnattr_t** (defined in **<spawn.h>**) and is used to specify the inheritance of process attributes across a spawn operation. IEEE Std. 1003.1-200x does not define comparison or assignment operators for the type **posix_spawnattr_t**.

The *posix_spawnattr_destroy*( ) function destroys a spawn attributes object. The effect of subsequent use of the object is undefined until the object is reinitialized by another call to *posix_spawnattr_init*( ). An implementation may cause *posix_spawnattr_destroy*( ) to set the object referenced by *attr* to an invalid value.

The *posix_spawnattr_init*( ) function initializes a spawn attributes object *attr* with the default value for all of the individual attributes used by the implementation. The effect of initializing an already initialized spawn attributes option is undefined.

Each implementation shall document the individual attributes it uses and their default values unless these values are defined by IEEE Std. 1003.1-200x. Attributes not defined by IEEE Std. 1003.1-200x, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined.

The resulting spawn attributes object (possibly modified by setting individual attribute values), is used to modify the behavior of *posix_spawn*( ) or *posix_spawnp*( ). After a spawn attributes object has been used to spawn a process by a call to a *posix_spawn*( ) or *posix_spawnp*( ), any function affecting the attributes object (including destruction) does not affect any process that has been spawned in this way.

RETURN VALUE

Upon successful completion, *posix_spawnattr_destroy*( ) and *posix_spawnattr_init*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *posix_spawnattr_init*( ) function shall fail if:

[ENOMEM] Insufficient memory exists to initialize the spawn attributes object.

The *posix_spawnattr_destroy*( ) function may fail if:

[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_spawnp*( ), 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 included, noting that the effect of initializing an already initialized spawn attributes option is undefined.

29157 **NAME**

29158 posix_spawnattr_getflags, posix_spawnattr_setflags — get and set spawn-flags attribute of
29159 spawn attributes object (**REALTIME**)

29160 **SYNOPSIS**

29161 SPN
```
#include <spawn.h>

int posix_spawnattr_getflags(const posix_spawnattr_t *restrict attr,
    short *restrict flags);
int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
```
29165

29166 **DESCRIPTION**

29167 The *spawn-flags* attribute is used to indicate which process attributes are to be changed in the
29168 new process image when invoking *posix_spawn*( ) or *posix_spawnp*( ). It is the bitwise-inclusive
29169 OR of zero or more of the flags POSIX_SPAWN_RESETIDS, POSIX_SPAWN_SETPGROUP,
29170 PS POSIX_SPAWN_SETSIGDEF, and POSIX_SPAWN_SETSIGMASK,
29171 POSIX_SPAWN_SETSCHEDPARAM, and POSIX_SPAWN_SETSCHEDULER. In addition, if the
29172 Process Scheduling option is supported, the flags POSIX_SPAWN_SETSCHEDPARAM and
29173 POSIX_SPAWN_SETSCHEDULER shall also be supported. These flags are defined in
29174 **<spawn.h>**. The default value of this attribute shall be with no flags set.

29175 The *posix_spawnattr_getflags*( ) function obtains the value of the *spawn-flags* attribute from the
29176 attributes object referenced by *attr*.

29177 The *posix_spawnattr_setflags*( ) function is used to set the *spawn-flags* attribute in an initialized
29178 attributes object referenced by *attr*.

29179 **RETURN VALUE**

29180 Upon successful completion, *posix_spawnattr_getflags*( ) shall return zero and store the value of
29181 the *spawn-flags* attribute of *attr* into the object referenced by the *flags* parameter; otherwise, an
29182 error number shall be returned to indicate the error.

29183 Upon successful completion, *posix_spawnattr_setflags*( ) shall return zero; otherwise, an error
29184 number shall be returned to indicate the error.

29185 **ERRORS**

29186 These functions may fail if:

29187 [EINVAL] The value specified by *attr* is invalid.

29188 The *posix_spawnattr_setflags*( ) function may fail if:

29189 [EINVAL] The value of the attribute being set is not valid.

29190 **EXAMPLES**

29191 None.

29192 **APPLICATION USAGE**

29193 These functions are part of the Spawn option and need not be provided on all implementations.

29194 **RATIONALE**

29195 None.

29196 **FUTURE DIRECTIONS**

29197 None.

29198 **SEE ALSO**

29199 *posix_spawn*( ), ,=1 .ds ;p *posix_spawnattr_destroy*( ) (on page 1405), ,=1 .ds ;p
29200 *posix_spawnattr_init*( ) (on page 1419), *posix_spawnattr_getsigdefault*( ), |
29201 *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ),
29202 *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setpgroup*( ), |
29203 *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ),
29204 *posix_spawnp*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<spawn.h>** |

29205 **CHANGE HISTORY**

29206 First released in Issue 6. Derived from IEEE Std. 1003.1d-1999.

NAME

posix_spawnattr_getpgroup, posix_spawnattr_setpgroup — get and set spawn-pgroup attribute of spawn attributes object (**REALTIME**)

SYNOPSIS

SPN

```
#include <spawn.h>

int posix_spawnattr_getpgroup(const posix_spawnattr_t *restrict attr,
    pid_t *restrict pgroup);
int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
```

DESCRIPTION

The *spawn-pgroup* attribute represents the process group to be joined by the new process image in a spawn operation (if POSIX_SPAWN_SETPGROUP is set in the *spawn-flags* attribute). The default value of this attribute shall be zero.

The *posix_spawnattr_getpgroup*( ) function obtains the value of the *spawn-pgroup* attribute from the attributes object referenced by *attr*.

The *posix_spawnattr_setpgroup*( ) function is used to set the *spawn-pgroup* attribute in an initialized attributes object referenced by *attr*.

RETURN VALUE

Upon successful completion, *posix_spawnattr_getpgroup*( ) shall return zero and store the value of the *spawn-pgroup* attribute of *attr* into the object referenced by the *pgroup* parameter; otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_spawnattr_setpgroup*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EINVAL] The value specified by *attr* is invalid.

The *posix_spawnattr_setpgroup*( ) function may fail if:

[EINVAL] The value of the attribute being set is not valid.

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_getflags*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ), *posix_spawnp*( ), 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.

NAME

posix_spawnattr_getschedparam, posix_spawnattr_setschedparam — get and set spawn-schedparam attribute of spawn attributes object (**REALTIME**)

SYNOPSIS

SPN PS
```
#include <spawn.h>
#include <sched.h>

int posix_spawnattr_getschedparam(const posix_spawnattr_t *restrict attr,
    struct sched_param *restrict schedparam);
int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,
    const struct sched_param *restrict schedparam);
```

DESCRIPTION

The *spawn-schedparam* attribute represents the scheduling parameters to be assigned to the new process image in a spawn operation (if POSIX_SPAWN_SETSCHEDULER or POSIX_SPAWN_SETSCHEDPARAM is set in the *spawn-flags* attribute). The default value of this attribute is unspecified.

The *posix_spawnattr_getschedparam*( ) function obtains the value of the *spawn-schedparam* attribute from the attributes object referenced by *attr*.

The *posix_spawnattr_setschedparam*( ) function is used to set the *spawn-schedparam* attribute in an initialized attributes object referenced by *attr*.

RETURN VALUE

Upon successful completion, *posix_spawnattr_getschedparam*( ) shall return zero and store the value of the *spawn-schedparam* attribute of *attr* into the object referenced by the *schedparam* parameter; otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_spawnattr_setschedparam*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EINVAL] The value specified by *attr* is invalid.

The *posix_spawnattr_setschedparam*( ) function may fail if:

[EINVAL] The value of the attribute being set is not valid.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn and Process Scheduling options 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_getflags*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ),

29295 *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ),
29296 *posix_spawnp*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sched.h>**, **<spawn.h>**

29297 **CHANGE HISTORY**

29298 First released in Issue 6. Derived from IEEE Std. 1003.1d-1999.

29299 **NAME**

29300 posix_spawnattr_getschedpolicy, posix_spawnattr_setschedpolicy — get and set spawn-
29301 schedpolicy attribute of spawn attributes object (**REALTIME**)

29302 **SYNOPSIS**

29303 SPN PS
```
#include <spawn.h>
#include <sched.h>

int posix_spawnattr_getschedpolicy(const posix_spawnattr_t *restrict attr,
    int *restrict schedpolicy);
int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
    int schedpolicy);
```
29309

29310 **DESCRIPTION**

29311 The *spawn-schedpolicy* attribute represents the scheduling policy to be assigned to the new
29312 process image in a spawn operation (if POSIX_SPAWN_SETSCHEDULER is set in the *spawn-*
29313 *flags* attribute). The default value of this attribute is unspecified.

29314 The *posix_spawnattr_getschedpolicy*( ) function obtains the value of the *spawn-schedpolicy* attribute
29315 from the attributes object referenced by *attr*.

29316 The *posix_spawnattr_setschedpolicy*( ) function is used to set the *spawn-schedpolicy* attribute in an
29317 initialized attributes object referenced by *attr*.

29318 **RETURN VALUE**

29319 Upon successful completion, *posix_spawnattr_getschedpolicy*( ) shall return zero and store the
29320 value of the *spawn-schedpolicy* attribute of *attr* into the object referenced by the *schedpolicy*
29321 parameter; otherwise, an error number shall be returned to indicate the error.

29322 Upon successful completion, *posix_spawnattr_setschedpolicy*( ) shall return zero; otherwise, an
29323 error number shall be returned to indicate the error.

29324 **ERRORS**

29325 These functions may fail if:

29326 [EINVAL] The value specified by *attr* is invalid.

29327 The *posix_spawnattr_setschedpolicy*( ) function may fail if:

29328 [EINVAL] The value of the attribute being set is not valid.

29329 **EXAMPLES**

29330 None.

29331 **APPLICATION USAGE**

29332 These functions are part of the Spawn and Process Scheduling options and need not be provided
29333 on all implementations.

29334 **RATIONALE**

29335 None.

29336 **FUTURE DIRECTIONS**

29337 None.

29338 **SEE ALSO**

29339 *posix_spawn*( ), *posix_spawnattr_destroy*( ), *posix_spawnattr_init*( ), *posix_spawnattr_getsigdefault*( ),
29340 *posix_spawnattr_getflags*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ),
29341 *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ),
29342 *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setsigmask*( ),

posix_spawnp( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sched.h>**, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std. 1003.1d-1999.

NAME

posix_spawnattr_getsigdefault, posix_spawnattr_setsigdefault — get and set spawn-sigdefault attribute of spawn attributes object (**REALTIME**)

SYNOPSIS

SPN

```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_getsigdefault(const posix_spawnattr_t *restrict attr,
    sigset_t *restrict sigdefault);
int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr,
    const sigset_t *restrict sigdefault);
```

DESCRIPTION

The *spawn-sigdefault* attribute represents the set of signals to be forced to default signal handling in the new process image (if POSIX_SPAWN_SETSIGDEF is set in the *spawn-flags* attribute) by a spawn operation. The default value of this attribute shall be an empty signal set.

The *posix_spawnattr_getsigdefault*( ) function obtains the value of the *spawn-sigdefault* attribute from the attributes object referenced by *attr*.

The *posix_spawnattr_setsigdefault*( ) function is used to set the *spawn-sigdefault* attribute in an initialized attributes object referenced by *attr*.

RETURN VALUE

Upon successful completion, *posix_spawnattr_getsigdefault*( ) shall return zero and store the value of the *spawn-sigdefault* attribute of *attr* into the object referenced by the *sigdefault* parameter; otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_spawnattr_setsigdefault*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EINVAL] The value specified by *attr* is invalid.

The *posix_spawnattr_setsigdefault*( ) function may fail if:

[EINVAL] The value of the attribute being set is not valid.

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_getflags*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setflags*( ), *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<signal.h>**, **<spawn.h>**

29390 **CHANGE HISTORY**
29391 First released in Issue 6. Derived from IEEE Std. 1003.1d-1999.

NAME

posix_spawnattr_getsigmask, posix_spawnattr_setsigmask — get and set spawn-sigmask attribute of spawn attributes object (**REALTIME**)

SYNOPSIS

SPN

```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_getsigmask(const posix_spawnattr_t *restrict attr,
    sigset_t *restrict sigmask);
int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr,
    const sigset_t *restrict sigmask);
```

DESCRIPTION

The *spawn-sigmask* attribute represents the signal mask in effect in the new process image of a spawn operation (if POSIX_SPAWN_SETSIGMASK is set in the *spawn-flags* attribute). The default value of this attribute is unspecified.

The *posix_spawnattr_getsigmask*( ) function obtains the value of the *spawn-sigmask* attribute from the attributes object referenced by *attr*.

The *posix_spawnattr_setsigmask*( ) function is used to set the *spawn-sigmask* attribute in an initialized attributes object referenced by *attr*.

RETURN VALUE

Upon successful completion, *posix_spawnattr_getsigmask*( ) shall return zero and store the value of the *spawn-sigmask* attribute of *attr* into the object referenced by the *sigmask* parameter; otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_spawnattr_setsigmask*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EINVAL] The value specified by *attr* is invalid.

The *posix_spawnattr_setsigmask*( ) function may fail if:

[EINVAL] The value of the attribute being set is not valid.

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_getflags*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ), *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<signal.h>**, **<spawn.h>**

29436 **CHANGE HISTORY**

29437 First released in Issue 6. Derived from IEEE Std. 1003.1d-1999.

29438 **NAME**

29439 posix_spawnattr_init — initialize spawn attributes object (**REALTIME**)

29440 **SYNOPSIS**

29441 SPN
```
#include <spawn.h>
```

29442
```
int posix_spawnattr_init(posix_spawnattr_t *attr);
```

29443

29444 **DESCRIPTION**

29445 Refer to *posix_spawnattr_destroy*( ).

29446 **NAME**

29447 posix_spawnattr_setflags — set spawn-flags attribute of spawn attributes object (**REALTIME**)

29448 **SYNOPSIS**

29449 SPN
```
#include <spawn.h>
```

29450
```
int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
```

29451

29452 **DESCRIPTION**

29453 Refer to *posix_spawnattr_getflags*( ).

29454 **NAME**

29455 posix_spawnattr_setpgroup — set spawn-pgroup attribute of spawn attributes object
29456 (**REALTIME**)

29457 **SYNOPSIS**

29458 SPN
```
#include <spawn.h>
```

29459
```
int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
```

29460

29461 **DESCRIPTION**

29462 Refer to *posix_spawnattr_getpgroup*( ).

29463 **NAME**

29464 posix_spawnattr_setschedparam — set spawn-schedparam attribute of spawn attributes object
29465 (**REALTIME**)

29466 **SYNOPSIS**

29467 SPN PS
```
#include <sched.h>
#include <spawn.h>

int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,
    const struct sched_param *restrict schedparam);
```

29471

29472 **DESCRIPTION**

29473 Refer to *posix_spawnattr_getschedparam*( ).

29474 **NAME**

29475 posix_spawnattr_setschedpolicy — set spawn-schedpolicy attribute of spawn attributes object
29476 (**REALTIME**)

29477 **SYNOPSIS**

29478 SPN PS
```
#include <sched.h>
#include <spawn.h>

int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
    int schedpolicy);
```
29479
29480
29481
29482

29483 **DESCRIPTION**

29484 Refer to *posix_spawnattr_getschedpolicy*( ).

29485 **NAME**

29486 posix_spawnattr_setsigdefault — set spawn-sigdefault attribute of spawn attributes object
29487 (**REALTIME**)

29488 **SYNOPSIS**

29489 SPN
```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr,
    const sigset_t *restrict sigdefault);
```
29493

29494 **DESCRIPTION**

29495 Refer to *posix_spawnattr_getsigdefault*( ).

29496 **NAME**

29497 posix_spawnattr_setsigmask — set spawn-sigmask attribute of spawn attributes object
29498 (**REALTIME**)

29499 **SYNOPSIS**

29500 SPN
```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr,
    const sigset_t *restrict sigmask);
```

29504

29505 **DESCRIPTION**

29506 Refer to *posix_spawnattr_getsigmask*( ).

29507 **NAME**

29508 posix_spawnp — spawn a process (**REALTIME**)

29509 **SYNOPSIS**

29510 SPN
```
#include <spawn.h>

int posix_spawnp(pid_t *restrict pid, const char *restrict file,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *restrict attrp,
    char *const argv[restrict], char *const envp[restrict]);
```

29516 **DESCRIPTION**

29517 Refer to *posix_spawn*( ).

29518 **NAME**

29519 posix_trace_attr_destroy, posix_trace_attr_init — trace stream attributes object destroy and
29520 initialization

29521 **SYNOPSIS**

29522 TRC
```
#include <trace.h>
```

29523
```
int posix_trace_attr_destroy(trace_attr_t *attr);
```
29524
```
int posix_trace_attr_init(trace_attr_t *attr);
```
29525

29526 **DESCRIPTION**

29527 The *posix_trace_attr_destroy*( ) function is used to destroy an initialized trace attributes object.
29528 The results of using the attributes object after it has been destroyed are unspecified. A destroyed
29529 trace attributes object can be reinitialized using *posix_trace_attr_init*( ).

29530 The *posix_trace_attr_init*( ) function initializes a trace attributes object *attr* with the default value
29531 for all of the individual attributes used by a given implementation. The read-only *generation-*
29532 *version* and *clock-resolution* attributes of the newly initialized trace attributes object shall be set to
29533 their appropriate values (Section 2.11.1.2 (on page 578)).

29534 The effect of initializing an already-initialized trace attributes object is unspecified.

29535 Implementations may add extensions to the trace attributes object structure as permitted in the
29536 Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 2, Conformance.

29537 The resulting attributes object (possibly modified by setting individual attributes values), when
29538 used by *posix_trace_create*( ), defines the attributes of the trace stream created. A single attributes
29539 object can be used in multiple calls to *posix_trace_create*( ). After one or more trace streams have
29540 been created using an attributes object, any function affecting that attributes object, including
29541 destruction, does not affect any trace stream previously created. An initialized attributes object
29542 also serves to receive the attributes of an existing trace stream or trace log when calling the
29543 *posix_trace_get_attr*( ) function.

29544 **RETURN VALUE**

29545 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
29546 return the corresponding error number.

29547 **ERRORS**

29548 The *posix_trace_attr_destroy*( ) function may fail if:

29549 [EINVAL] The value of *attr* is invalid.

29550 The *posix_trace_attr_init*( ) function shall fail if:

29551 [ENOMEM] Insufficient memory exists to initialize the trace attributes object.

29552 **EXAMPLES**

29553 None.

29554 **APPLICATION USAGE**

29555 None.

29556 **RATIONALE**

29557 None.

29558 **FUTURE DIRECTIONS**

29559 None.

29560 **SEE ALSO**

29561 *posix_trace_create*( ), *posix_trace_get_attr*( ), *uname*( ), the Base Definitions volume of
29562 IEEE Std. 1003.1-200x, **<trace.h>**

29563 **CHANGE HISTORY**

29564 First released in Issue 6. Derived from IEEE Std. 1003.1q-2000.

NAME

posix_trace_attr_getclockres, posix_trace_attr_getcreatetime, posix_trace_attr_getgenversion, posix_trace_attr_getname, posix_trace_attr_setname — retrieve and set information about a trace stream

SYNOPSIS

TRC
```
#include <time.h>
#include <trace.h>

int posix_trace_attr_getclockres(const trace_attr_t *attr,
    struct timespec *resolution);
int posix_trace_attr_getcreatetime(const trace_attr_t *attr,
    struct timespec *createtime);

#include <trace.h>

int posix_trace_attr_getgenversion(const trace_attr_t *attr,
    char *genversion);
int posix_trace_attr_getname(const trace_attr_t *attr,
    char *tracename);
int posix_trace_attr_setname(trace_attr_t *attr,
    const char *tracename);
```

DESCRIPTION

The *posix_trace_attr_getclockres*( ) function shall copy the clock resolution of the clock used to generate timestamps from the *clock-resolution* attribute of the attributes object pointed to by the *attr* argument into the structure pointed to by the *resolution* argument.

The *posix_trace_attr_getcreatetime*( ) function shall copy the trace stream creation time from the *creation-time* attribute of the attributes object pointed to by the *attr* argument into the structure pointed to by the *createtime* argument. The *creation-time* attribute shall represent the time of creation of the trace stream.

The *posix_trace_attr_getgenversion*( ) function shall copy the string containing version information from the *generation-version* attribute of the attributes object pointed to by the *attr* argument into the string pointed to by the *genversion* argument. The *genversion* argument shall be the address of a character array which can store at least {TRACE_NAME_MAX} characters.

The *posix_trace_attr_getname*( ) function shall copy the string containing the trace name from the *trace-name* attribute of the attributes object pointed to by the *attr* argument into the string pointed to by the *tracename* argument. The *tracename* argument shall be the address of a character array which can store at least {TRACE_NAME_MAX} characters.

The *posix_trace_attr_setname*( ) function shall set the name in the *trace-name* attribute of the attributes object pointed to by the *attr* argument, using the trace name string supplied by the *tracename* argument. If the supplied string contains more than {TRACE_NAME_MAX} characters, the name copied into the *trace-name* attribute may be truncated to one less than the length of {TRACE_NAME_MAX} characters. The default value is a null string.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

If successful, the *posix_trace_attr_getclockres*( ) function stores the *clock-resolution* attribute value in the object pointed to by *resolution*. Otherwise, the content of this object is unspecified.

If successful, the *posix_trace_attr_getcreatetime*( ) function stores the trace stream creation time in the object pointed to by *createtime*. Otherwise, the content of this object is unspecified.

If successful, the *posix_trace_attr_getgenversion*( ) function stores the trace version information in the string pointed to by *genversion*. Otherwise, the content of this string is unspecified.

If successful, the *posix_trace_attr_getname*( ) function stores the trace name in the string pointed to by *tracename*. Otherwise, the content of this string is unspecified.

ERRORS

The *posix_trace_attr_getclockres*( ), *posix_trace_attr_getcreatetime*( ), *posix_trace_attr_getgenversion*( ), and *posix_trace_attr_getname*( ) functions may fail if:

[EINVAL] The value specified by one of the arguments is invalid.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_attr_init( ), *posix_trace_create*( ), *posix_trace_get_attr*( ), *uname*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<time.h>**, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std. 1003.1q-2000.

NAME

posix_trace_attr_getinherited, posix_trace_attr_getlogfullpolicy, posix_trace_attr_getstreamfullpolicy, posix_trace_attr_setinherited, posix_trace_attr_setlogfullpolicy, posix_trace_attr_setstreamfullpolicy — retrieve and set the behavior of a trace stream

SYNOPSIS

TRC
```
#include <trace.h>
```

TRC TRI
```
int posix_trace_attr_getinherited(const trace_attr_t *attr,
    int *inheritancepolicy);
```
TRC TRL
```
int posix_trace_attr_getlogfullpolicy(const trace_attr_t *attr,
    int *logpolicy);
```
TRC
```
int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *attr,
    int *streampolicy);
```
TRC TRI
```
int posix_trace_attr_setinherited(trace_attr_t *attr,
    int inheritancepolicy);
```
TRC TRL
```
int posix_trace_attr_setlogfullpolicy(trace_attr_t *attr,
    int logpolicy);
```
TRC
```
int posix_trace_attr_setstreamfullpolicy(trace_attr_t *attr,
    int streampolicy);
```

DESCRIPTION

TRI The *posix_trace_attr_getinherited*( ) and *posix_trace_attr_setinherited*( ) functions, respectively, get and set the inheritance policy stored in the *inheritance* attribute for traced processes across the *fork*( ) and *spawn*( ) operations. The *inheritance* attribute of the attributes object pointed to by the *attr* argument shall be set to one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_CLOSE_FOR_CHILD
: After a *fork*( ) or *spawn*( ) operation, the child shall not be traced, and tracing of the parent shall continue.

POSIX_TRACE_INHERITED
: After a *fork*( ) or *spawn*( ) operation, if the parent is being traced, its child shall be concurrently traced using the same trace stream.

The default value for the *inheritance* attribute is POSIX_TRACE_CLOSE_FOR_CHILD.

TRL The *posix_trace_attr_getlogfullpolicy*( ) and *posix_trace_attr_setlogfullpolicy*( ) functions, respectively, get and set the trace log full policy stored in the *log-full-policy* attribute of the attributes object pointed to by the *attr* argument.

The *log-full-policy* attribute shall be set to one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_LOOP
: The trace log shall loop until the associated trace stream is stopped. This policy means that when the trace log gets full, the file system shall reuse the resources allocated to the oldest trace events that were recorded. In this way, the trace log will always contain the most recent trace events flushed.

POSIX_TRACE_UNTIL_FULL
: The trace stream shall be flushed to the trace log until the trace log is full. This condition can be deduced from the *posix_log_full_status* member status (see the *posix_trace_status_info*( ) function). The last recorded trace event shall be the POSIX_TRACE_STOP trace event.

29680 POSIX_TRACE_APPEND
29681 The associated trace stream shall be flushed to the trace log without log size limitation. If
29682 the application specifies POSIX_TRACE_APPEND, the implementation shall ignore the
29683 *log-max-size* attribute.

29684 The default value for the *log-full-policy* attribute is POSIX_TRACE_LOOP.

29685 The *posix_trace_attr_getstreamfullpolicy*( ) and *posix_trace_attr_setstreamfullpolicy*( ) functions,
29686 respectively, get and set the trace stream full policy stored in the *stream-full-policy* attribute of the
29687 attributes object pointed to by the *attr* argument.

29688 The *stream-full-policy* attribute shall be set to one of the following values defined by manifest
29689 constants in the **<trace.h>** header:

29690 POSIX_TRACE_LOOP
29691 The trace stream shall loop until explicitly stopped by the *posix_trace_stop*( ) function. This
29692 policy means that when the trace stream is full, the trace system shall reuse the resources
29693 allocated to the oldest trace events recorded. In this way, the trace stream will always
29694 contain the most recent trace events recorded.

29695 POSIX_TRACE_UNTIL_FULL
29696 The trace stream will run until the trace stream resources are exhausted. Then the trace
29697 stream will stop. This condition can be deduced from *posix_stream_status* and
29698 *posix_stream_full_status* statuses (see the *posix_trace_status_info*( ) function). When this trace
29699 stream is read, a POSIX_TRACE_STOP trace event shall be reported after reporting the last
29700 recorded trace event. The trace system shall reuse the resources allocated to any trace
29701 events already reported—see the *posix_trace_getnext_event*( ), *posix_trace_trygetnext_event*( ),
29702 and *posix_trace_timedgetnext_event*( ) functions—or already flushed for an active trace stream
29703 with log if the Trace Log option is supported; see the *posix_trace_flush*( ) function. The trace
29704 system shall restart the trace stream when it is empty and may restart it sooner. A
29705 POSIX_TRACE_START trace event shall be reported before reporting the next recorded
29706 trace event.

29707 POSIX_TRACE_FLUSH
29708 If the Trace Log option is supported, this policy is identical to the
29709 POSIX_TRACE_UNTIL_FULL trace stream full policy except that the trace stream shall be
29710 flushed regularly as if *posix_trace_flush*( ) had been explicitly called. Defining this policy for
29711 an active trace stream without log shall be invalid.

29712 The default value for the *stream-full-policy* attribute shall be POSIX_TRACE_LOOP for an active
29713 trace stream without log.

29714 If the Trace Log option is supported, the default value for the *stream-full-policy* attribute shall be
29715 POSIX_TRACE_FLUSH for an active trace stream with log.

29716 **RETURN VALUE**

29717 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
29718 return the corresponding error number.

29719 TRI If successful, the *posix_trace_attr_getinherited*( ) function stores the *inheritance* attribute value in
29720 the object pointed to by *inheritancepolicy*. Otherwise, the content of this object is undefined.

29721 TRL If successful, the *posix_trace_attr_getlogfullpolicy*( ) function stores the *log-full-policy* attribute
29722 value in the object pointed to by *logpolicy*. Otherwise, the content of this object is undefined.

29723 If successful, the *posix_trace_attr_getstreamfullpolicy*( ) function stores the *stream-full-policy*
29724 attribute value in the object pointed to by *streampolicy*. Otherwise, the content of this object is
29725 undefined.

29726 **ERRORS**

29727 These functions may fail if:

29728 [EINVAL] The value specified by at least one of the arguments is invalid.

29729 **EXAMPLES**

29730 None.

29731 **APPLICATION USAGE**

29732 None.

29733 **RATIONALE**

29734 None.

29735 **FUTURE DIRECTIONS**

29736 None.

29737 **SEE ALSO**

29738 *fork*( ), *posix_trace_attr_init*( ), *posix_trace_create*( ), *posix_trace_flush*( ), *posix_trace_get_attr*( ),
29739 *posix_trace_getnext_event*( ), *posix_trace_start*( ), **posix_trace_status_info Structure**,
29740 *posix_trace_timedgetnext_event*( ), <REFERENCE UNDEFINED>(spawn), the Base Definitions
29741 volume of IEEE Std. 1003.1-200x, **<trace.h>**

29742 **CHANGE HISTORY**

29743 First released in Issue 6. Derived from IEEE Std. 1003.1q-2000.

29744 **NAME**

29745 posix_trace_attr_getlogsize, posix_trace_attr_getmaxdatasize,
29746 posix_trace_attr_getmaxsystemeventsize, posix_trace_attr_getmaxuserevent size,
29747 posix_trace_attr_getstreamsize, posix_trace_attr_setlogsize, posix_trace_attr_setmaxdatasize,
29748 posix_trace_attr_setstreamsize — retrieve and set trace stream size attributes

29749 **SYNOPSIS**

29750 TRC
```
#include <sys/types.h>
```
29751
```
#include <trace.h>
```

29752 TRC TRL
```
int posix_trace_attr_getlogsize(const trace_attr_t *attr,
    size_t *logsize);
```
29754 TRC
```
int posix_trace_attr_getmaxdatasize(const trace_attr_t *attr,
    size_t *maxdatasize);
int posix_trace_attr_getmaxsystemeventsize(const trace_attr_t *attr,
    size_t *eventsize);
int posix_trace_attr_getmaxuserevent size(const trace_attr_t *attr,
    size_t data_len, size_t *eventsize);
int posix_trace_attr_getstreamsize(const trace_attr_t *attr,
    size_t *streamsize);
```
29762 TRC TRL
```
int posix_trace_attr_setlogsize(trace_attr_t *attr,
    size_t logsize);
```
29764 TRC
```
int posix_trace_attr_setmaxdatasize(trace_attr_t *attr,
    size_t maxdatasize);
int posix_trace_attr_setstreamsize(trace_attr_t *attr,
    size_t streamsize);
```
29768

29769 **DESCRIPTION**

29770 TRL The *posix_trace_attr_getlogsize*( ) function shall copy the log size, in bytes, from the *log-max-size* attribute of the attributes object pointed to by the *attr* argument into the variable pointed to by the *logsize* argument. This log size is the maximum total of bytes that shall be allocated for system and user trace events in the trace log. The default value for the *log-max-size* attribute is implementation-defined.

29775 The *posix_trace_attr_setlogsize*( ) function shall set the maximum allowed size, in bytes, in the *log-max-size* attribute of the attributes object pointed to by the *attr* argument, using the size value supplied by the *logsize* argument.

29778 The trace log size shall be used if the *log-full-policy* attribute is set to POSIX_TRACE_LOOP or POSIX_TRACE_UNTIL_FULL. If the *log-full-policy* attribute is set to POSIX_TRACE_APPEND, the implementation shall ignore the *log-max-size* attribute.

29781 The *posix_trace_attr_getmaxdatasize*( ) function shall copy the maximum user trace event data size, in bytes, from the *max-data-size* attribute of the attributes object pointed to by the *attr* argument into the variable pointed to by the *maxdatasize* argument. The default value for the *max-data-size* attribute is implementation-defined.

29785 The *posix_trace_attr_getmaxsystemeventsize*( ) function calculates the maximum memory size, in bytes, required to store a single system trace event. This value is calculated for the trace stream attributes object pointed to by the *attr* argument and is returned in the variable pointed to by the *eventsize* argument.

29789 The values returned as the maximum memory sizes of the user and system trace events shall be such that if the sum of the maximum memory sizes of a set of the trace events that may be recorded in a trace stream is less than or equal to the *stream-min-size* attribute of that trace

stream, the system provides the necessary resources for recording all those trace events, without loss.

The *posix_trace_attr_getmaxusereventsize*( ) function calculates the maximum memory size, in bytes, required to store a single user trace event generated by a call to *posix_trace_event*( ) with a *data_len* parameter equal to the *data_len* value specified in this call. This value is calculated for the trace stream attributes object pointed to by the *attr* argument and is returned in the variable pointed to by the *eventsize* argument.

The *posix_trace_attr_getstreamsize*( ) function shall copy the stream size, in bytes, from the *stream-min-size* attribute of the attributes object pointed to by the *attr* argument into the variable pointed to by the *streamsize* argument.

This stream size is the current total memory size reserved for system and user trace events in the trace stream. The default value for the *stream-min-size* attribute is implementation-defined. The stream size refers to memory used to store trace event records. Other stream data (for example, trace attribute values) shall not be included in this size.

The *posix_trace_attr_setmaxdatasize*( ) function shall set the maximum allowed size, in bytes, in the *max-data-size* attribute of the attributes object pointed to by the *attr* argument, using the size value supplied by the *maxdatasize* argument. This maximum size is the maximum allowed size for the user data argument which may be passed to *posix_trace_event*( ). The implementation shall be allowed to truncate data passed to *trace_user_event* which is longer than *maxdatasize*.

The *posix_trace_attr_setstreamsize*( ) function shall set the minimum allowed size, in bytes, in the *stream-min-size* attribute of the attributes object pointed to by the *attr* argument, using the size value supplied by the *streamsize* argument.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

TRL The *posix_trace_attr_getlogsize*( ) function stores the maximum trace log allowed size in the object pointed to by *logsize*, if successful.

The *posix_trace_attr_getmaxdatasize*( ) function stores the maximum trace event record memory size in the object pointed to by *maxdatasize*, if successful.

The *posix_trace_attr_getmaxsystemeventsize*( ) function stores the maximum memory size to store a single system trace event in the object pointed to by *eventsize*, if successful.

The *posix_trace_attr_getmaxusereventsize*( ) function stores the maximum memory size to store a single user trace event in the object pointed to by *eventsize*, if successful.

The *posix_trace_attr_getstreamsize*( ) function stores the maximum trace stream allowed size in the object pointed to by *streamsize*, if successful.

ERRORS

These functions may fail if:

[EINVAL] The value specified by one of the arguments is invalid.

29830 **EXAMPLES**

29831 None.

29832 **APPLICATION USAGE**

29833 None.

29834 **RATIONALE**

29835 None.

29836 **FUTURE DIRECTIONS**

29837 None.

29838 **SEE ALSO**

29839 *posix_trace_attr_init*( ), *posix_trace_create*( ), *posix_trace_event*( ), *posix_trace_get_attr*( ), the Base
29840 Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<trace.h>**

29841 **CHANGE HISTORY**

29842 First released in Issue 6. Derived from the IEEE Std. 1003.1q-2000.

29843 **NAME**

29844 posix_trace_attr_init — trace stream attributes object initialization

29845 **SYNOPSIS**

29846 TRC
```
#include <trace.h>
```

29847
```
int posix_trace_attr_init(trace_attr_t *attr);
```

29848

29849 **DESCRIPTION**

29850 Refer to *posix_trace_attr_destroy*( ).

29851 **NAME**

29852 posix_trace_clear — clear trace stream and trace log

29853 **SYNOPSIS**

29854 TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_clear(trace_id_t trid);
```
29855

29856

29857

29858 **DESCRIPTION**

29859 The *posix_trace_clear*( ) function shall reinitialize the trace stream identified by the argument *trid*
29860 as if it were returning from the *posix_trace_create*( ) function, except that the same allocated
29861 resources are reused, the mapping of trace event type identifiers to trace event names is
29862 unchanged, and the trace stream status remains unchanged (that is, if it was running, it remains
29863 running and if it was suspended, it remains suspended).

29864 All trace events in the trace stream recorded before the call to *posix_trace_clear*( ) are lost. The
29865 *posix_stream_full_status* status shall be set to POSIX_TRACE_NOT_FULL. There is no guarantee
29866 that all trace events that occurred during the *posix_trace_clear*( ) call are recorded; the behavior
29867 with respect to trace points that may occur during this call, is unspecified.

29868 TRL If the Trace Log option is supported and the trace stream has been created with a log, the
29869 *posix_trace_clear*( ) function shall reinitialize the trace stream with the same behavior as if the
29870 trace stream was created without the log, plus it shall reinitialize the trace log associated with
29871 the trace stream identified by the argument *trid* as if it were returning from the
29872 *posix_trace_create_withlog*( ) function, except that the same allocated resources, for the trace log,
29873 may be reused and the associated trace stream status remains unchanged. The first trace event
29874 recorded in the trace log after the call to *posix_trace_clear*( ) shall be the same as the first trace
29875 event recorded in the active trace stream after the call to *posix_trace_clear*( ). The
29876 *posix_log_full_status* status shall be set to POSIX_TRACE_NOT_FULL. There is no guarantee that
29877 all trace events that occurred during the *posix_trace_clear*( ) call are recorded in the trace log; the
29878 behavior with respect to trace points that may occur during this call is unspecified. If the log full
29879 policy is POSIX_TRACE_APPEND, the effect of a call to this function is unspecified for the trace
29880 log associated with the trace stream identified by the *trid* argument.

29881 **RETURN VALUE**

29882 Upon successful completion, the *posix_trace_clear*( ) function shall return a value of zero.
29883 Otherwise, it shall return the corresponding error number.

29884 **ERRORS**

29885 The *posix_trace_clear*( ) function may fail if:

29886 [EINVAL] The value of the *trid* argument does not correspond to an active trace stream.

29887 **EXAMPLES**

29888 None.

29889 **APPLICATION USAGE**

29890 None.

29891 **RATIONALE**

29892 None.

29893 **FUTURE DIRECTIONS**

29894 None.

29895 **SEE ALSO**

29896 *posix_trace_attr_init*( ), *posix_trace_create*( ), *posix_trace_flush*( ), *posix_trace_get_attr*( ), the Base
29897 Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<trace.h>**

29898 **CHANGE HISTORY**

29899 First released in Issue 6. Derived from IEEE Std. 1003.1q-2000.

29900 **NAME**

29901 posix_trace_close, posix_trace_open, posix_trace_rewind — trace log management

29902 **SYNOPSIS**

29903 TRC TRL
```
#include <trace.h>

int posix_trace_close(trace_id_t trid);
int posix_trace_open(int file_desc, trace_id_t *trid);
int posix_trace_rewind(trace_id_t trid);
```
29904
29905
29906
29907

29908 **DESCRIPTION**

29909 The *posix_trace_close*( ) function shall deallocate the trace log identifier indicated by *trid*, and all
29910 of its associated resources. If there is no valid trace log pointed to by the *trid*, this function shall
29911 fail.

29912 The *posix_trace_open*( ) function allocates the necessary resources and establishes the connection
29913 between a trace log identified by the *file_desc* argument and a trace stream identifier identified by
29914 the object pointed to by the *trid* argument. The *file_desc* argument should be a valid open file
29915 descriptor that corresponds to a trace log. The *file_desc* argument shall be open for reading. The
29916 current trace event timestamp, which specifies the timestamp of the trace event that will be read
29917 by the next call to *posix_trace_getnext_event*( ), shall be set to the timestamp of the oldest trace
29918 event recorded in the trace log identified by *trid*.

29919 The *posix_trace_open*( ) function returns a trace stream identifier in the variable pointed to by the
29920 *trid* argument, that may only be used by the following functions:

29921 *posix_trace_close*( )	*posix_trace_get_attr*( )	
29922 *posix_trace_eventid_equal*( )	*posix_trace_get_status*( )	
29923 *posix_trace_eventid_get_name*( )	*posix_trace_getnext_event*( )	
29924 *posix_trace_eventtypelist_getnext_id*( )	*posix_trace_rewind*( )	
29925 *posix_trace_eventtypelist_rewind*( )		

29926 In particular, notice that the operations normally used by a trace controller process, such as
29927 *posix_trace_start*( ), *posix_trace_stop*( ), or *posix_trace_shutdown*( ), cannot be invoked using the
29928 trace stream identifier returned by the *posix_trace_open*( ) function.

29929 The *posix_trace_rewind*( ) function shall reset the current trace event timestamp, which specifies
29930 the timestamp of the trace event that will be read by the next call to *posix_trace_getnext_event*( ),
29931 to the timestamp of the oldest trace event recorded in the trace log identified by *trid*.

29932 **RETURN VALUE**

29933 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
29934 return the corresponding error number.

29935 If successful, the *posix_trace_open*( ) function stores the trace stream identifier value in the object
29936 pointed to by *trid*.

29937 **ERRORS**

29938 The *posix_trace_open*( ) function may fail if:

29939 [EINTR] The operation was interrupted by a signal and thus no trace log was opened.

29940 [EINVAL] The object pointed to by *file_desc* does not correspond to a valid trace log.

29941 The *posix_trace_close*( ) and *posix_trace_rewind*( ) functions may fail if:

29942 [EINVAL] The object pointed to by *trid* does not correspond to a valid trace log.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_get_attr( ), *posix_trace_get_filter*( ), *posix_trace_getnext_event*( ), 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.

29956 **NAME**

29957 posix_trace_create, posix_trace_create_withlog, posix_trace_flush, posix_trace_shutdown —
29958 trace stream initialization, flush, and shutdown from a process

29959 **SYNOPSIS**

29960 TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_create(pid_t pid, const trace_attr_t *attr,
    trace_id_t *trid);
```
29964 TRC TRL
```
int posix_trace_create_withlog(pid_t pid, const trace_attr_t *attr,
    int file_desc, trace_id_t *trid);
int posix_trace_flush(trace_id_t trid);
int posix_trace_shutdown(trace_id_t trid);
```
29968

29969 **DESCRIPTION**

29970 The *posix_trace_create*( ) function creates an active trace stream. It allocates all the resources
29971 needed by the trace stream being created for tracing the process specified by *pid* in accordance
29972 with the *attr* argument. The *attr* argument represents the initial attributes of the trace stream and
29973 shall have been initialized by the function *posix_trace_attr_init*( ) prior the *posix_trace_create*( )
29974 call. If the argument *attr* is NULL, the default attributes shall be used. The *attr* attributes object
29975 shall be manipulated through a set of functions described in the *posix_trace_attr* family of
29976 functions. If the attributes of the object pointed to by *attr* are modified later, the attributes of the
29977 trace stream shall not be affected. The *creation-time* attribute of the newly created trace stream
29978 shall be set to the value of the system clock, if the Timers option is not supported, or to the value
29979 of the CLOCK_REALTIME clock, if the Timers option is supported.

29980 The *pid* argument represents the target process to be traced. If the process executing this
29981 function does not have appropriate privileges to trace the process identified by *pid*, an error shall
29982 be returned. If the *pid* argument is zero, the calling process shall be traced.

29983 The *posix_trace_create*( ) function stores the trace stream identifier of the new trace stream in the
29984 object pointed to by the *trid* argument. This trace stream identifier shall be used in subsequent
29985 calls to control tracing. The *trid* argument may only be used by the following functions:

29986 *posix_trace_clear*( )	*posix_trace_getnext_event*( )	
29987 *posix_trace_eventid_equal*( )	*posix_trace_shutdown*( )	
29988 *posix_trace_eventid_get_name*( )	*posix_trace_start*( )	
29989 *posix_trace_eventtypelist_getnext_id*( )	*posix_trace_stop*( )	
29990 *posix_trace_eventtypelist_rewind*( )	*posix_trace_timedgetnext_event*( )	
29991 *posix_trace_get_attr*( )	*posix_trace_trid_eventid_open*( )	
29992 *posix_trace_get_status*( )	*posix_trace_trygetnext_event*( )	

29993 TEF If the Trace Event Filter option is supported, the following additional functions may use the *trid*
29994 argument:

29995 *posix_trace_get_filter*( )	*posix_trace_set_filter*( )

29996

29997 In particular, notice that the operations normally used by a trace analyser process, such as
29998 *posix_trace_rewind*( ) or *posix_trace_close*( ), cannot be invoked using the trace stream identifier
29999 returned by the *posix_trace_create*( ) function.

30000 TEF A trace stream shall be created in a suspended state. If the Trace Event Filter option is
30001 supported, its trace event type filter shall be empty.

30002 The *posix_trace_create*( ) function may be called multiple times from the same or different
30003 processes, with the system-wide limit indicated by the runtime invariant value
30004 {TRACE_SYS_MAX}, which has the minimum value {_POSIX_TRACE_SYS_MAX}.

30005 The trace stream identifier returned by the *posix_trace_create*( ) function in the argument pointed
30006 to by *trid* is valid only in the process that made the function call. If it is used from another
30007 process, that is a child process, in functions defined in IEEE Std. 1003.1-200x, these functions
30008 shall return with the error [EINVAL].

30009 TRL The *posix_trace_create_withlog*( ) function creates a trace stream as in the *posix_trace_create*( )
30010 function and behaves the same way, plus it associates a trace log with this trace stream. The
30011 *file_desc* argument shall be the file descriptor designating the trace log destination. The function
30012 shall fail if this file descriptor refers to a file with a file type that is not compatible with the log
30013 policy associated with the trace log. The list of the appropriate file types that are compatible
30014 with each log policy shall be implementation-defined.

30015 The *posix_trace_create_withlog*( ) function returns in the parameter pointed to by *trid* the trace
30016 stream identifier, which uniquely identifies the newly created trace stream, and shall be used in
30017 subsequent calls to control tracing. The *trid* argument may only be used by the following
30018 functions:

30019 *posix_trace_clear*( )	*posix_trace_getnext_event*( )	
30020 *posix_trace_eventid_equal*( )	*posix_trace_shutdown*( )	
30021 *posix_trace_eventid_get_name*( )	*posix_trace_start*( )	
30022 *posix_trace_eventtypelist_getnext_id*( )	*posix_trace_stop*( )	
30023 *posix_trace_eventtypelist_rewind*( )	*posix_trace_timedgetnext_event*( )	
30024 *posix_trace_flush*( )	*posix_trace_trid_eventid_open*( )	
30025 *posix_trace_get_attr*( )	*posix_trace_trygetnext_event*( )	
30026 *posix_trace_get_status*( )		

30027

30028 TRL TEF If the Trace Event Filter option is supported, the following additional functions may use the *trid*
30029 argument:

30030 *posix_trace_get_filter*( )	*posix_trace_set_filter*( )

30031 In particular, notice that the operations normally used by a trace analyser process, such as
30032 *posix_trace_rewind*( ) or *posix_trace_close*( ), cannot be invoked using the trace stream identifier
30033 returned by the *posix_trace_create_withlog*( ) function.

30034 The *posix_trace_flush*( ) function initiates a flush operation which copies the contents of the trace
30035 stream identified by the argument *trid* into the trace log associated with the trace stream at the
30036 creation time. If no trace log has been associated with the trace stream pointed to by *trid*, this
30037 function shall return an error. The termination of the flush operation can be polled by the
30038 *posix_trace_get_status*( ) function. During the flush operation, it shall be possible to trace new
30039 trace events up to the point when the trace stream becomes full. After flushing is completed, the
30040 space used by the flushed trace events shall be available for tracing new trace events.

30041 If flushing the trace stream causes the resulting trace log to become full, the trace log full policy
30042 shall be applied. If the trace *log-full-policy* attribute is set, the following occurs:

POSIX_TRACE_UNTIL_FULL
The trace events that have not yet been flushed are discarded.

POSIX_TRACE_LOOP
The trace events that have not yet been flushed are written to the beginning of the trace log, overwriting previous trace events stored there.

POSIX_TRACE_APPEND
The trace events that had not yet been flushed shall be appended to the trace log.

For an active trace stream with log, when the *posix_trace_shutdown*( ) function is called, all trace events that have not yet been flushed to the trace log shall be flushed, as in the *posix_trace_flush*( ) function, and the trace log shall be closed.

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.

In addition, some unspecified information shall be written to the trace log to allow detection of a valid trace log during the *posix_trace_open*( ) operation.

The *posix_trace_shutdown*( ) function shall stop the tracing of trace events in the trace stream identified by *trid*, as if *posix_trace_stop*( ) had been invoked. The *posix_trace_shutdown*( ) function shall free all the resources associated with the trace stream.

The *posix_trace_shutdown*( ) function shall not return until all the resources associated with the trace stream have been freed. When the *posix_trace_shutdown*( ) function returns, the *trid* 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 *trace_getnext_event*( ) functions (which specified this *trid*) before this call is unblocked with the error [EINVAL].

If the process exits, invokes an *exec*( ) call, 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 *posix_trace_shutdown*( ) function.

The *posix_trace_shutdown*( ) function shall not return until all trace events have been flushed.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

TRL The *posix_trace_create*( ) and *posix_trace_create_withlog*( ) functions store the trace stream identifier value in the object pointed to by *trid*, if successful.

ERRORS

TRL The *posix_trace_create*( ) and *posix_trace_create_withlog*( ) functions shall fail if:

[EAGAIN] No more trace streams can be started now. {TRACE_SYS_MAX} has been exceeded.

[EINTR] The operation was interrupted by a signal. No trace stream was created.

[EINVAL] One or more of the trace parameters specified by the *attr* parameter is invalid.

[ENOMEM] The implementation does not currently have sufficient memory to create the trace stream with the specified parameters.

[EPERM] The caller does not have appropriate privilege to trace the process specified by *pid*.

[ESRCH] The *pid* argument does not refer to an existing process.

TRL The *posix_trace_create_withlog*( ) function shall fail if:

[EBADF] The *file_desc* argument is not a valid file descriptor open for writing.

[EINVAL] The *file_desc* argument refers to a file with a file type that does not support the log policy associated with the trace log.

[ENOSPC] No space left on device. The device corresponding to the argument *file_desc* does not contain the space required to create this trace log.

TRL The *posix_trace_flush*( ) and *posix_trace_shutdown*( ) functions shall fail if:

[EINVAL] The value of the *trid* argument does not correspond to an active trace stream with log.

[EFBIG] The trace log file has attempted to exceed an implementation-defined maximum file size.

[ENOSPC] No space left on device.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clock_getres( ), *exec*, *posix_trace_attr_init*( ), *posix_trace_clear*( ), *posix_trace_close*( ), *posix_trace_eventid_equal*( ), *posix_trace_eventtypelist_getnext_id*( ), *posix_trace_flush*( ), *posix_trace_get_attr*( ), *posix_trace_get_filter*( ), *posix_trace_get_status*( ), *posix_trace_getnext_event*( ), *posix_trace_open*( ), *posix_trace_rewind*( ), *posix_trace_set_filter*( ), *posix_trace_shutdown*( ), *posix_trace_start*( ), *posix_trace_timedgetnext_event*( ), *posix_trace_trid_eventid_open*( ), *posix_trace_start*( ), *time*( ), 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.

30120 **NAME**

30121 posix_trace_event, posix_trace_eventid_open — trace functions for instrumenting application
30122 code

30123 **SYNOPSIS**

30124 TRC
```
#include <sys/types.h>
#include <trace.h>

void posix_trace_event(trace_event_id_t event_id, const void *data_ptr,
    size_t data_len);
int posix_trace_eventid_open(const char *event_name,
    trace_event_id_t *event_id);
```
30130

30131 **DESCRIPTION**

30132 The *posix_trace_event*( ) function records the *event_id* and the user data pointed to by *data_ptr* in
30133 the trace stream into which the calling process is being traced and in which *event_id* is not
30134 filtered out. If the total size of the user trace event data represented by *data_len* is not greater
30135 than the declared maximum size for user trace event data, then the *truncation-status* attribute of
30136 the trace event recorded is POSIX_TRACE_NOT_TRUNCATED. Otherwise, the user trace event
30137 data is truncated to this declared maximum size and the *truncation-status* attribute of the trace
30138 event recorded is POSIX_TRACE_TRUNCATED_RECORD.

30139 If there is no trace stream created for the process or if the created trace stream is not running or if
30140 the trace event specified by *event_id* is filtered out in the trace stream, the *posix_trace_event*( )
30141 function has no effect.

30142 The *posix_trace_eventid_open*( ) function is used to associate a user trace event name with a trace
30143 event type identifier for the calling process. The trace event name is the string pointed to by the
30144 argument *event_name*. It shall have a maximum of {TRACE_EVENT_NAME_MAX} characters
30145 (which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). The number of user
30146 trace event type identifiers that can be defined for any given process is limited by the maximum
30147 value {TRACE_USER_EVENT_MAX}, which has the minimum value
30148 {POSIX_TRACE_USER_EVENT_MAX}.

30149 If the Trace Inheritance option is not supported, the *posix_trace_trid_eventid_open*( ) function shall
30150 associate the user trace event name pointed to by the *event_name* argument with a trace event
30151 type identifier that is unique for the traced process, and is returned in the variable pointed to by
30152 the *event* argument. If the user trace event name has already been mapped for the traced process,
30153 then the previously assigned trace event type identifier shall be returned. If the per-process user
30154 trace event name limit represented by {TRACE_USER_EVENT_MAX} has been reached, the
30155 pre-defined POSIX_TRACE_UNNAMED_USEREVENT (see Table 2-10 (on page 582)) user trace
30156 event shall be returned.

30157 TRI If the Trace Inherit option is supported, the *posix_trace_trid_eventid_open*( ) function shall
30158 associate the user trace event name pointed to by the *event_name* argument with a trace event
30159 type identifier that is unique for all the processes being traced in this same trace stream, and is
30160 returned in the variable pointed to by the *event* argument. If the user trace event name has
30161 already been mapped for the traced processes, then the previously assigned trace event type
30162 identifier shall be returned. If the per-process user trace event name limit represented by
30163 {TRACE_USER_EVENT_MAX} has been reached, the pre-defined
30164 POSIX_TRACE_UNNAMED_USEREVENT (Table 2-10 (on page 582)) user trace event shall be
30165 returned.

30166 **Note:** The above procedure, together with the fact that multiple processes can only be
30167 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.

If there is no trace stream created, the *posix_trace_eventid_open*( ) function shall store this information for future trace streams created for this process.

RETURN VALUE

No return value is defined for the *posix_trace_event*( ) function.

Upon successful completion, the *posix_trace_eventid_open*( ) function shall return a value of zero. Otherwise, it shall return the corresponding error number. The *posix_trace_eventid_open*( ) function stores the trace event type identifier value in the object pointed to by *event_id*, if successful.

ERRORS

The *posix_trace_eventid_open*( ) function may fail if:

[ENAMETOOLONG]
The size of the name pointed to by *event_name* argument was longer than the implementation-defined value {TRACE_EVENT_NAME_MAX}.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_start( ), *posix_trace_trid_eventid_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.

30197 **NAME**

30198 posix_trace_eventid_equal, posix_trace_eventid_get_name, posix_trace_trid_eventid_open —
30199 manipulate trace event type identifier

30200 **SYNOPSIS**

30201 TRC
```
#include <trace.h>

int posix_trace_eventid_equal(trace_id_t trid, trace_eventid_t event1,
    trace_eventid_t event2);
int posix_trace_eventid_get_name(trace_id_t trid, trace_eventid_t event,
    char *event_name);
```
30206 TRC TEF
```
int posix_trace_trid_eventid_open(trace_id_t trid,
    const char *event_name, trace_eventid_t *event);
```
30208

30209 **DESCRIPTION**

30210 The *posix_trace_eventid_equal*( ) function compares the trace event type identifiers *event1* and
30211 *event2* from the same trace stream or the same trace log identified by the *trid* argument. If the
30212 trace event type identifiers *event1* and *event2* are from different trace streams, the return value
30213 shall be unspecified.

30214 The *posix_trace_eventid_get_name*( ) function returns in the argument pointed to by *event_name*,
30215 the trace event name associated with the trace event type identifier identified by the argument
30216 *event*, for the trace stream or for the trace log identified by the *trid* argument. The name of the
30217 trace event shall have a maximum of {TRACE_EVENT_NAME_MAX} characters (which has the
30218 minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). Successive calls to this function
30219 with the same trace event type identifier and the same trace stream identifier shall return the
30220 same event name.

30221 TEF The *posix_trace_trid_eventid_open*( ) function is used to associate a user trace event name with a
30222 trace event type identifier for a given trace stream. The trace stream is identified by the *trid*
30223 argument, and it shall be an active trace stream. The trace event name is the string pointed to by
30224 the argument *event_name*. It shall have a maximum of {TRACE_EVENT_NAME_MAX}
30225 characters (which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). The
30226 number of user trace event type identifiers that can be defined for any given process is limited
30227 by the maximum value {TRACE_USER_EVENT_MAX}, which has the minimum value
30228 {_POSIX_TRACE_USER_EVENT_MAX}.

30229 If the Trace Inheritance option is not supported, the *posix_trace_trid_eventid_open*( ) function shall
30230 associate the user trace event name pointed to by the *event_name* argument with a trace event
30231 type identifier that is unique for the process being traced in the trace stream identified by the *trid*
30232 argument, and is returned in the variable pointed to by the *event* argument. If the user trace
30233 event name has already been mapped for the traced process, then the previously assigned trace
30234 event type identifier shall be returned. If the per-process user trace event name limit represented
30235 by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined
30236 POSIX_TRACE_UNNAMED_USEREVENT (see Table 2-10 (on page 582)) user trace event shall
30237 be returned.

30238 TEF TRI If the Trace Inheritance option is supported, the *posix_trace_trid_eventid_open*( ) function shall
30239 associate the user trace event name pointed to by the *event_name* argument with a trace event
30240 type identifier that is unique for all the processes being traced in the trace stream identified by
30241 the *trid* argument, and is returned in the variable pointed to by the *event* argument. If the user
30242 trace event name has already been mapped for the traced processes, then the previously
30243 assigned trace event type identifier shall be returned. If the per-process user trace event name
30244 limit represented by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined
30245 POSIX_TRACE_UNNAMED_USEREVENT (see Table 2-10 (on page 582)) user trace event shall

be returned.

RETURN VALUE

TEF Upon successful completion, the *posix_trace_eventid_get_name*( ) and *posix_trace_trid_eventid_open*( ) functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

The *posix_trace_eventid_equal*( ) function shall return a non-zero value if *event1* and *event2* are equal; otherwise, a value of zero shall be returned. No errors are defined. If either *event1* or *event2* are not valid trace event type identifiers for the trace stream specified by *trid* or if the *trid* is invalid, the behavior shall be unspecified.

The *posix_trace_eventid_get_name*( ) function stores the trace event name value in the object pointed to by *event_name*, if successful.

TEF The *posix_trace_trid_eventid_open*( ) function stores the trace event type identifier value in the object pointed to by *event*, if successful.

ERRORS

TEF The *posix_trace_eventid_get_name*( ) and *posix_trace_trid_eventid_open*( ) functions may fail if:

[EINVAL] The *trid* argument was not a valid trace type identifier.

TEF The *posix_trace_trid_eventid_open*( ) function may fail if:

[ENAMETOOLONG]
The size of the name pointed to by *event_name* argument was longer than the implementation-defined value {TRACE_EVENT_NAME_MAX}.

The *posix_trace_eventid_get_name*( ) function may fail if:

[EINVAL] The trace event type identifier *event* was not associated with any name.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_event( ), <REFERENCE UNDEFINED>(posix_trace_eventid), *posix_trace_getnext_event*( ), 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.

30282 **NAME**

30283 posix_trace_eventset_add, posix_trace_eventset_del, posix_trace_eventset_empty,
30284 posix_trace_eventset_fill, posix_trace_eventset_ismember — manipulate trace event type sets

30285 **SYNOPSIS**

30286 TRC TEF
```
#include <trace.h>

int posix_trace_eventset_add(trace_event_id_t event_id,
    trace_event_set_t *set);
int posix_trace_eventset_del(trace_event_id_t event_id,
    trace_event_set_t *set);
int posix_trace_eventset_empty(trace_event_set_t *set);
int posix_trace_eventset_fill(trace_event_set_t *set, int what);
int posix_trace_eventset_ismember(trace_event_id_t event_id,
    const trace_event_set_t *set, int *ismember);
```
30295

30296 **DESCRIPTION**

30297 These primitives manipulate sets of trace event types. They operate on data objects addressable
30298 by the application, not on the current trace event filter of any trace stream.

30299 The *posix_trace_eventset_add*( ) and *posix_trace_eventset_del*( ) functions, respectively, add or
30300 delete the individual trace event type specified by the value of the argument *event_id* to or from
30301 the trace event type set pointed to by the argument *set*. Adding a trace event type already in the
30302 set or deleting a trace event type not in the set shall not be considered an error.

30303 The *posix_trace_eventset_empty*( ) function initializes the trace event type set pointed to by the *set*
30304 argument such that all trace event types defined, both system and user, shall be excluded from
30305 the set.

30306 The *posix_trace_eventset_fill*( ) function initializes the trace event type set pointed to by the
30307 argument *set*, such that the set of trace event types defined by the argument *what* shall be
30308 included in the set. The value of the argument *what* shall consist of one of the following values,
30309 as defined in the **<trace.h>** header:

30310 POSIX_TRACE_WOPID_EVENTS
30311 All the process-independent implementation-defined system trace event types are included
30312 in the set.

30313 POSIX_TRACE_SYSTEM_EVENTS All the implementation-defined system trace event types are
30314 included in the set, as are those defined in IEEE Std. 1003.1-200x.

30315 POSIX_TRACE_ALL_EVENTS All trace event types defined, both system and user, are included
30316 in the set.

30317 Applications shall call either *posix_trace_eventset_empty*( ) or *posix_trace_eventset_fill*( ) at least
30318 once for each object of type **trace_event_set_t** prior to any other use of that object. If such an
30319 object is not initialized in this way, but is nonetheless supplied as an argument to any of the
30320 *posix_trace_eventset_add*( ), *posix_trace_eventset_del*( ), or *posix_trace_eventset_ismember*( ) functions,
30321 the results are undefined.

30322 The *posix_trace_eventset_ismember*( ) function tests whether the trace event type specified by the
30323 value of the argument *event_id* is a member of the set pointed to by the argument *set*. The value
30324 returned in the object pointed to by *ismember* argument is zero if the trace event type identifier is
30325 not a member of the set and and a value different from zero if it is a member of the set.

30326 **RETURN VALUE**

30327 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
30328 return the corresponding error number.

30329 **ERRORS**

30330 These functions may fail if:

30331 [EINVAL] The value of one of the arguments is invalid.

30332 **EXAMPLES**

30333 None.

30334 **APPLICATION USAGE**

30335 None.

30336 **RATIONALE**

30337 None.

30338 **FUTURE DIRECTIONS**

30339 None.

30340 **SEE ALSO**

30341 *posix_trace_set_filter*( ), *posix_trace_trid_eventid_open*( ), the Base Definitions volume of
30342 IEEE Std. 1003.1-200x, **<trace.h>**

30343 **CHANGE HISTORY**

30344 First released in Issue 6. Derived from IEEE Std. 1003.1q-2000.

30345 **NAME**

30346 posix_trace_eventtypelist_getnext_id, posix_trace_eventtypelist_rewind — iterate over a
30347 mapping of trace event types

30348 **SYNOPSIS**

30349 TRC
```
#include <trace.h>
```

30350
```
int posix_trace_eventtypelist_getnext_id(trace_id_t trid,
    trace_eventid_t *event, int *unavailable);
int posix_trace_eventtypelist_rewind(trace_id_t trid);
```
30353

30354 **DESCRIPTION**

30355 The first time *posix_trace_eventtypelist_getnext_id*( ) is called, the function shall return in the
30356 variable pointed to by *event* the first trace event type identifier of the list of trace events of the
30357 trace stream identified by the *trid* argument. Successive calls to
30358 *posix_trace_eventtypelist_getnext_id*( ) return in the variable pointed to by *event* the next trace
30359 event type identifier in that same list. Each time a trace event type identifier is successfully
30360 written into the variable pointed to by the *event* argument, the variable pointed to by the
30361 *unavailable* argument shall be set to zero. When no more trace event type identifiers are
30362 available, and so none is returned, the variable pointed to by the *unavailable* argument shall be
30363 set to a value different from zero.

30364 The *posix_trace_eventtypelist_rewind*( ) function shall reset the next trace event type identifier to
30365 be read to the first trace event type identifier from the list of trace events used in the trace stream
30366 identified by *trid*.

30367 **RETURN VALUE**

30368 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
30369 return the corresponding error number.

30370 The *posix_trace_eventtypelist_getnext_id*( ) function stores the trace event type identifier value in
30371 the object pointed to by *event*, if successful.

30372 **ERRORS**

30373 These functions may fail if:

30374 [EINVAL] The *trid* argument was not a valid trace stream identifier.

30375 **EXAMPLES**

30376 None.

30377 **APPLICATION USAGE**

30378 None.

30379 **RATIONALE**

30380 None.

30381 **FUTURE DIRECTIONS**

30382 None.

30383 **SEE ALSO**

30384 *posix_trace_event*( ), <REFERENCE UNDEFINED>(posix_trace_eventid),
30385 *posix_trace_getnext_event*( ), *posix_trace_trid_eventid_open*( ), the Base Definitions volume of
30386 IEEE Std. 1003.1-200x, **<trace.h>**

30387 **CHANGE HISTORY**
30388 First released in Issue 6. Derived from IEEE Std. 1003.1q-2000.

30389 **NAME**

30390 posix_trace_flush — trace stream flush from a process

30391 **SYNOPSIS**

30392 TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_flush(trace_id_t trid);
```
30395

30396 **DESCRIPTION**

30397 Refer to *posix_trace_create*( ).

30398 **NAME**

30399 posix_trace_get_attr, posix_trace_get_status — retrieve the trace attributes or trace statuses

30400 **SYNOPSIS**

30401 TRC
```
#include <trace.h>

int posix_trace_get_attr(trace_id_t trid, trace_attr_t *attr);
int posix_trace_get_status(trace_id_t trid,
    struct posix_trace_status_info *statusinfo);
```

30405

30406 **DESCRIPTION**

30407 The *posix_trace_get_attr*( ) function shall copy the attributes of the active trace stream identified
30408 TRL by *trid* into the object pointed to by the *attr* argument. If the Trace Log option is supported, *trid*
30409 may represent a pre-recorded trace log.

30410 The *posix_trace_get_status*( ) function returns, in the structure pointed to by the *statusinfo*
30411 argument, the current trace status for the trace stream identified by the *trid* argument. These
30412 status values returned in the structure pointed to by *statusinfo* shall have been appropriately
30413 TRL read to ensure that the returned values are consistent. If the Trace Log option is supported and
30414 the *trid* argument refers to a pre-recorded trace stream, the status shall be the status of the
30415 completed trace stream.

30416 Each time the *posix_trace_get_status*( ) function is used, the overrun status of the trace stream
30417 TRL shall be reset to POSIX_TRACE_NO_OVERRUN immediately after the call completes. If the
30418 Trace Log option is supported, the *posix_trace_get_status*( ) function shall behave the same as
30419 when the option is not supported except for the following differences:

30420 - If the *trid* argument refers to a trace stream with log, each time the *posix_trace_get_status*( )
30421 function is used, the log overrun status of the trace stream shall be reset to
30422 POSIX_TRACE_NO_OVERRUN and the *flush_error* status shall be reset to zero immediately
30423 after the call completes.

30424 - If the *trid* argument refers to a pre-recorded trace stream, the status returned shall be the
30425 status of the completed trace stream and the status values of the trace stream shall not be
30426 reset.

30427

30428 **RETURN VALUE**

30429 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
30430 return the corresponding error number.

30431 The *posix_trace_get_attr*( ) function stores the trace attributes in the object pointed to by *attr*, if
30432 successful.

30433 The *posix_trace_get_status*( ) function stores the trace status in the object pointed to by *statusinfo*,
30434 if successful.

30435 **ERRORS**

30436 These functions may fail if:

30437 [EINVAL] The trace stream argument *trid* does not correspond to a valid active trace
30438 stream or a valid trace log.

30439 **EXAMPLES**

30440 None.

30441 **APPLICATION USAGE**

30442 None.

30443 **RATIONALE**

30444 None.

30445 **FUTURE DIRECTIONS**

30446 None.

30447 **SEE ALSO**

30448 *posix_trace_attr_destroy*( ), *posix_trace_attr_init*( ), *posix_trace_create*( ), *posix_trace_open*( ), the Base
30449 Definitions volume of IEEE Std. 1003.1-200x, **<trace.h>**

30450 **CHANGE HISTORY**

30451 First released in Issue 6. Derived from IEEE Std. 1003.1q-2000.

30452 **NAME**

30453 posix_trace_get_filter, posix_trace_set_filter — retrieve and set filter of an initialized trace
30454 stream

30455 **SYNOPSIS**

30456 TRC TEF
```
#include <trace.h>

int posix_trace_get_filter(trace_id_t trid, trace_event_set_t *set);
int posix_trace_set_filter(trace_id_t trid,
    const trace_event_set_t *set, int how);
```

30460

30461 **DESCRIPTION**

30462 The *posix_trace_get_filter*( ) function shall be used to retrieve, into the argument pointed to by *set*,
30463 the actual trace event filter from the trace stream specified by *trid*.

30464 The *posix_trace_set_filter*( ) function shall be used to change the set of filtered trace event types
30465 after a trace stream identified by the *trid* argument is created. This function may be called prior
30466 to starting the trace stream, or while the trace stream is active. By default, if no call is made to
30467 *posix_trace_set_filter*( ), all trace events shall be recorded (that is, none of the trace event types are
30468 filtered out).

30469 If this function is called while the trace is in progress, a special system trace event,
30470 POSIX_TRACE_FILTER, shall be recorded in the trace indicating both the old and the new sets
30471 of filtered trace event types (see Table 2-7 (on page 580) and Table 2-9 (on page 581)).

30472 If the *posix_trace_set_filter*( ) function is interrupted by a signal, an error is returned and the filter
30473 is not changed. In this case, the state of the trace stream shall not be changed.

30474 The value of the argument *how* indicates the manner in which the set is to be changed and shall
30475 have one of the following values, as defined in the **<trace.h>** header:

30476 POSIX_TRACE_SET_EVENTSET
30477 The resulting set of trace event types to be filtered shall be the trace event type set pointed
30478 to by the argument *set*.

30479 POSIX_TRACE_ADD_EVENTSET
30480 The resulting set of trace event types to be filtered shall be the union of the current set and
30481 the trace event type set pointed to by the argument *set*.

30482 POSIX_TRACE_SUB_EVENTSET
30483 The resulting set of trace event types to be filtered shall be all trace event types in the
30484 current set that are not in the set pointed to by the argument *set*; that is, remove each
30485 element of the specified set from the current filter.

30486 **RETURN VALUE**

30487 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
30488 return the corresponding error number.

30489 The *posix_trace_get_filter*( ) function stores the set of filtered trace event types in *set*, if successful.

30490 **ERRORS**

30491 These functions may fail if:

30492 [EINVAL] The value of the *trid* argument does not correspond to an active trace stream
30493 or the value of the argument pointed to by *set* is invalid.

30494 [EINTR] The operation was interrupted by a signal.

30495 **EXAMPLES**

30496 None.

30497 **APPLICATION USAGE**

30498 None.

30499 **RATIONALE**

30500 None.

30501 **FUTURE DIRECTIONS**

30502 None.

30503 **SEE ALSO**

30504 *posix_trace_eventset_add*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<trace.h>**

30505 **CHANGE HISTORY**

30506 First released in Issue 6. Derived from IEEE Std. 1003.1q-2000.

30507 **NAME**

30508 posix_trace_get_status — retrieve the trace statuses

30509 **SYNOPSIS**

30510 TRC
```
#include <trace.h>

int posix_trace_get_status(trace_id_t trid,
    struct posix_trace_status_info *statusinfo);
```
30511

30512

30513

30514 **DESCRIPTION**

30515 Refer to *posix_trace_get_attr*( ).

30516 **NAME**

30517 posix_trace_getnext_event, posix_trace_timedgetnext_event, posix_trace_trygetnext_event —
30518 retrieve a trace event

30519 **SYNOPSIS**

30520 TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_getnext_event(trace_id_t trid,
    struct posix_trace_event_info *event, void *data,
    size_t num_bytes, size_t *data_len, int *unavailable);
```
30525 TRC TMO
```
int posix_trace_timedgetnext_event(trace_id_t trid,
    struct posix_trace_event_info *event, void *data,
    size_t num_bytes, size_t *data_len, int *unavailable,
    const struct timespec *abs_timeout);
int posix_trace_trygetnext_event(trace_id_t trid,
    struct posix_trace_event_info *event, void *data,
    size_t num_bytes, size_t *data_len, int *unavailable);
```
30532

30533 **DESCRIPTION**

30534 The *posix_trace_getnext_event*( ) function shall be used to report a recorded trace event either
30535 TRL from an active trace stream without log or a pre-recorded trace stream identified by the *trid*
30536 argument. The *posix_trace_trygetnext_event*( ) function shall be used to report a recorded trace
30537 event from an active trace stream without log identified by the *trid* argument.

30538 The trace event information associated with the recorded trace event shall be copied by the
30539 function into the structure pointed to by the argument *event* and the data associated with the
30540 trace event shall be copied into the buffer pointed to by the *data* argument.

30541 The *posix_trace_getnext_event*( ) function shall block if the *trid* argument identifies an active trace
30542 stream and there is currently no trace event ready to be retrieved. When returning, if a recorded
30543 trace event was reported, the variable pointed to by the *unavailable* argument shall be set to zero.
30544 Otherwise, the variable pointed to by the *unavailable* argument shall be set to a value different
30545 from zero.

30546 TMO The *posix_trace_timedgetnext_event*( ) function attempts to get another trace event from an active
30547 trace stream without log, as in the *posix_trace_getnext_event*( ) function. However, if no trace
30548 event is available from the trace stream, this wait shall be terminated when the timeout specified
30549 by the argument *abs_timeout* expires, and the function shall return the error [ETIMEDOUT].

30550 The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the
30551 clock upon which timeouts are based (that is, when the value of that clock equals or exceeds
30552 *abs_timeout*), or if the absolute time specified by *abs_timeout* has already passed at the time of the
30553 call.

30554 TMO TMR If the Timers option is supported, the timeout is based on the CLOCK_REALTIME clock; if the
30555 Timers option is not supported, the timeout is based on the system clock as returned by the
30556 *time*( ) function. The resolution of the timeout is the resolution of the clock on which it is based.
30557 The **timespec** data type is defined as a structure in the header **<time.h>**.

30558 Under no circumstance will the function fail with a timeout if a trace event is immediately
30559 available from the trace stream. The validity of the *abs_timeout* argument need not be checked if
30560 a trace event is immediately available from the trace stream.

30561 TMO TMR The behavior of this function for a pre-recorded trace stream is unspecified.

30562 TRL The *posix_trace_trygetnext_event*( ) function shall not block. This function shall return an error if
30563 the *trid* argument identifies a pre-recorded trace stream. If a recorded trace event was reported,
30564 the variable pointed to by the *unavailable* argument shall be set to zero. Otherwise, if no trace
30565 event was reported, the variable pointed to by the *unavailable* argument shall be set to a value
30566 different from zero.

30567 The argument *num_bytes* shall be the size of the buffer pointed to by the *data* argument. The
30568 argument *data_len* reports to the application the length in bytes of the data record just
30569 transferred. If *num_bytes* is greater than or equal to the size of the data associated with the trace
30570 event pointed to by the *event* argument, all the recorded data shall be transferred. In this case, the
30571 *truncation-status* member of the trace event structure shall be either
30572 POSIX_TRACE_NOT_TRUNCATED, if the trace event data was recorded without truncation
30573 while tracing, or POSIX_TRACE_TRUNCATED_RECORD, if the trace event data was truncated
30574 when it was recorded. If the *num_bytes* argument is less than the length of recorded trace event
30575 data, the data transferred shall be truncated to a length of *num_bytes*, the value stored in the
30576 variable pointed to by *data_len* shall be equal to *num_bytes*, and the *truncation-status* member of
30577 the *event* structure argument shall be set to POSIX_TRACE_TRUNCATED_READ (see the
30578 *posix_trace_event_info*( ) function).

30579 The report of a trace event shall be sequential starting from the oldest recorded trace event. Trace
30580 events shall be reported in the order in which they were generated, up to an implementation-
30581 defined time resolution that causes the ordering of trace events occurring very close to each
30582 other to be unknown. Once reported, a trace event cannot be reported again from an active trace
30583 stream. Once a trace event is reported from an active trace stream without log, the trace stream
30584 shall make the resources associated with that trace event available to record future generated
30585 trace events.

30586 **RETURN VALUE**

30587 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
30588 return the corresponding error number.

30589 If successful, these functions store:

30590 - The recorded trace event in the object pointed to by *event*
30591 - The trace event information associated with the recorded trace event in the object pointed to
30592 by *data*
30593 - The length of this trace event information in the object pointed to by *data_len*
30594 - The value of zero in the object pointed to by *unavailable*

30595 **ERRORS**

30596 These functions may fail if:

30597 [EINVAL] The trace stream identifier argument *trid* is invalid.

30598 The *posix_trace_getnext_event*( ) and *posix_trace_timedgetnext_event*( ) functions may fail if:

30599 [EINTR] The operation was interrupted by a signal, and so the call had no effect.

30600 The *posix_trace_trygetnext_event*( ) function may fail if:

30601 [EINVAL] The trace stream identifier argument *trid* does not correspond to an active
30602 trace stream.

30603 TMO The *posix_trace_timedgetnext_event*( ) function may fail if:

30604 [EINVAL] There is no trace event immediately available from the trace stream, and the
30605 *timeout* argument is invalid.

30606 [ETIMEDOUT] No trace event was available from the trace stream before the specified
30607 timeout *timeout* expired.
30608

30609 **EXAMPLES**

30610 None.

30611 **APPLICATION USAGE**

30612 None.

30613 **RATIONALE**

30614 None.

30615 **FUTURE DIRECTIONS**

30616 None.

30617 **SEE ALSO**

30618 *posix_trace_create*( ), **posix_trace_event_info Structure**, *posix_trace_open*( ), the Base Definitions
30619 volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<trace.h>**

30620 **CHANGE HISTORY**

30621 First released in Issue 6. Derived from IEEE Std. 1003.1q-2000.

30622 **NAME**

30623 posix_trace_open — trace log management

30624 **SYNOPSIS**

30625 TCT TRL
```
#include <trace.h>

int posix_trace_open(int file_desc, trace_id_t *trid);
```
30626

30627

30628 **DESCRIPTION**

30629 Refer to *posix_trace_close*( ).

30630 **NAME**

30631 posix_trace_rewind — trace log management

30632 **SYNOPSIS**

30633 TCT TRL
```
#include <trace.h>

int posix_trace_rewind(trace_id_t trid);
```
30634

30635

30636 **DESCRIPTION**

30637 Refer to *posix_trace_close*( ).

30638 **NAME**

30639 posix_trace_set_filter — set filter of an initialized trace stream

30640 **SYNOPSIS**

30641 TRC TEF
```
#include <trace.h>

int posix_trace_set_filter(trace_id_t trid,
    const trace_event_set_t *set, int how);
```
30642

30643

30644

30645 **DESCRIPTION**

30646 Refer to *posix_trace_get_filter*( ).

30647 **NAME**

30648 posix_trace_shutdown — trace stream shutdown from a process

30649 **SYNOPSIS**

30650 TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_shutdown(trace_id_t trid);
```
30653

30654 **DESCRIPTION**

30655 Refer to *posix_trace_create*( ).

30656 **NAME**

30657 posix_trace_start, posix_trace_stop — trace start and stop

30658 **SYNOPSIS**

30659 TRC
```
#include <trace.h>

int posix_trace_start(trace_id_t trid);
int posix_trace_stop (trace_id_t trid);
```
30662

30663 **DESCRIPTION**

30664 The *posix_trace_start*( ) and *posix_trace_stop*( ) functions, respectively, start and stop the trace
30665 stream identified by the argument *trid*.

30666 The effect of calling the *posix_trace_start*( ) function shall be recorded in the trace stream as the
30667 POSIX_TRACE_START system trace event and the status of the trace stream shall become
30668 POSIX_TRACE_RUNNING. If the trace stream is in progress when this function is called, the
30669 POSIX_TRACE_START system trace event shall not be recorded and the trace stream shall
30670 continue to run. If the trace stream is full, the POSIX_TRACE_START system trace event shall
30671 not be recorded and the status of the trace stream shall not be changed.

30672 The effect of calling the *posix_trace_stop*( ) function shall be recorded in the trace stream as the
30673 POSIX_TRACE_STOP system trace event and the status of the trace stream shall become
30674 POSIX_TRACE_SUSPENDED. If the trace stream is suspended when this function is called, the
30675 POSIX_TRACE_STOP system trace event shall not be recorded and the trace stream shall remain
30676 suspended. If the trace stream is full, the POSIX_TRACE_STOP system trace event shall not be
30677 recorded and the status of the trace stream shall not be changed.

30678 **RETURN VALUE**

30679 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
30680 return the corresponding error number.

30681 **ERRORS**

30682 These functions may fail if:

30683 [EINVAL] The value of the argument *trid* does not correspond to an active trace stream
30684 and thus no trace stream was started or stopped.

30685 [EINTR] The operation was interrupted by a signal and thus the trace stream was not
30686 necessarily started or stopped.

30687 **EXAMPLES**

30688 None.

30689 **APPLICATION USAGE**

30690 None.

30691 **RATIONALE**

30692 None.

30693 **FUTURE DIRECTIONS**

30694 None.

30695 **SEE ALSO**

30696 *posix_trace_create*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<trace.h>**

30697 **CHANGE HISTORY**
30698 First released in Issue 6. Derived from IEEE Std. 1003.1q-2000.

30699 **NAME**

30700 posix_trace_timedgetnext_event, — retrieve a trace event

30701 **SYNOPSIS**

30702 TRC TMO
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_timedgetnext_event(trace_id_t trid,
    struct posix_trace_event_info *event, void *data,
    size_t num_bytes, size_t *data_len, int *unavailable,
    const struct timespec *abs_timeout);
```
30708

30709 **DESCRIPTION**

30710 Refer to *posix_trace_getnext_event*( ).

30711 **NAME**

30712 posix_trace_trid_eventid_open — manipulate trace event type identifier

30713 **SYNOPSIS**

30714 TRC TEF
```
#include <trace.h>
```

30715
```
int posix_trace_trid_eventid_open(trace_id_t trid,
    const char *event_name, trace_eventid_t *event);
```

30716

30717

30718 **DESCRIPTION**

30719 Refer to *posix_trace_eventid_equal*( ).

30720 **NAME**

30721 posix_trace_trygetnext_event — retrieve a trace event

30722 **SYNOPSIS**

30723 TRC TMO
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_trygetnext_event(trace_id_t trid,
    struct posix_trace_event_info *event, void *data,
    size_t num_bytes, size_t *data_len, int *unavailable);
```
30728

30729 **DESCRIPTION**

30730 Refer to *posix_trace_getnext_event*( ).

30731 **NAME**

30732 posix_typed_mem_get_info — query typed memory information

30733 **SYNOPSIS**

30734 TYM
```
#include <sys/mman.h>

int posix_typed_mem_get_info(int fildes,
    struct posix_typed_mem_info *info);
```
30735

30736

30737

30738 **DESCRIPTION**

30739 The *posix_typed_mem_get_info*( ) function returns, in the *posix_tmi_length* field of the
30740 **posix_typed_mem_info** structure pointed to by *info*, the maximum length which may be
30741 successfully allocated by the typed memory object designated by *fildes*. This maximum length
30742 shall take into account the flag POSIX_TYPED_MEM_ALLOCATE or
30743 POSIX_TYPED_MEM_ALLOCATE_CONTIG specified when the typed memory object
30744 represented by *fildes* was opened. The maximum length is dynamic; therefore, the value returned
30745 is valid only while the current mapping of the corresponding typed memory pool remains
30746 unchanged.

30747 If *fildes* represents a typed memory object opened with neither the
30748 POSIX_TYPED_MEM_ALLOCATE flag nor the POSIX_TYPED_MEM_ALLOCATE_CONTIG |
30749 flag specified, the returned value of *info->posix_tmi_length* is unspecified. |

30750 The *posix_typed_mem_get_info*( ) function may return additional implementation-defined |
30751 information in other fields of the **posix_typed_mem_info** structure pointed to by *info*. |

30752 If the memory object specified by *fildes* is not a typed memory object, then the behavior of this
30753 function is undefined.

30754 **RETURN VALUE**

30755 Upon successful completion, the *posix_typed_mem_get_info*( ) function shall return zero;
30756 otherwise, the corresponding error status value shall be returned.

30757 **ERRORS**

30758 The *posix_typed_mem_get_info*( ) function shall fail if:

30759 [EBADF] The *fildes* argument is not a valid open file descriptor.

30760 [ENODEV] The *fildes* argument is not connected to a memory object supported by this
30761 function.

30762 This function shall not return an error code of [EINTR]. |

30763 **EXAMPLES**

30764 None.

30765 **APPLICATION USAGE**

30766 None.

30767 **RATIONALE**

30768 An application that needs to allocate a block of typed memory with length dependent upon the
30769 amount of memory currently available must either query the typed memory object to obtain the
30770 amount available, or repeatedly invoke *mmap*( ) attempting to guess an appropriate length.
30771 While the latter method is existing practice with *malloc*( ), it is awkward and imprecise. The
30772 *posix_typed_mem_get_info*( ) function allows an application to immediately determine available
30773 memory. This is particularly important for typed memory objects that may in some cases be
30774 scarce resources. Note that when a typed memory pool is a shared resource, some form of
30775 mutual exclusion or synchronization may be required while typed memory is being queried and

30776 allocated to prevent race conditions.

30777 The existing *fstat*( ) function is not suitable for this purpose. We realize that implementations
30778 may wish to provide other attributes of typed memory objects (for example, alignment
30779 requirements, page size, and so on). The *fstat*( ) function returns a structure which is not
30780 extensible and, furthermore, contains substantial information that is inappropriate for typed
30781 memory objects.

30782 **FUTURE DIRECTIONS**

30783 None.

30784 **SEE ALSO**

30785 *fstat*( ), *mmap*( ), *posix_typed_mem_open*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
30786 **<sys/mman.h>**

30787 **CHANGE HISTORY**

30788 First released in Issue 6. Derived from IEEE Std. 1003.1j-2000.

30789 **NAME**

30790 posix_typed_mem_open — open a typed memory object

30791 **SYNOPSIS**

30792 TYM
```
#include <sys/mman.h>

int posix_typed_mem_open(const char *name, int oflag, int tflag);
```
30793

30794

30795 **DESCRIPTION**

30796 The *posix_typed_mem_open*( ) function establishes a connection between the typed memory object
30797 specified by the string pointed to by *name* and a file descriptor. It creates an open file description
30798 that refers to the typed memory object and a file descriptor that refers to that open file
30799 description. The file descriptor is used by other functions to refer to that typed memory object. It
30800 is unspecified whether the name appears in the file system and is visible to other functions that
30801 take path names as arguments. The *name* argument shall conform to the construction rules for a
30802 path name. If *name* begins with the slash character, then processes calling
30803 *posix_typed_mem_open*( ) with the same value of *name* shall refer to the same typed memory
30804 object. If *name* does not begin with the slash character, the effect is implementation-defined. The
30805 interpretation of slash characters other than the leading slash character in *name* is
30806 implementation-defined.

30807 Each typed memory object supported in a system is identified by a name which specifies not
30808 only its associated typed memory pool, but also the path or port by which it is accessed. That is,
30809 the same typed memory pool accessed via several different ports has several different
30810 corresponding names. The binding between names and typed memory objects is established in
30811 an implementation-defined manner. Unlike shared memory objects, there is ordinarily no way
30812 for a program to create a typed memory object.

30813 The value of *tflag* determines how the typed memory object behaves when subsequently
30814 mapped by calls to *mmap*( ). At most, one of the following flags defined in **<sys/mman.h>** may
30815 be specified:

30816 POSIX_TYPED_MEM_ALLOCATE
30817 Allocate on *mmap*( ).

30818 POSIX_TYPED_MEM_ALLOCATE_CONTIG
30819 Allocate contiguously on *mmap*( ).

30820 POSIX_TYPED_MEM_MAP_ALLOCATABLE
30821 Map on *mmap*( ), without affecting allocatability.

30822 If *tflag* has the flag POSIX_TYPED_MEM_ALLOCATE specified, any subsequent call to *mmap*( )
30823 using the returned file descriptor shall result in allocation and mapping of typed memory from
30824 the specified typed memory pool. The allocated memory may be a contiguous previously
30825 unallocated area of the typed memory pool or several non-contiguous previously unallocated
30826 areas (mapped to a contiguous portion of the process address space). If *tflag* has the flag
30827 POSIX_TYPED_MEM_ALLOCATE_CONTIG specified, any subsequent call to *mmap*( ) using the
30828 returned file descriptor shall result in allocation and mapping of a single contiguous previously
30829 unallocated area of the typed memory pool (also mapped to a contiguous portion of the process
30830 address space). If *tflag* has none of the flags POSIX_TYPED_MEM_ALLOCATE or
30831 POSIX_TYPED_MEM_ALLOCATE_CONTIG specified, any subsequent call to *mmap*( ) using the
30832 returned file descriptor shall map an application-chosen area from the specified typed memory
30833 pool such that this mapped area becomes unavailable for allocation until unmapped by all
30834 processes. If *tflag* has the flag POSIX_TYPED_MEM_MAP_ALLOCATABLE specified, any
30835 subsequent call to *mmap*( ) using the returned file descriptor shall map an application-chosen
30836 area from the specified typed memory pool without an effect on the availability of that area for

allocation; that is, mapping such an object leaves each byte of the mapped area unallocated if it was unallocated prior to the mapping or allocated if it was allocated prior to the mapping. The appropriate privilege to specify the POSIX_TYPED_MEM_MAP_ALLOCATABLE flag is implementation-defined.

If successful, *posix_typed_mem_open*( ) returns a file descriptor for the typed memory object that is the lowest numbered file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor does not share it with any other processes. It is unspecified whether the file offset is set. The FD_CLOEXEC file descriptor flag associated with the new file descriptor shall be cleared.

The behavior of *msync*( ), *ftruncate*( ), and all file operations other than *mmap*( ), *posix_mem_offset*( ), *posix_typed_mem_get_info*( ), *fstat*( ), *dup*( ), *dup2*( ), and *close*( ), is unspecified when passed a file descriptor connected to a typed memory object by this function.

The file status flags of the open file description shall be set according to the value of *oflag*. Applications shall specify exactly one of the three access mode values described below and defined in the header **<fcntl.h>**, as the value of *oflag*.

O_RDONLY Open for read access only.

O_WRONLY Open for write access only.

O_RDWR Open for read or write access.

RETURN VALUE

Upon successful completion, the *posix_typed_mem_open*( ) function shall return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *posix_typed_mem_open*( ) function shall fail if:

[EACCES] The typed memory object exists and the permissions specified by *oflag* are denied.

[EINTR] The *posix_typed_mem_open*( ) operation was interrupted by a signal.

[EINVAL] The flags specified in *tflag* are invalid (more than one of POSIX_TYPED_MEM_ALLOCATE, POSIX_TYPED_MEM_ALLOCATE_CONTIG, or POSIX_TYPED_MEM_MAP_ALLOCATABLE is specified).

[EMFILE] Too many file descriptors are currently in use by this process.

[ENAMETOOLONG] The length of the *name* argument exceeds {PATH_MAX} or a path name component is longer than {NAME_MAX}.

[ENFILE] Too many file descriptors are currently open in the system.

[ENOENT] The named typed memory object does not exist.

[EPERM] The caller lacks the appropriate privilege to specify the flag POSIX_TYPED_MEM_MAP_ALLOCATABLE in argument *tflag*.

30876 **EXAMPLES**

30877 None.

30878 **APPLICATION USAGE**

30879 None.

30880 **RATIONALE**

30881 None.

30882 **FUTURE DIRECTIONS**

30883 None.

30884 **SEE ALSO**

30885 *close*( ), *dup*( ), *exec*, *fcntl*( ), *fstat*( ), *ftruncate*( ), *mmap*( ), *msync*( ), *posix_mem_offset*( ),
30886 *posix_typed_mem_get_info*( ), *umask*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
30887 **<fcntl.h,> <sys/mman.h>**

30888 **CHANGE HISTORY**

30889 First released in Issue 6. Derived from IEEE Std. 1003.1j-2000.

30890 **NAME**

30891 pow, powf, powl — power function

30892 **SYNOPSIS**

30893 `#include <math.h>`

30894 `double pow(double x, double y);`
30895 `float powf(float x, float y);`
30896 `long double powl(long double x, long double y);`

30897 **DESCRIPTION**

30898 CX The functionality described on this reference page is aligned with the ISO C standard. Any
30899 conflict between the requirements described here and the ISO C standard is unintentional. This
30900 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

30901 These functions shall compute the value of *x* raised to the power *y*, x^y. If *x* is negative, the
30902 application shall ensure that *y* is an integer value.

30903 An application wishing to check for error situations should set *errno* to 0 before calling *pow*(). If
30904 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

30905 **RETURN VALUE**

30906 Upon successful completion, these functions shall return the value of *x* raised to the power *y*.

30907 If *x* is 0 and *y* is 0, 1.0 shall be returned.

30908 XSI If *y* is NaN, or *y* is non-zero and *x* is NaN, NaN shall be returned and *errno* may be set to
30909 [EDOM]. If *y* is 0.0 and *x* is NaN, either 1.0 shall be returned, or NaN shall be returned and *errno*
30910 may be set to [EDOM].

30911 XSI If *x* is 0.0 and *y* is negative, –HUGE_VAL shall be returned and *errno* may be set to [EDOM] or
30912 [ERANGE].

30913 If the correct value would cause overflow, ±HUGE_VAL shall be returned, and *errno* set to
30914 [ERANGE].

30915 If the correct value would cause underflow, 0 is returned and *errno* may be set to [ERANGE].

30916 **ERRORS**

30917 These functions shall fail if:

30918 [EDOM] The value of *x* is negative and *y* is non-integral.

30919 [ERANGE] The value to be returned would have caused overflow.

30920 These functions may fail if:

30921 XSI [EDOM] The value of *x* is 0.0 and *y* is negative, or *y* is NaN.

30922 [ERANGE] The correct value would cause underflow.

30923 XSI No other errors shall occur.

30924 **EXAMPLES**

30925 None.

30926 **APPLICATION USAGE**

30927 None.

30928 **RATIONALE**

30929 None.

30930 **FUTURE DIRECTIONS**

30931 None.

30932 **SEE ALSO**

30933 *exp*( ), *isnan*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

30934 **CHANGE HISTORY**

30935 First released in Issue 1. Derived from Issue 1 of the SVID.

30936 **Issue 4**

30937 References to *matherr*( ) are removed.

30938 The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the
30939 ISO C standard and to rationalize error handling in the mathematics functions.

30940 The return value specified for [EDOM] is marked as an extension.

30941 **Issue 5**

30942 The DESCRIPTION is updated to indicate how an application should check for an error. This
30943 text was previously published in the APPLICATION USAGE section.

30944 **Issue 6**

30945 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

30946 The *powf*( ) and *powl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

30947 **NAME**

30948 pread — read from a file

30949 **SYNOPSIS**

30950 XSI
```
#include <unistd.h>
```

30951
```
ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);
```

30952

30953 **DESCRIPTION**

30954 Refer to *read*( ).

30955 **NAME**

30956 printf — print formatted output

30957 **SYNOPSIS**

30958 `#include <stdio.h>`

30959 `int printf(const char *restrict format, ...);` |

30960 **DESCRIPTION** |

30961 Refer to *fprintf*( ).

NAME

pselect, select — synchronous I/O multiplexing

SYNOPSIS

```
#include <sys/select.h>

int pselect(int nfds, fd_set *readfds, fd_set *writefds,
    fd_set *errorfds, const struct timespec *timeout,
    const sigset_t *sigmask);
int select(int nfds, fd_set *restrict readfds,
    fd_set *restrict writefds, fd_set *restrict errorfds,
    struct timeval *restrict timeout);
void FD_CLR(int fd, fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

DESCRIPTION

The *pselect*( ) function shall examine the file descriptor sets whose addresses are passed in the *readfds*, *writefds*, and *errorfds* parameters to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively.

The *select*( ) function shall be equivalent to the *pselect*( ) function, except as follows:

- For the *select*( ) function, the timeout period is given in seconds and microseconds in an argument of type **struct timeval**, whereas for the *pselect*( ) function the timeout period is given in seconds and nanoseconds in an argument of type **struct timespec**.
- The *select*( ) function has no *sigmask* argument; it shall behave as *pselect*( ) does when *sigmask* is a null pointer.
- Upon successful completion, the *select*( ) function may modify the object pointed to by the *timeout* argument.

The *pselect*( ) and *select*( ) functions shall support regular files, terminal and pseudo-terminal devices, XSR STREAMS-based files, FIFOs, pipes, and sockets. The behavior of *pselect*( ) and *select*( ) on file descriptors that refer to other types of file is unspecified.

The *nfds* argument specifies the range of descriptors to be tested. The first *nfds* descriptors shall be checked in each set; that is, the descriptors from zero through *nfds*−1 in the descriptor sets shall be examined.

If the *readfds* argument is not a null pointer, it points to an object of type **fd_set** that on input specifies the file descriptors to be checked for being ready to read, and on output indicates which file descriptors are ready to read.

If the *writefds* argument is not a null pointer, it points to an object of type **fd_set** that on input specifies the file descriptors to be checked for being ready to write, and on output indicates which file descriptors are ready to write.

If the *errorfds* argument is not a null pointer, it points to an object of type **fd_set** that on input specifies the file descriptors to be checked for error conditions pending, and on output indicates which file descriptors have error conditions pending.

Upon successful completion, the *pselect*( ) function shall modify the objects pointed to by the *readfds*, *writefds*, and *errorfds* arguments to indicate which file descriptors are ready for reading, ready for writing, or have an error condition pending, respectively, and shall return the total number of ready descriptors in all the output sets. For each file descriptor less than *nfds*, the corresponding bit shall be set on successful completion if it was set on input and the associated

condition is true for that file descriptor.

If none of the selected descriptors are ready for the requested operation, the *pselect*( ) function may block until at least one of the requested operations becomes ready. The parameter *timeout* controls how long the *pselect*( ) function may take to complete. If the *timeout* parameter is not a null pointer, it specifies a maximum interval to wait for the selection to complete. If the specified time interval expires without any requested operation becoming ready, the function shall return. If the *timeout* parameter is a null pointer, then the call to *pselect*( ) shall block indefinitely until at least one descriptor meets the specified criteria. To effect a poll, the *timeout* parameter should not be a null pointer, and should point to a zero-valued **timespec** structure.

The use of a timeout does not affect any pending timers set up by *alarm*( ), *ualarm*( ), or *setitimer*( ).

Implementations may place limitations on the maximum timeout interval supported. All implementations shall support a maximum timeout interval of at least 31 days. If the *timeout* argument specifies a timeout interval greater than the implementation-defined maximum value, the maximum value shall be used as the actual timeout value. Implementations may also place limitations on the granularity of timeout intervals. If the requested timeout interval requires a finer granularity than the implementation supports, the actual timeout interval shall be rounded up to the next supported value.

If *sigmask* is not a null pointer, then the *pselect*( ) function shall replace the signal mask of the process by the set of signals pointed to by *sigmask* before examining the descriptors, and shall restore the signal mask of the process before returning.

A descriptor shall be considered ready for reading when a call to an input function with O_NONBLOCK clear would not block, whether or not the function would transfer data successfully. (The function might return data, an end-of-file indication, or an error other than one indicating that it is blocked, and in each of these cases the descriptor shall be considered ready for reading.)

A descriptor shall be considered ready for writing when a call to an output function with O_NONBLOCK clear would not block, whether or not the function would transfer data successfully.

If a socket has a pending error, it shall be considered to have an exceptional condition pending. Otherwise, what constitutes an exceptional condition is file type-specific. For a file descriptor for use with a socket, it is protocol-specific except as noted below. For other file types it is implementation-defined. If the operation is meaningless for a particular file type, *pselect*( ) shall indicate that the descriptor is ready for read or write operations, and shall indicate that the descriptor has no exceptional condition pending.

If a descriptor refers to a socket, the implied input function is the *recvmsg*( ) function with parameters requesting normal and ancillary data, such that the presence of either type shall cause the socket to be marked as readable. The presence of out-of-band data will be checked if the socket option SO_OOBINLINE has been enabled, as out-of-band data is enqueued with normal data. If the socket is currently listening, then it will be marked as readable if an incoming connection request has been received, and a call to the *accept*( ) function is guaranteed to complete without blocking.

If a descriptor refers to a socket, the implied output function is the *sendmsg*( ) function supplying an amount of normal data equal to the current value of the SO_SNDLOWAT option for the socket. If a non-blocking call to the *connect*( ) function has been made for a socket, and the connection attempt has either succeeded or failed leaving a pending error, the socket shall be marked as writable.

A socket shall be considered to have an exceptional condition pending if a receive operation with O_NONBLOCK clear for the open file description and with the MSG_OOB flag set would return out-of-band data without blocking. (It is protocol-specific whether the MSG_OOB flag would be used to read out-of-band data.) A socket shall also be considered to have an exceptional condition pending if an out-of-band data mark is present in the receive queue. Other circumstances under which a socket may be considered to have an exceptional condition pending are protocol-specific and implementation-defined.

If the *readfds*, *writefds*, and *errorfds* arguments are all null pointers and the *timeout* argument is not a null pointer, *select*( ) blocks for the time specified, or until interrupted by a signal. If the *readfds*, *writefds*, and *errorfds* arguments are all null pointers and the *timeout* argument is a null pointer, *pselect*( ) blocks until interrupted by a signal.

File descriptors associated with regular files shall always select true for ready to read, ready to write, and error conditions.

On failure, the objects pointed to by the *readfds*, *writefds*, and *errorfds* arguments are not modified. If the timeout interval expires without the specified condition being true for any of the specified file descriptors, the objects pointed to by the *readfds*, *writefds*, and *errorfds* arguments have all bits set to 0.

File descriptor masks of type **fd_set** can be initialized and tested with *FD_CLR*( ), *FD_ISSET*( ), *FD_SET*( ), and *FD_ZERO*( ). It is unspecified whether each of these is a macro or a function. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with any of these names, the behavior is undefined.

The macro *FD_CLR*(*fd*, *fdsetp*) shall remove the file descriptor *fd* from the set pointed to by *fdsetp*. If *fd* is not a member of this set, there shall be no effect on the set, nor will an error be returned.

The macro *FD_ISSET*(*fd*, *fdsetp*) shall evaluate to non-zero if the file descriptor *fd* is a member of the set pointed to by *fdsetp*, and shall evaluate to zero otherwise.

The macro *FD_SET*(*fd*, *fdsetp*) shall add the file descriptor *fd* to the set pointed to by *fdsetp*. If the file descriptor *fd* is already in this set, there shall be no effect on the set, nor will an error be returned.

The macro *FD_ZERO*(*fdsetp*) shall initialize the descriptor set pointed to by *fdsetp* to the null set. No error is returned if the set is not empty at the time *FD_ZERO*( ) is invoked.

The behavior of these macros is undefined if the *fd* argument is less than 0 or greater than or equal to FD_SETSIZE, or if *fd* is not a valid file descriptor, or if any of the arguments are expressions with side effects.

RETURN VALUE

Upon successful completion, the *pselect*( ) and *select*( ) functions shall return the total number of bits set in the bit masks. Otherwise, –1 shall be returned, and shall set *errno* to indicate the error.

FD_CLR( ), *FD_SET*( ), and *FD_ZERO*( ) return no value. *FD_ISSET*( ) returns a non-zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and 0 otherwise.

ERRORS

Under the following conditions, *pselect*( ) and *select*( ) shall fail and set *errno* to:

[EBADF] One or more of the file descriptor sets specified a file descriptor that is not a valid open file descriptor.

[EINTR] The function was interrupted before any of the selected events occurred and before the timeout interval expired.

XSI If SA_RESTART has been set for the interrupting signal, it is implementation-defined whether the function restarts or returns with [EINTR].

[EINVAL] An invalid timeout interval was specified.

[EINVAL] The *nfds* argument is less than 0 or greater than FD_SETSIZE.

XSR [EINVAL] One of the specified file descriptors refers to a STREAM or multiplexer that is linked (directly or indirectly) downstream from a multiplexer.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

In previous versions of the Single UNIX Specification, the *select*( ) function was defined in the **<sys/time.h>** header. This is now changed to **<sys/select.h>**. The rationale for this change was as follows: the introduction of the *pselect*( ) function included the **<sys/select.h>** header and the **<sys/select.h>** header defines all the related definitions for the *pselect*( ) and *select*( ) functions. Backwards-compatibility to existing XSI implementations is handled by allowing **<sys/time.h>** to include **<sys/select.h>**.

FUTURE DIRECTIONS

None.

SEE ALSO

accept( ), *alarm*( ), *connect*( ), *fcntl*( ), *poll*( ), *read*( ), *recvmsg*( ), *sendmsg*( ), *setitimer*( ), *ualarm*( ), *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 corrigenda item U026/6 has been 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.

31143 **NAME**

31144 pthread_atfork — register fork handlers

31145 **SYNOPSIS**

31146 THR
```
#include <pthread.h>
#include <sys/types.h>
#include <unistd.h>

int pthread_atfork(void (*prepare)(void), void (*parent)(void),
    void (*child)(void));
```
31151

31152 **DESCRIPTION**

31153 The *pthread_atfork*( ) function shall declare fork handlers to be called before and after *fork*( ), in
31154 the context of the thread that called *fork*( ). The *prepare* fork handler shall be called before *fork*( )
31155 processing commences. The *parent* fork handle shall be called after *fork*( ) processing completes
31156 in the parent process. The *child* fork handler shall be called after *fork*( ) processing completes in
31157 the child process. If no handling is desired at one or more of these three points, the
31158 corresponding fork handler address(es) may be set to NULL.

31159 The order of calls to *pthread_atfork*( ) is significant. The *parent* and *child* fork handlers shall be
31160 called in the order in which they were established by calls to *pthread_atfork*( ). The *prepare* fork
31161 handlers shall be called in the opposite order.

31162 **RETURN VALUE**

31163 Upon successful completion, *pthread_atfork*( ) shall return a value of zero; otherwise, an error
31164 number shall be returned to indicate the error.

31165 **ERRORS**

31166 The *pthread_atfork*( ) function shall fail if:

31167 [ENOMEM] Insufficient table space exists to record the fork handler addresses. |

31168 The *pthread_atfork*( ) function shall not return an error code of [EINTR]. |

31169 **EXAMPLES**

31170 None.

31171 **APPLICATION USAGE**

31172 None.

31173 **RATIONALE**

31174 There are at least two serious problems with the semantics of *fork*( ) in a multi-threaded
31175 program. One problem has to do with state (for example, memory) covered by mutexes.
31176 Consider the case where one thread has a mutex locked and the state covered by that mutex is
31177 inconsistent while another thread calls *fork*( ). In the child, the mutex is in the locked state
31178 (locked by a nonexistent thread and thus can never be unlocked). Having the child simply
31179 reinitialize the mutex is unsatisfactory since this approach does not resolve the question about
31180 how to correct or otherwise deal with the inconsistent state in the child.

31181 It is suggested that programs that use *fork*( ) call an *exec* function very soon afterwards in the
31182 child process, thus resetting all states. In the meantime, only a short list of async-signal-safe
31183 library routines are promised to be available.

31184 Unfortunately, this solution does not address the needs of multi-threaded libraries. Application
31185 programs may not be aware that a multi-threaded library is in use, and they feel free to call any
31186 number of library routines between the *fork*( ) and *exec* calls, just as they always have. Indeed,
31187 they may be extant single-threaded programs and cannot, therefore, be expected to obey new
31188 restrictions imposed by the threads library.

On the other hand, the multi-threaded library needs a way to protect its internal state during *fork*( ) in case it is re-entered later in the child process. The problem arises especially in multi-threaded I/O libraries, which are almost sure to be invoked between the *fork*( ) and *exec* calls to effect I/O redirection. The solution may require locking mutex variables during *fork*( ), or it may entail simply resetting the state in the child after the *fork*( ) processing completes.

The *pthread_atfork*( ) function provides multi-threaded libraries with a means to protect themselves from innocent application programs that call *fork*( ), and it provides multi-threaded application programs with a standard mechanism for protecting themselves from *fork*( ) calls in a library routine or the application itself.

The expected usage is that the *prepare* handler acquires all mutex locks and the other two fork handlers release them.

For example, an application can supply a *prepare* routine that acquires the necessary mutexes the library maintains and supply *child* and *parent* routines that release those mutexes, thus ensuring that the child gets a consistent snapshot of the state of the library (and that no mutexes are left stranded). Alternatively, some libraries might be able to supply just a *child* routine that re-initializes the mutexes in the library and all associated states to some known value (for example, what it was when the image was originally executed).

When *fork*( ) is called, only the calling thread is duplicated in the child process. Synchronization variables remain in the same state in the child as they were in the parent at the time *fork*( ) was called. Thus, for example, mutex locks may be held by threads that no longer exist in the child process, and any associated states may be inconsistent. The parent process may avoid this by explicit code that acquires and releases locks critical to the child via *pthread_atfork*( ). In addition, any critical threads need to be recreated and re-initialized to the proper state in the child (also via *pthread_atfork*( )).

A higher-level package may acquire locks on its own data structures before invoking lower-level packages. Under this scenario, the order specified for fork handler calls allows a simple rule of initialization for avoiding package deadlock: a package initializes all packages on which it depends before it calls the *pthread_atfork*( ) function for itself.

FUTURE DIRECTIONS

None.

SEE ALSO

atexit( ), *fork*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 5. Derived from the POSIX Threads Extension.

IEEE PASC Interpretation 1003.1c #4 is included.

Issue 6

The *pthread_atfork*( ) function is marked as part of the Threads option.

The **<pthread.h>** header is added to the SYNOPSIS.

31227 **NAME**

31228 pthread_attr_destroy, pthread_attr_init — destroy and initialize threads attributes object

31229 **SYNOPSIS**

31230 THR
```
#include <pthread.h>

int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_init(pthread_attr_t *attr);
```
31233

31234 **DESCRIPTION**

31235 The *pthread_attr_destroy*( ) function is used to destroy a thread attributes object. An
31236 implementation may cause *pthread_attr_destroy*( ) to set *attr* to an implementation-defined
31237 invalid value. The behavior of using the attribute after it has been destroyed is undefined.

31238 The *pthread_attr_init*( ) function initializes a thread attributes object *attr* with the default value
31239 for all of the individual attributes used by a given implementation.

31240 The resulting attributes object (possibly modified by setting individual attribute values), when
31241 used by *pthread_create*( ) defines the attributes of the thread created. A single attributes object can
31242 be used in multiple simultaneous calls to *pthread_create*( ). The effect of initializing an already
31243 initialized thread attributes object is undefined.

31244 **RETURN VALUE**

31245 Upon successful completion, *pthread_attr_destroy*( ) and *pthread_attr_init*( ) shall return a value of
31246 0; otherwise, an error number shall be returned to indicate the error.

31247 **ERRORS**

31248 The *pthread_attr_init*( ) function shall fail if:

31249 [ENOMEM] Insufficient memory exists to initialize the thread attributes object.

31250 These functions shall not return an error code of [EINTR].

31251 **EXAMPLES**

31252 None.

31253 **APPLICATION USAGE**

31254 None.

31255 **RATIONALE**

31256 Attributes objects are provided for threads, mutexes, and condition variables as a mechanism to
31257 support probable future standardization in these areas without requiring that the function itself
31258 be changed.

31259 Attributes objects provide clean isolation of the configurable aspects of threads. For example,
31260 ‘‘stack size’’ is an important attribute of a thread, but it cannot be expressed portably. When
31261 porting a threaded program, stack sizes often need to be adjusted. The use of attributes objects
31262 can help by allowing the changes to be isolated in a single place, rather than being spread across
31263 every instance of thread creation.

31264 Attributes objects can be used to set up ‘‘classes’ of threads with similar attributes; for example,
31265 ‘‘threads with large stacks and high priority’’ or ‘‘threads with minimal stacks’’. These classes
31266 can be defined in a single place and then referenced wherever threads need to be created.
31267 Changes to ‘‘class’’ decisions become straightforward, and detailed analysis of each
31268 *pthread_create*( ) call is not required.

31269 The attributes objects are defined as opaque types as an aid to extensibility. If these objects had
31270 been specified as structures, adding new attributes would force recompilation of all multi-
31271 threaded programs when the attributes objects are extended; this might not be possible if

different program components were supplied by different vendors.

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.

Because 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.

The following proposal was provided as a suggested alternative to the supplied attributes:

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).

The difficulties with this solution are:

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 or 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-bit out of a possible 16-bit 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).

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 discontiguous and can be allocated and released on demand.

The attribute mechanism has been designed in large measure for extensibility. Future extensions to the attribute mechanism or to any attributes object defined in this volume of IEEE Std. 1003.1-200x has to be done with care so as not to affect binary-compatibility.

Attributes objects, even if allocated by means of dynamic allocation functions such as *malloc*( ), may have their size fixed at compile time. This means, for example, a *pthread_create*( ) in an implementation with extensions to the **pthread_attr_t** cannot look beyond the area that the binary application assumes is valid. This suggests that implementations should maintain a size field in the attributes object, as well as possibly version information, if extensions in different directions (possibly by different vendors) are to be accommodated.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_getstackaddr( ), *pthread_attr_getstacksize*( ), *pthread_attr_getdetachstate*( ), *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_destroy*( ) and *pthread_attr_init*( ) functions marked as part of the Threads option.

IEEE PASC Interpretation 1003.1 #107 is applied, noting that the effect of initializing an already initialized thread attributes object is undefined.

31330 **NAME**

31331 pthread_attr_getdetachstate, pthread_attr_setdetachstate — get or set detachstate attribute

31332 **SYNOPSIS**

31333 THR
```
#include <pthread.h>

int pthread_attr_getdetachstate(const pthread_attr_t *attr,
    int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```
31337

31338 **DESCRIPTION**

31339 The *detachstate* attribute controls whether the thread is created in a detached state. If the thread
31340 is created detached, then use of the ID of the newly created thread by the *pthread_detach*( ) or
31341 *pthread_join*( ) function is an error.

31342 The *pthread_attr_getdetachstate*( ) and *pthread_attr_setdetachstate*( ) functions, respectively, get and
31343 set the *detachstate* attribute in the *attr* object.

31344 For *pthread_attr_getdetachstate*( ), *detachstate* shall be set to either
31345 PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE.

31346 For *pthread_attr_setdetachstate*( ), the application shall set *detachstate* to either
31347 PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE.

31348 A value of PTHREAD_CREATE_DETACHED causes all threads created with *attr* to be in the
31349 detached state, whereas using a value of PTHREAD_CREATE_JOINABLE shall cause all threads
31350 created with *attr* to be in the joinable state. The default value of the *detachstate* attribute is
31351 PTHREAD_CREATE_JOINABLE.

31352 **RETURN VALUE**

31353 Upon successful completion, *pthread_attr_getdetachstate*( ) and *pthread_attr_setdetachstate*( ) shall
31354 return a value of 0; otherwise, an error number shall be returned to indicate the error.

31355 The *pthread_attr_getdetachstate*( ) function stores the value of the *detachstate* attribute in *detachstate*
31356 if successful.

31357 **ERRORS**

31358 The *pthread_attr_setdetachstate*( ) function shall fail if:

31359 [EINVAL] The value of *detachstate* was not valid |

31360 These functions shall not return an error code of [EINTR]. |

31361 **EXAMPLES**

31362 None.

31363 **APPLICATION USAGE**

31364 None.

31365 **RATIONALE**

31366 None.

31367 **FUTURE DIRECTIONS**

31368 None.

31369 **SEE ALSO**

31370 *pthread_attr_destroy*( ), *pthread_attr_getstackaddr*( ), *pthread_attr_getstacksize*( ), *pthread_create*( ), the |
31371 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 DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

pthread_attr_getguardsize, pthread_attr_setguardsize — get or set the thread guardsize attribute

SYNOPSIS

XSI

```
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
    size_t *restrict guardsize);
int pthread_attr_setguardsize(pthread_attr_t *attr,
    size_t guardsize);
```

DESCRIPTION

The *guardsize* attribute controls the size of the guard area for the created thread's stack. The *guardsize* attribute provides protection against overflow of the stack pointer. If a thread's stack is created with guard protection, the implementation allocates extra memory at the overflow end of the stack as a buffer against stack overflow of the stack pointer. If an application overflows into this buffer an error results (possibly in a SIGSEGV signal being delivered to the thread).

The *pthread_attr_getguardsize*( ) function gets the *guardsize* attribute in the *attr* object. This attribute is returned in the *guardsize* parameter.

The *pthread_attr_setguardsize*( ) function sets the *guardsize* attribute in the *attr* object. The new value of this attribute is obtained from the *guardsize* parameter. If *guardsize* is zero, a guard area shall not be provided for threads created with *attr*. If *guardsize* is greater than zero, a guard area of at least size *guardsize* bytes is provided for each thread created with *attr*.

A conforming implementation is permitted to round up the value contained in *guardsize* to a multiple of the configurable system variable {PAGESIZE} (see **<sys/mman.h>**). If an implementation rounds up the value of *guardsize* to a multiple of {PAGESIZE}, a call to *pthread_attr_getguardsize*( ) specifying *attr* shall store in the *guardsize* parameter the guard size specified by the previous *pthread_attr_setguardsize*( ) function call.

The default value of the *guardsize* attribute is {PAGESIZE} bytes. The actual value of {PAGESIZE} is implementation-defined and may not be the same on all implementations.

If the *stackaddr* attribute has been set (that is, the caller is allocating and managing its own thread stacks), the *guardsize* attribute is ignored and no protection shall be provided by the implementation. It is the responsibility of the application to manage stack overflow along with stack allocation and management in this case.

RETURN VALUE

If successful, the *pthread_attr_getguardsize*( ) and *pthread_attr_setguardsize*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_attr_getguardsize*( ) and *pthread_attr_setguardsize*( ) functions shall fail if:

[EINVAL] The attribute *attr* is invalid.

[EINVAL] The parameter *guardsize* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *guardsize* attribute is provided to the application for two reasons:

1. Overflow protection can potentially result in wasted system resources. An application that creates a large number of threads, and which knows its threads never overflow their stack, can save system resources by turning off guard areas.
2. When threads allocate large data structures on the stack, large guard areas may be needed to detect stack overflow.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std. 1003.1-200x, **<pthread.h>**, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

In the ERRORS section, a third [EINVAL] error condition is removed as it is covered by the second error condition.

The **restrict** keyword is added to the *pthread_attr_getguardsize*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

31441 **NAME**

31442 pthread_attr_getinheritsched, pthread_attr_setinheritsched — get and set inheritsched attribute
31443 (**REALTIME THREADS**)

31444 **SYNOPSIS**

31445 TPS
```
#include <pthread.h>

int pthread_attr_getinheritsched(const pthread_attr_t *restrict attr,
    int *restrict inheritsched);
int pthread_attr_setinheritsched(pthread_attr_t *attr,
    int inheritsched);
```
31450

31451 **DESCRIPTION**

31452 The *pthread_attr_getinheritsched*( ), and *pthread_attr_setinheritsched*( ) functions, respectively, get
31453 and set the *inheritsched* attribute in the *attr* argument.

31454 When the attributes objects are used by *pthread_create*( ), the *inheritsched* attribute determines
31455 how the other scheduling attributes of the created thread shall be set:

31456 PTHREAD_INHERIT_SCHED
31457 Specifies that the scheduling policy and associated attributes shall be inherited from the
31458 creating thread, and the scheduling attributes in this *attr* argument shall be ignored.

31459 PTHREAD_EXPLICIT_SCHED
31460 Specifies that the scheduling policy and associated attributes shall be set to the
31461 corresponding values from this attributes object.

31462 The symbols PTHREAD_INHERIT_SCHED and PTHREAD_EXPLICIT_SCHED are defined in
31463 the header **<pthread.h>**.

31464 **RETURN VALUE**

31465 If successful, the *pthread_attr_getinheritsched*( ) and *pthread_attr_setinheritsched*( ) functions shall
31466 return zero; otherwise, an error number shall be returned to indicate the error.

31467 **ERRORS**

31468 The *pthread_attr_setinheritsched*( ) function may fail if:

31469 [EINVAL] The value of *inheritsched* is not valid.

31470 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

31471 These functions shall not return an error code of [EINTR].

31472 **EXAMPLES**

31473 None.

31474 **APPLICATION USAGE**

31475 After these attributes have been set, a thread can be created with the specified attributes using
31476 *pthread_create*( ). Using these routines does not affect the current running thread.

31477 **RATIONALE**

31478 None.

31479 **FUTURE DIRECTIONS**

31480 None.

31481 **SEE ALSO**

31482 *pthread_attr_destroy*( ), *pthread_attr_getscope*( ), *pthread_attr_getschedpolicy*( ),
31483 *pthread_attr_getschedparam*( ), *pthread_create*( ), the Base Definitions volume of |
31484 IEEE Std. 1003.1-200x, **<pthread.h>**, **<sched.h>** |

31485 **CHANGE HISTORY**

31486 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31487 Marked as part of the Realtime Threads Feature Group.

31488 **Issue 6**

31489 The *pthread_attr_getinheritsched*( ) and *pthread_attr_setinheritsched*( ) functions are marked as part |
31490 of the Thread Execution Scheduling option. |

31491 The [ENOSYS] error condition has been removed as stubs need not be provided if an
31492 implementation does not support the Thread Execution Scheduling option. |

31493 The **restrict** keyword is added to the *pthread_attr_getinheritsched*( ) prototype for alignment with |
31494 the ISO / IEC 9899: 1999 standard. |

NAME

pthread_attr_getschedparam, pthread_attr_setschedparam — get and set schedparam attribute

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_attr_getschedparam(const pthread_attr_t *restrict attr,
    struct sched_param *restrict param);
int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
    const struct sched_param *restrict param);
```

DESCRIPTION

The *pthread_attr_getschedparam*( ), and *pthread_attr_setschedparam*( ) functions, respectively, get and set the scheduling parameter attributes in the *attr* argument. The contents of the *param* structure are defined in **<sched.h>**. For the SCHED_FIFO and SCHED_RR policies, the only required member of *param* is *sched_priority*.

TSP

For the SCHED_SPORADIC policy, the required members of the *param* structure are *sched_priority*, *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*, and *sched_ss_max_repl*. The specified *sched_ss_repl_period* must be greater than or equal to the specified *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail. The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the function to succeed; if not, the function shall fail.

RETURN VALUE

If successful, the *pthread_attr_getschedparam*( ) and *pthread_attr_setschedparam*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_attr_setschedparam*( ) function may fail if:

[EINVAL] The value of *param* is not valid.

[ENOTSUP] An attempt was made to set the attribute to an unsupported value.

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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

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>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_attr_getschedparam*( ) and *pthread_attr_setschedparam*( ) functions are marked as part of the Threads 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_getschedparam*( ) and *pthread_attr_setschedparam*( ) prototypes for alignment with the ISO / IEC 9899: 1999 standard.

31544 **NAME**

31545 pthread_attr_getschedpolicy, pthread_attr_setschedpolicy — get and set schedpolicy attribute
31546 (**REALTIME THREADS**)

31547 **SYNOPSIS**

31548 TPS
```
#include <pthread.h>

int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr,
    int *restrict policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```
31552

31553 **DESCRIPTION**

31554 The *pthread_attr_getschedpolicy*( ) and *pthread_attr_setschedpolicy*( ) functions, respectively, get and
31555 set the *schedpolicy* attribute in the *attr* argument.

31556 The supported values of *policy* shall include SCHED_FIFO, SCHED_RR, and SCHED_OTHER,
31557 which are defined by the header **<sched.h>**. When threads executing with the scheduling policy
31558 TSP SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC are waiting on a mutex, they acquire the
31559 mutex in priority order when the mutex is unlocked.

31560 **RETURN VALUE**

31561 If successful, the *pthread_attr_getschedpolicy*( ) and *pthread_attr_setschedpolicy*( ) functions shall
31562 return zero; otherwise, an error number shall be returned to indicate the error.

31563 **ERRORS**

31564 The *pthread_attr_setschedpolicy*( ) function may fail if:

31565 [EINVAL] The value of *policy* is not valid.

31566 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

31567 These functions shall not return an error code of [EINTR].

31568 **EXAMPLES**

31569 None.

31570 **APPLICATION USAGE**

31571 After these attributes have been set, a thread can be created with the specified attributes using
31572 *pthread_create*( ). Using these routines does not affect the current running thread.

31573 **RATIONALE**

31574 None.

31575 **FUTURE DIRECTIONS**

31576 None.

31577 **SEE ALSO**

31578 *pthread_attr_destroy*( ), *pthread_attr_getscope*( ), *pthread_attr_getinheritsched*( ),
31579 *pthread_attr_getschedparam*( ), *pthread_create*( ), the Base Definitions volume of
31580 IEEE Std. 1003.1-200x, **<pthread.h>**, **<sched.h>**

31581 **CHANGE HISTORY**

31582 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31583 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 Thread Execution Scheduling option.

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.

31592 **NAME**

31593 pthread_attr_getscope, pthread_attr_setscope — get and set contentionscope attribute
31594 (**REALTIME THREADS**)

31595 **SYNOPSIS**

31596 TPS
```
#include <pthread.h>

int pthread_attr_getscope(const pthread_attr_t *restrict attr,
    int *restrict contentionscope);
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

31600

31601 **DESCRIPTION**

31602 The *pthread_attr_getscope*( ) and *pthread_attr_setscope*( ) functions, respectively, are used to get
31603 and set the *contentionscope* attribute in the *attr* object.

31604 The *contentionscope* attribute may have the values PTHREAD_SCOPE_SYSTEM, signifying
31605 system scheduling contention scope, or PTHREAD_SCOPE_PROCESS, signifying process
31606 scheduling contention scope. The symbols PTHREAD_SCOPE_SYSTEM and
31607 PTHREAD_SCOPE_PROCESS are defined by the header **<pthread.h>**.

31608 **RETURN VALUE**

31609 If successful, the *pthread_attr_getscope*( ) and *pthread_attr_setscope*( ) functions shall return zero;
31610 otherwise, an error number shall be returned to indicate the error.

31611 **ERRORS**

31612 The *pthread_attr_setscope*( ) function may fail if:

31613 [EINVAL] The value of *contentionscope* is not valid.

31614 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

31615 These functions shall not return an error code of [EINTR].

31616 **EXAMPLES**

31617 None.

31618 **APPLICATION USAGE**

31619 After these attributes have been set, a thread can be created with the specified attributes using
31620 *pthread_create*( ). Using these routines does not affect the current running thread.

31621 **RATIONALE**

31622 None.

31623 **FUTURE DIRECTIONS**

31624 None.

31625 **SEE ALSO**

31626 *pthread_attr_destroy*( ), *pthread_attr_getinheritsched*( ), *pthread_attr_getschedpolicy*( ),
31627 *pthread_attr_getschedparam*( ), *pthread_create*( ), the Base Definitions volume of
31628 IEEE Std. 1003.1-200x, **<pthread.h>**, **<sched.h>**

31629 **CHANGE HISTORY**

31630 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31631 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 Thread Execution Scheduling option.

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.

NAME

pthread_attr_getstack, pthread_attr_setstack — get and set stack attribute

SYNOPSIS

XSI

```
#include <pthread.h>

int pthread_attr_getstack(const pthread_attr_t *attr, void **stackaddr,
    size_t *stacksize);
int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
    size_t stacksize);
```

DESCRIPTION

The *pthread_attr_getstack*( ) and *pthread_attr_setstack*( ) functions, respectively, shall get and set the thread creation stack attribute in the *attr* object.

The *stack* attribute specifies the area of storage to be used for the created thread's stack. The base (lowest addressable byte) of the storage is *stackaddr*, and the size of the storage is *stacksize* bytes. The *stacksize* shall be at least {PTHREAD_STACK_MIN}. The *stackaddr* shall be aligned appropriately to be used as a stack; for example, *pthread_attr_setstack*( ) may fail with [EINVAL] if (*stackaddr* 0x7) is not 0. All pages within the stack described by *stackaddr* and *stacksize* shall be both readable and writable by the thread.

RETURN VALUE

Upon successful completion, these functions shall return a value of 0; otherwise, an error number shall be returned to indicate the error.

The *pthread_attr_getstack*( ) function shall store the stack attribute values in *stackaddr* and *stacksize* if successful.

ERRORS

The *pthread_attr_setstack*( ) function shall fail if:

[EINVAL] The value of *stacksize* is less than {PTHREAD_STACK_MIN} or exceeds an implementation-defined limit.

The *pthread_attr_setstack*( ) function may fail if:

[EINVAL] The value of *stackaddr* does not have proper alignment to be used as a stack, or if (*stackaddr* + *stacksize*) lacks proper alignment.

[EACCES] The stack page(s) described by *stackaddr* and *stacksize* are not both readable and writable by the thread.

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.

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.

NAME

pthread_attr_getstackaddr, pthread_attr_setstackaddr — get and set stackaddr attribute

SYNOPSIS

THR TSA
```
#include <pthread.h>

int pthread_attr_getstackaddr(const pthread_attr_t *restrict attr,
    void **restrict stackaddr);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

DESCRIPTION

The *pthread_attr_getstackaddr*( ) and *pthread_attr_setstackaddr*( ) functions, respectively, get and set the thread creation *stackaddr* attribute in the *attr* object.

The *stackaddr* attribute specifies the location of storage to be used for the created thread's stack. The size of the storage is at least {PTHREAD_STACK_MIN}.

RETURN VALUE

Upon successful completion, *pthread_attr_getstackaddr*( ) and *pthread_attr_setstackaddr*( ) shall return a value of 0; otherwise, an error number shall be returned to indicate the error.

The *pthread_attr_getstackaddr*( ) function stores the *stackaddr* attribute value in *stackaddr* if successful.

ERRORS

No errors are defined.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_destroy( ), *pthread_attr_getdetachstate*( ), *pthread_attr_getstacksize*( ), *pthread_create*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<limits.h>**, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_attr_getstackaddr*( ) and *pthread_attr_setstackaddr*( ) functions are marked as part of the Threads and Thread Stack Address Attribute options.

The **restrict** keyword is added to the *pthread_attr_getstackaddr*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

31731 **NAME**

31732 pthread_attr_getstacksize, pthread_attr_setstacksize — get and set stacksize attribute

31733 **SYNOPSIS**

31734 THR TSA
```
#include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
    size_t *restrict stacksize);
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```
31738

31739 **DESCRIPTION**

31740 The *pthread_attr_getstacksize*( ) and *pthread_attr_setstacksize*( ) functions, respectively, get and set
31741 the thread creation *stacksize* attribute in the *attr* object.

31742 The *stacksize* attribute defines the minimum stack size (in bytes) allocated for the created threads
31743 stack.

31744 **RETURN VALUE**

31745 Upon successful completion, *pthread_attr_getstacksize*( ) and *pthread_attr_setstacksize*( ) shall
31746 return a value of 0; otherwise, an error number shall be returned to indicate the error.

31747 The *pthread_attr_getstacksize*( ) function stores the *stacksize* attribute value in *stacksize* if
31748 successful.

31749 **ERRORS**

31750 The *pthread_attr_setstacksize*( ) function shall fail if:

31751 [EINVAL] The value of *stacksize* is less than {PTHREAD_STACK_MIN} or exceeds a
31752 system-imposed limit.

31753 These functions shall not return an error code of [EINTR].

31754 **EXAMPLES**

31755 None.

31756 **APPLICATION USAGE**

31757 None.

31758 **RATIONALE**

31759 None.

31760 **FUTURE DIRECTIONS**

31761 None.

31762 **SEE ALSO**

31763 *pthread_attr_destroy*( ), *pthread_attr_getstackaddr*( ), *pthread_attr_getdetachstate*( ), *pthread_create*( ),
31764 the Base Definitions volume of IEEE Std. 1003.1-200x, **<limits.h>**, **<pthread.h>**

31765 **CHANGE HISTORY**

31766 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31767 **Issue 6**

31768 The *pthread_attr_getstacksize*( ) and *pthread_attr_setstacksize*( ) functions are marked as part of the
31769 Threads and Thread Stack Address Attribute options.

31770 The **restrict** keyword is added to the *pthread_attr_getstacksize*( ) prototype for alignment with the
31771 ISO / IEC 9899: 1999 standard.

31772 **NAME**

31773 pthread_attr_init — initialize threads attributes object

31774 **SYNOPSIS**

31775 THR
```
#include <pthread.h>
```

31776
```
int pthread_attr_init(pthread_attr_t *attr);
```

31777

31778 **DESCRIPTION**

31779 Refer to *pthread_attr_destroy*( ).

31780 **NAME**

31781 pthread_attr_setdetachstate — set detachstate attribute

31782 **SYNOPSIS**

31783 THR
```
#include <pthread.h>
```

31784
```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

31785

31786 **DESCRIPTION**

31787 Refer to *pthread_attr_getdetachstate*( ).

31788 **NAME**

31789 pthread_attr_setguardsize — set thread guardsize attribute

31790 **SYNOPSIS**

31791 XSI
```
#include <pthread.h>
```

31792
```
int pthread_attr_setguardsize(pthread_attr_t *attr,
    size_t guardsize);
```

31793

31794

31795 **DESCRIPTION**

31796 Refer to *pthread_attr_getguardsize*( ).

31797 **NAME**

31798 pthread_attr_setinheritsched — set inheritsched attribute (**REALTIME THREADS**)

31799 **SYNOPSIS**

31800 TPS
```
#include <pthread.h>
```

31801
```
int pthread_attr_setinheritsched(pthread_attr_t *attr,
```
31802
```
    int inheritsched);
```
31803

31804 **DESCRIPTION**

31805 Refer to *pthread_attr_getinheritsched*( ).

31806 **NAME**

31807 pthread_attr_setschedparam — set schedparam attribute

31808 **SYNOPSIS**

31809 THR
```
#include <pthread.h>
```

31810 31811 31812
```
int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
    const struct sched_param *restrict param);
```

31813 **DESCRIPTION**

31814 Refer to *pthread_attr_getschedparam*( ).

31815 **NAME**

31816 pthread_attr_setschedpolicy — set schedpolicy attribute (**REALTIME THREADS**)

31817 **SYNOPSIS**

31818 TPS
```
#include <pthread.h>
```

31819
```
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

31820

31821 **DESCRIPTION**

31822 Refer to *pthread_attr_getschedpolicy*( ).

31823 **NAME**

31824 pthread_attr_setscope — set contentionscope attribute (**REALTIME THREADS**)

31825 **SYNOPSIS**

31826 TPS
```
#include <pthread.h>
```

31827
```
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

31828

31829 **DESCRIPTION**

31830 Refer to *pthread_attr_getscope*( ).

31831 **NAME**
31832 pthread_attr_setstack — set stack attribute

31833 **SYNOPSIS**
31834 XSI
```
#include <pthread.h>

int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
    size_t stacksize);
```
31835
31836
31837

31838 **DESCRIPTION**
31839 Refer to *pthread_attr_getstack*( ).

31840 **NAME**

31841 pthread_attr_setstackaddr — set stackaddr attribute

31842 **SYNOPSIS**

31843 THR TSA
```
#include <pthread.h>
```

31844
```
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

31845

31846 **DESCRIPTION**

31847 Refer to *pthread_attr_getstackaddr*( ).

31848 **NAME**

31849 pthread_attr_setstacksize — set stacksize attribute

31850 **SYNOPSIS**

31851 THR TSA
```
#include <pthread.h>

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```
31852

31853

31854 **DESCRIPTION**

31855 Refer to *pthread_attr_getstacksize*( ).

NAME

pthread_barrier_destroy, pthread_barrier_init — destroy and initialize a barrier object

SYNOPSIS

BAR
```
#include <pthread.h>

int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
    const pthread_barrierattr_t *restrict attr, unsigned count);
```

DESCRIPTION

The *pthread_barrier_destroy*( ) function destroys the barrier referenced by *barrier* and releases any resources used by the barrier. The effect of subsequent use of the barrier is undefined until the barrier is reinitialized by another call to *pthread_barrier_init*( ). An implementation may use this function to set *barrier* to an invalid value. The results are undefined if *pthread_barrier_destroy*( ) is called when any thread is blocked on the barrier, or if this function is called with an uninitialized barrier.

The *pthread_barrier_init*( ) function shall allocate any resources required to use the barrier referenced by *barrier* and initializes the barrier with attributes referenced by *attr*. If *attr* is NULL, the default barrier attributes are used; the effect is the same as passing the address of a default barrier attributes object. The results are undefined if *pthread_barrier_init*( ) is called when any thread is blocked on the barrier (that is, has not returned from the *pthread_barrier_wait*( ) call). The results are undefined if a barrier is used without first being initialized. The results are undefined if *pthread_barrier_init*( ) is called specifying an already initialized barrier.

The *count* argument specifies the number of threads that must call *pthread_barrier_wait*( ) before any of them successfully return from the call. The value specified by *count* must be greater than zero.

If the *pthread_barrier_init*( ) function fails, the barrier is not initialized and the contents of *barrier* are undefined.

Only the object referenced by *barrier* may be used for performing synchronization. The result of referring to copies of that object in calls to *pthread_barrier_destroy*( ) or *pthread_barrier_wait*( ) is undefined.

RETURN VALUE

Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_barrier_destroy*( ) function may fail if:

[EBUSY] The implementation has detected an attempt to destroy a barrier while it is in use (for example, while being used in a *pthread_barrier_wait*( ) call) by another thread.

[EINVAL] The value specified by *barrier* is invalid.

The *pthread_barrier_init*( ) function shall fail if:

[EAGAIN] The system lacks the necessary resources to initialize another barrier.

[EINVAL] The value specified by *count* is equal to zero.

[ENOMEM] Insufficient memory exists to initialize the barrier.

The *pthread_barrier_init*( ) function may fail if:

[EBUSY] The implementation has detected an attempt to reinitialize a barrier while it is in use (for example, while being used in a *pthread_barrier_wait*( ) call) by another thread.

[EINVAL] The value specified by *attr* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_barrier_destroy*( ) and *pthread_barrier_init*( ) functions are part of the Barriers option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_barrier_wait( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std. 1003.1j-2000.

NAME

pthread_barrier_init — initialize a barrier object

SYNOPSIS

BAR

```
#include <pthread.h>

int pthread_barrier_init(pthread_barrier_t *restrict barrier,
    const pthread_barrierattr_t *restrict attr, unsigned count);
```

DESCRIPTION

Refer to *pthread_barrier_destroy*( ).

31927 **NAME**

31928 pthread_barrier_wait — synchronize at a barrier

31929 **SYNOPSIS**

31930 BAR
```
#include <pthread.h>

int pthread_barrier_wait(pthread_barrier_t *barrier);
```
31931

31932

31933 **DESCRIPTION**

31934 The *pthread_barrier_wait*( ) function synchronizes participating threads at the barrier referenced
31935 by *barrier*. The calling thread blocks (that is, does not return from the *pthread_barrier_wait*( ) call)
31936 until the required number of threads have called *pthread_barrier_wait*( ) specifying the barrier.

31937 When the required number of threads have called *pthread_barrier_wait*( ) specifying the barrier,
31938 the constant PTHREAD_BARRIER_SERIAL_THREAD is is returned to one unspecified thread
31939 and zero is returned to each of the remaining threads. At this point, the barrier is reset to the
31940 state it had as a result of the most recent *pthread_barrier_init*( ) function that referenced it.

31941 The constant PTHREAD_BARRIER_SERIAL_THREAD is defined in **<pthread.h>** and its value
31942 is distinct from any other value returned by *pthread_barrier_wait*( ).

31943 The results are undefined if this function is called with an uninitialized barrier.

31944 If a signal is delivered to a thread blocked on a barrier, upon return from the signal handler the
31945 thread shall resume waiting at the barrier if the barrier wait has not completed (that is, if the
31946 required number of threads have not arrived at the barrier during the execution of the signal
31947 handler); otherwise, the thread shall continue as normal from the completed barrier wait. Until
31948 the thread in the signal handler returns from it, it is unspecified whether other threads may
31949 proceed past the barrier once they have all reached it.

31950 A thread that has blocked on a barrier shall not prevent any unblocked thread that is eligible to
31951 use the same processing resources from eventually making forward progress in its execution.
31952 Eligibility for processing resources shall be determined by the scheduling policy.

31953 **RETURN VALUE**

31954 Upon successful completion, the *pthread_barrier_wait*( ) function shall return
31955 PTHREAD_BARRIER_SERIAL_THREAD for a single (arbitrary) thread synchronized at the
31956 barrier and zero for each of the other threads. Otherwise, an error number shall be returned to
31957 indicate the error.

31958 **ERRORS**

31959 The *pthread_barrier_wait*( ) function may fail if:

31960 [EINVAL] The value specified by *barrier* does not refer to an initialized barrier object.

31961 This function shall not return an error code of [EINTR]. |

31962 **EXAMPLES**

31963 None.

31964 **APPLICATION USAGE**

31965 Applications using this function may be subject to priority inversion, as discussed in the Base |
31966 Definitions volume of IEEE Std. 1003.1-200x, Section 3.287, Priority Inversion. |

31967 The *pthread_barrier_wait*( ) function is part of the Barriers option and need not be provided on all |
31968 implementations.

31969 **RATIONALE**

31970 None.

31971 **FUTURE DIRECTIONS**

31972 None.

31973 **SEE ALSO**

31974 *pthread_barrier_destroy*( ), *pthread_barrier_init*( ), the Base Definitions volume of |
31975 IEEE Std. 1003.1-200x, **<pthread.h>** |

31976 **CHANGE HISTORY**

31977 First released in Issue 6. Derived from IEEE Std. 1003.1j-2000.

31978 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

31979 **NAME**

31980 pthread_barrierattr_destroy, pthread_barrierattr_init — destroy and initialize barrier attributes
31981 object

31982 **SYNOPSIS**

31983 BAR
```
#include <pthread.h>

int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
```
31986

31987 **DESCRIPTION**

31988 The *pthread_barrierattr_destroy*( ) function destroys a barrier attributes object. The effect of
31989 subsequent use of the object is undefined until the object is reinitialized by another call to
31990 *pthread_barrierattr_init*( ). An implementation may cause *pthread_barrierattr_destroy*( ) to set the
31991 object referenced by *attr* to an invalid value.

31992 The *pthread_barrierattr_init*( ) function initializes a barrier attributes object *attr* with the default
31993 value for all of the attributes defined by the implementation.

31994 The results are undefined if *pthread_barrierattr_init*( ) is called specifying an already initialized
31995 barrier attributes object.

31996 After a barrier attributes object has been used to initialize one or more barriers, any function
31997 affecting the attributes object (including destruction) does not affect any previously initialized
31998 barrier.

31999 **RETURN VALUE**

32000 If successful, the *pthread_barrierattr_destroy*( ) and *pthread_barrierattr_init*( ) functions shall return
32001 zero; otherwise, an error number shall be returned to indicate the error.

32002 **ERRORS**

32003 The *pthread_barrierattr_destroy*( ) function may fail if:

32004 [EINVAL] The value specified by *attr* is invalid.

32005 The *pthread_barrierattr_init*( ) function shall fail if:

32006 [ENOMEM] Insufficient memory exists to initialize the barrier attributes object.

32007 These functions shall not return an error code of [EINTR]. |

32008 **EXAMPLES**

32009 None.

32010 **APPLICATION USAGE**

32011 The *pthread_barrierattr_destroy*( ) and *pthread_barrierattr_init*( ) functions are part of the Barriers |
32012 option and need not be provided on all implementations. |

32013 **RATIONALE**

32014 None.

32015 **FUTURE DIRECTIONS**

32016 None.

32017 **SEE ALSO**

32018 *pthread_barrierattr_getpshared*( ), *pthread_barrierattr_setpshared*( ), the Base Definitions volume of |
32019 IEEE Std. 1003.1-200x, **<pthread.h>**. |

32020 **CHANGE HISTORY**

32021 First released in Issue 6. Derived from IEEE Std. 1003.1j-2000.

32022 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

32023 **NAME**

32024 pthread_barrierattr_getpshared, pthread_barrierattr_setpshared — get and set process-shared
32025 attribute of barrier attributes object

32026 **SYNOPSIS**

32027 BAR TSH
```
#include <pthread.h>

int pthread_barrierattr_getpshared(const pthread_barrierattr_t *restrict attr,
    int *restrict pshared);
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
    int pshared);
```
32032

32033 **DESCRIPTION**

32034 The process-shared attribute is set to PTHREAD_PROCESS_SHARED to permit a barrier to be
32035 operated upon by any thread that has access to the memory where the barrier is allocated. If the
32036 process-shared attribute is PTHREAD_PROCESS_PRIVATE, the barrier shall only be operated
32037 upon by threads created within the same process as the thread that initialized the barrier; if
32038 threads of different processes attempt to operate on such a barrier, the behavior is undefined.
32039 The default value of the attribute shall be PTHREAD_PROCESS_PRIVATE. Both constants
32040 PTHREAD_PROCESS_SHARED and PTHREAD_PROCESS_PRIVATE are defined in
32041 **<pthread.h>**.

32042 The *pthread_barrierattr_getpshared*( ) function obtains the value of the process-shared attribute
32043 from the attributes object referenced by *attr*. The *pthread_barrierattr_setpshared*( ) function is used
32044 to set the process-shared attribute in an initialized attributes object referenced by *attr*.

32045 Additional attributes, their default values, and the names of the associated functions to get and
32046 set those attribute values are implementation-defined.

32047 **RETURN VALUE**

32048 If successful, the *pthread_barrierattr_getpshared*( ) function shall return zero and store the value of
32049 the process-shared attribute of *attr* into the object referenced by the *pshared* parameter.
32050 Otherwise, an error number shall be returned to indicate the error.

32051 If successful, the *pthread_barrierattr_setpshared*( ) function shall return zero; otherwise, an error
32052 number shall be returned to indicate the error.

32053 **ERRORS**

32054 These functions may fail if:

32055 [EINVAL] The value specified by *attr* is invalid.

32056 The *pthread_barrierattr_setpshared*( ) function may fail if:

32057 [EINVAL] The new value specified for the process-shared attribute is not one of the legal
32058 values PTHREAD_PROCESS_SHARED or PTHREAD_PROCESS_PRIVATE.

32059 These functions shall not return an error code of [EINTR].

32060 **EXAMPLES**

32061 None.

32062 **APPLICATION USAGE**

32063 The *pthread_barrierattr_getpshared*( ) and *pthread_barrierattr_setpshared*( ) functions are part of the |
32064 Barriers option and need not be provided on all implementations. |

32065 **RATIONALE**

32066 None.

32067 **FUTURE DIRECTIONS**

32068 None.

32069 **SEE ALSO**

32070 *pthread_barrier_init*( ), *pthread_barrierattr_destroy*( ), *pthread_barrierattr_init*( ), the Base Definitions |
32071 volume of IEEE Std. 1003.1-200x, **<pthread.h>** |

32072 **CHANGE HISTORY**

32073 First released in Issue 6. Derived from IEEE Std. 1003.1j-2000 |

32074 **NAME**

32075 pthread_barrierattr_init — initialize barrier attributes object

32076 **SYNOPSIS**

32077 BAR
```
#include <pthread.h>
```

32078
```
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
```

32079

32080 **DESCRIPTION**

32081 Refer to *pthread_barrierattr_destroy*( ).

32082 **NAME**

32083 pthread_barrierattr_setpshared — set process-shared attribute of barrier attributes object

32084 **SYNOPSIS**

32085 BAR TSH
```
#include <pthread.h>

int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
    int pshared);
```
32088

32089 **DESCRIPTION**

32090 Refer to *pthread_barrierattr_getpshared*( ).

32091 **NAME**

32092 pthread_cancel — cancel execution of a thread

32093 **SYNOPSIS**

32094 THR
```
#include <pthread.h>
```

32095
```
int pthread_cancel(pthread_t thread);
```

32096

32097 **DESCRIPTION**

32098 The *pthread_cancel*( ) function requests that *thread* be canceled. The target thread's cancelability
32099 state and type determines when the cancelation takes effect. When the cancelation is acted on,
32100 the cancelation cleanup handlers for *thread* are called. When the last cancelation cleanup handler
32101 returns, the thread-specific data destructor functions shall be called for *thread*. When the last
32102 destructor function returns, *thread* shall be terminated.

32103 The cancelation processing in the target thread runs asynchronously with respect to the calling
32104 thread returning from *pthread_cancel*( ).

32105 **RETURN VALUE**

32106 If successful, the *pthread_cancel*( ) function shall return zero; otherwise, an error number shall be
32107 returned to indicate the error.

32108 **ERRORS**

32109 The *pthread_cancel*( ) function may fail if:

32110 [ESRCH] No thread could be found corresponding to that specified by the given thread
32111 ID.

32112 The *pthread_cancel*( ) function shall not return an error code of [EINTR].

32113 **EXAMPLES**

32114 None.

32115 **APPLICATION USAGE**

32116 None.

32117 **RATIONALE**

32118 Two alternative functions were considered to sending the cancelation notification to a thread.
32119 One would be to define a new SIGCANCEL signal that had the cancelation semantics when
32120 delivered; the other was to define the new *pthread_cancel*( ) function, which would trigger the
32121 cancelation semantics.

32122 The advantage of a new signal was that so much of the delivery criteria were identical to that
32123 used when trying to deliver a signal that making cancelation notification a signal was seen as
32124 consistent. Indeed, many implementations implement cancelation using a special signal. On the
32125 other hand, there would be no signal functions that could be used with this signal except
32126 *pthread_kill*( ), and the behavior of the delivered cancelation signal would be unlike any
32127 previously existing defined signal.

32128 The benefits of a special function include the recognition that this signal would be defined
32129 because of the similar delivery criteria and that this is the only common behavior between a
32130 cancelation request and a signal. In addition, the cancelation delivery mechanism does not have
32131 to be implemented as a signal. There are also strong, if not stronger, parallels with language
32132 exception mechanisms than with signals that are potentially obscured if the delivery mechanism
32133 is visibly closer to signals.

32134 In the end, it was considered that as there were so many exceptions to the use of the new signal
32135 with existing signals functions that it would be misleading. A special function has resolved this

problem. This function was carefully defined so that an implementation wishing to provide the cancelation functions on top of signals could do so. The special function also means that implementations are not obliged to implement cancelation with signals.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_exit( ), *pthread_cond_wait*( ), *pthread_cond_timedwait*( ), *pthread_join*( ), *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_cancel*( ) function is marked as part of the Threads option.

32148 **NAME**

32149 pthread_cleanup_pop, pthread_cleanup_push — establish cancelation handlers

32150 **SYNOPSIS**

32151 THR
```
#include <pthread.h>

void pthread_cleanup_pop(int execute);
void pthread_cleanup_push(void (*routine)(void*), void *arg);
```
32154

32155 **DESCRIPTION**

32156 The *pthread_cleanup_pop*( ) function shall remove the routine at the top of the calling thread's
32157 cancelation cleanup stack and optionally invoke it (if *execute* is non-zero).

32158 The *pthread_cleanup_push*( ) function shall push the specified cancelation cleanup handler *routine*
32159 onto the calling thread's cancelation cleanup stack. The cancelation cleanup handler shall be
32160 popped from the cancelation cleanup stack and invoked with the argument *arg* when:

32161 • The thread exits (that is, calls *pthread_exit*( )).

32162 • The thread acts upon a cancelation request.

32163 • The thread calls *pthread_cleanup_pop*( ) with a non-zero *execute* argument.

32164 These functions may be implemented as macros. The application shall ensure that they appear
32165 as statements, and in pairs within the same lexical scope (that is, the *pthread_cleanup_push*( )
32166 macro may be thought to expand to a token list whose first token is `'{'` with
32167 *pthread_cleanup_pop*( ) expanding to a token list whose last token is the corresponding `'}'`).

32168 The effect of calling *longjmp*( ) or *siglongjmp*( ) is undefined if there have been any calls to
32169 *pthread_cleanup_push*( ) or *pthread_cleanup_pop*( ) made without the matching call since the jump
32170 buffer was filled. The effect of calling *longjmp*( ) or *siglongjmp*( ) from inside a cancelation
32171 cleanup handler is also undefined unless the jump buffer was also filled in the cancelation
32172 cleanup handler.

32173 **RETURN VALUE**

32174 The *pthread_cleanup_push*( ) and *pthread_cleanup_pop*( ) functions shall return no value.

32175 **ERRORS**

32176 No errors are defined.

32177 These functions shall not return an error code of [EINTR].

32178 **EXAMPLES**

32179 The following is an example using thread primitives to implement a cancelable, writers-priority
32180 read-write lock:

```
32181 typedef struct {
32182     pthread_mutex_t lock;
32183     pthread_cond_t rcond,
32184         wcond;
32185     int lock_count; /* < 0 .. Held by writer. */
32186                     /* > 0 .. Held by lock_count readers. */
32187                     /* = 0 .. Held by nobody. */
32188     int waiting_writers; /* Count of waiting writers. */
32189 } rwlock;

32190 void
32191 waiting_reader_cleanup(void *arg)
32192 {
```

```
32193           rwlock *l;
32194           l = (rwlock *) arg;
32195           pthread_mutex_unlock(&l->lock);
32196       }
32197       void
32198       lock_for_read(rwlock *l)
32199       {
32200           pthread_mutex_lock(&l->lock);
32201           pthread_cleanup_push(waiting_reader_cleanup, l);
32202           while ((l->lock_count < 0) && (l->waiting_writers != 0))
32203               pthread_cond_wait(&l->rcond, &l->lock);
32204           l->lock_count++;
32205          /*
32206           * Note the pthread_cleanup_pop executes
32207           * waiting_reader_cleanup.
32208           */
32209           pthread_cleanup_pop(1);
32210       }
32211       void
32212       release_read_lock(rwlock *l)
32213       {
32214           pthread_mutex_lock(&l->lock);
32215           if (--l->lock_count == 0)
32216               pthread_cond_signal(&l->wcond);
32217           pthread_mutex_unlock(l);
32218       }
32219       void
32220       waiting_writer_cleanup(void *arg)
32221       {
32222           rwlock *l;
32223           l = (rwlock *) arg;
32224           if ((--l->waiting_writers == 0) && (l->lock_count >= 0)) {
32225              /*
32226               * This only happens if we have been canceled.
32227               */
32228               pthread_cond_broadcast(&l->wcond);
32229       }
32230           pthread_mutex_unlock(&l->lock);
32231       }
32232       void
32233       lock_for_write(rwlock *l)
32234       {
32235           pthread_mutex_lock(&l->lock);
32236           l->waiting_writers++;
32237           pthread_cleanup_push(waiting_writer_cleanup, l);
32238           while (l->lock_count != 0)
32239               pthread_cond_wait(&l->wcond, &l->lock);
32240           l->lock_count = -1;
32241          /*
```

```
32242           * Note the pthread_cleanup_pop executes
32243           * waiting_writer_cleanup.
32244           */
32245          pthread_cleanup_pop(1);
32246      }
32247      void
32248      release_write_lock(rwlock *l)
32249      {
32250          pthread_mutex_lock(&l->lock);
32251          l->lock_count = 0;
32252          if (l->waiting_writers == 0)
32253              pthread_cond_broadcast(&l->rcond)
32254          else
32255              pthread_cond_signal(&l->wcond);
32256          pthread_mutex_unlock(&l->lock);
32257      }
32258      /*
32259       * This function is called to initialize the read/write lock.
32260       */
32261      void
32262      initialize_rwlock(rwlock *l)
32263      {
32264          pthread_mutex_init(&l->lock, pthread_mutexattr_default);
32265          pthread_cond_init(&l->wcond, pthread_condattr_default);
32266          pthread_cond_init(&l->rcond, pthread_condattr_default);
32267          l->lock_count = 0;
32268          l->waiting_writers = 0;
32269      }
32270      reader_thread()
32271      {
32272          lock_for_read(&lock);
32273          pthread_cleanup_push(release_read_lock, &lock);
32274         /*
32275          * Thread has read lock.
32276          */
32277          pthread_cleanup_pop(1);
32278      }
32279      writer_thread()
32280      {
32281          lock_for_write(&lock);
32282          pthread_cleanup_push(release_write_lock, &lock);
32283         /*
32284          * Thread has write lock.
32285          */
32286      pthread_cleanup_pop(1);
32287      }
```

APPLICATION USAGE

The two routines that push and pop cancelation 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 cancelation 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 cancelation 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 cancelation 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 cancelation 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 of 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 cancelation cleanup-safe function. If an application has no cancelation points in its signal handlers, blocks any signal whose handler may have cancelation points while calling async-unsafe functions, or disables cancelation while calling async-unsafe functions, all functions may be safely called from cancelation cleanup routines.

FUTURE DIRECTIONS

None.

32332 **SEE ALSO**

32333 *pthread_cancel*( ), *pthread_setcancelstate*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
32334 **<pthread.h>**

32335 **CHANGE HISTORY**

32336 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32337 **Issue 6**

32338 The *pthread_cleanup_pop*( ) and *pthread_cleanup_push*( ) functions are marked as part of the |
32339 Threads option. |

32340 The APPLICATION USAGE section is added.

32341 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

32342 **NAME**

32343 pthread_cond_broadcast, pthread_cond_signal — broadcast or signal a condition

32344 **SYNOPSIS**

32345 THR
```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```
32348

32349 **DESCRIPTION**

32350 These functions are used to unblock threads blocked on a condition variable.

32351 The *pthread_cond_broadcast*( ) function shall unblock all threads currently blocked on the
32352 specified condition variable *cond*.

32353 The *pthread_cond_signal*( ) function shall unblock at least one of the threads that are blocked on
32354 the specified condition variable *cond* (if any threads are blocked on *cond*).

32355 If more than one thread is blocked on a condition variable, the scheduling policy determines the
32356 order in which threads are unblocked. When each thread unblocked as a result of a
32357 *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ) returns from its call to *pthread_cond_wait*( ) or
32358 *pthread_cond_timedwait*( ), the thread owns the mutex with which it called *pthread_cond_wait*( ) or
32359 *pthread_cond_timedwait*( ). The thread(s) that are unblocked shall contend for the mutex
32360 according to the scheduling policy (if applicable), and as if each had called *pthread_mutex_lock*( ).

32361 The *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ) functions may be called by a thread
32362 whether or not it currently owns the mutex that threads calling *pthread_cond_wait*( ) or
32363 *pthread_cond_timedwait*( ) have associated with the condition variable during their waits;
32364 however, if predictable scheduling behavior is required, then that mutex shall be locked by the
32365 thread calling *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ).

32366 The *pthread_cond_broadcast*( ) and *pthread_cond_signal*( ) functions have no effect if there are no
32367 threads currently blocked on *cond*.

32368 **RETURN VALUE**

32369 If successful, the *pthread_cond_broadcast*( ) and *pthread_cond_signal*( ) functions shall return zero;
32370 otherwise, an error number shall be returned to indicate the error.

32371 **ERRORS**

32372 The *pthread_cond_broadcast*( ) and *pthread_cond_signal*( ) function may fail if:

32373 [EINVAL] The value *cond* does not refer to an initialized condition variable. |

32374 These functions shall not return an error code of [EINTR]. |

32375 **EXAMPLES**

32376 None.

32377 **APPLICATION USAGE**

32378 The *pthread_cond_broadcast*( ) function is used whenever the shared-variable state has been
32379 changed in a way that more than one thread can proceed with its task. Consider a single
32380 producer/multiple consumer problem, where the producer can insert multiple items on a list
32381 that is accessed one item at a time by the consumers. By calling the *pthread_cond_broadcast*( )
32382 function, the producer would notify all consumers that might be waiting, and thereby the
32383 application would receive more throughput on a multiprocessor. In addition,
32384 *pthread_cond_broadcast*( ) makes it easier to implement a read-write lock. The |
32385 *pthread_cond_broadcast*( ) function is needed in order to wake up all waiting readers when a
32386 writer releases its lock. Finally, the two-phase commit algorithm can use this broadcast function

to notify all clients of an impending transaction commit.

It is not safe to use the *pthread_cond_signal*() function in a signal handler that is invoked asynchronously. Even if it were safe, there would still be a race between the test of the Boolean *pthread_cond_wait*() that could not be efficiently eliminated.

Mutexes and condition variables are thus not suitable for releasing a waiting thread by signaling from code running in a signal handler.

RATIONALE

Multiple Awakenings by Condition Signal

On a multiprocessor, it may be impossible for an implementation of *pthread_cond_signal*() to avoid the unblocking of more than one thread blocked on a condition variable. For example, consider the following partial implementation of *pthread_cond_wait*() and *pthread_cond_signal*(), executed by two threads in the order given. One thread is trying to wait on the condition variable, another is concurrently executing *pthread_cond_signal*(), while a third thread is already waiting.

```
pthread_cond_wait(mutex, cond):
    value = cond->value; /* 1 */
    pthread_mutex_unlock(mutex); /* 9 */
    pthread_mutex_lock(cond->mutex); /* 10 */
    if (value == cond->value) { /* 11 */
        me->next_cond = cond->waiter;
        cond->waiter = me;
        pthread_mutex_unlock(cond->mutex);
        unable_to_run(me);
    } else
        pthread_mutex_unlock(cond->mutex); /* 12 */
    pthread_mutex_lock(mutex); /* 13 */

pthread_cond_signal(cond):
    pthread_mutex_lock(cond->mutex); /* 2 */
    cond->value++; /* 3 */
    if (cond->waiter) { /* 4 */
        sleeper = cond->waiter; /* 5 */
        cond->waiter = sleeper->next_cond; /* 6 */
        able_to_run(sleeper); /* 7 */
    }
    pthread_mutex_unlock(cond->mutex); /* 8 */
```

The effect is that more than one thread can return from its call to *pthread_cond_wait*() or *pthread_cond_timedwait*() as a result of one call to *pthread_cond_signal*(). This effect is called ‘‘spurious wakeup’’. Note that the situation is self-correcting in that the number of threads that are so awakened is finite; for example, the next thread to call *pthread_cond_wait*() after the sequence of events above blocks.

While this problem could be resolved, the loss of efficiency for a fringe condition that occurs only rarely is unacceptable, especially given that one has to check the predicate associated with a condition variable anyway. Correcting this problem would unnecessarily reduce the degree of concurrency in this basic building block for all higher-level synchronization operations.

An added benefit of allowing spurious wakeups is that applications are forced to code a predicate-testing-loop around the condition wait. This also makes the application tolerate superfluous condition broadcasts or signals on the same condition variable that may be coded in

some other part of the application. The resulting applications are thus more robust. Therefore, IEEE Std. 1003.1-200x explicitly documents that spurious wakeups may occur.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_destroy(), *pthread_cond_timedwait*(), *pthread_cond_wait*(), 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_cond_broadcast*() and *pthread_cond_signal*() functions are marked as part of the Threads option.

The APPLICATION USAGE section is added.

NAME

pthread_cond_destroy, pthread_cond_init — destroy and initialize condition variables

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *restrict cond,
    const pthread_condattr_t *restrict attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

DESCRIPTION

The *pthread_cond_destroy*( ) function destroys the given condition variable specified by *cond*; the object becomes, in effect, uninitialized. An implementation may cause *pthread_cond_destroy*( ) to set the object referenced by *cond* to an invalid value. A destroyed condition variable object can be re-initialized using *pthread_cond_init*( ); the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.

The *pthread_cond_init*( ) function initializes the condition variable referenced by *cond* with attributes referenced by *attr*. If *attr* is NULL, the default condition variable attributes are used; the effect is the same as passing the address of a default condition variable attributes object. Upon successful initialization, the state of the condition variable becomes initialized.

Only *cond* itself may be used for performing synchronization. The result of referring to copies of *cond* in calls to *pthread_cond_wait*( ), *pthread_cond_timedwait*( ), *pthread_cond_signal*( ), *pthread_cond_broadcast*( ), and *pthread_cond_destroy*( ) is undefined.

Attempting to initialize an already initialized condition variable results in undefined behavior.

In cases where default condition variable attributes are appropriate, the macro PTHREAD_COND_INITIALIZER can be used to initialize condition variables that are statically allocated. The effect shall be equivalent to dynamic initialization by a call to *pthread_cond_init*( ) with parameter *attr* specified as NULL, except that no error checks are performed.

RETURN VALUE

If successful, the *pthread_cond_destroy*( ) and *pthread_cond_init*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

The [EBUSY] and [EINVAL] error checks, if implemented, shall act as if they were performed immediately at the beginning of processing for the function and caused an error return prior to modifying the state of the condition variable specified by *cond*.

ERRORS

The *pthread_cond_destroy*( ) function may fail if:

[EBUSY] The implementation has detected an attempt to destroy the object referenced by *cond* while it is referenced (for example, while being used in a *pthread_cond_wait*( ) or *pthread_cond_timedwait*( )) by another thread.

[EINVAL] The value specified by *cond* is invalid.

The *pthread_cond_init*( ) function shall fail if:

[EAGAIN] The system lacked the necessary resources (other than memory) to initialize another condition variable.

[ENOMEM] Insufficient memory exists to initialize the condition variable.

The *pthread_cond_init*( ) function may fail if:

[EBUSY] The implementation has detected an attempt to re-initialize the object referenced by *cond*, a previously initialized, but not yet destroyed, condition variable.

[EINVAL] The value specified by *attr* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

A condition variable can be destroyed immediately after all the threads that are blocked on it are awakened. For example, consider the following code:

```
struct list {
    pthread_mutex_t lm;
    ...
}

struct elt {
    key k;
    int busy;
    pthread_cond_t notbusy;
    ...
}

/* Find a list element and reserve it. */
struct elt *
list_find(struct list *lp, key k)
{
    struct elt *ep;

    pthread_mutex_lock(&lp->lm);
    while ((ep = find_elt(l, k) != NULL) && ep->busy)
        pthread_cond_wait(&ep->notbusy, &lp->lm);
    if (ep != NULL)
        ep->busy = 1;
    pthread_mutex_unlock(&lp->lm);
    return(ep);
}

delete_elt(struct list *lp, struct elt *ep)
{
    pthread_mutex_lock(&lp->lm);
    assert(ep->busy);
    ... remove ep from list ...
    ep->busy = 0;  /* Paranoid. */
(A) pthread_cond_broadcast(&ep->notbusy);
    pthread_mutex_unlock(&lp->lm);
(B) pthread_cond_destroy(&rp->notbusy);
    free(ep);
}
```

In this example, the condition variable and its list element may be freed (line B) immediately after all threads waiting for it are awakened (line A), since the mutex and the code ensure that no other thread can touch the element to be deleted.

32539 **APPLICATION USAGE**

32540 None.

32541 **RATIONALE**

32542 See *pthread_mutex_init*( ); a similar rationale applies to condition variables.

32543 **FUTURE DIRECTIONS**

32544 None.

32545 **SEE ALSO**

32546 *pthread_cond_broadcast*( ), *pthread_cond_signal*( ), *pthread_cond_timedwait*( ), *pthread_cond_wait*( ), |
32547 the Base Definitions volume of IEEE Std. 1003.1-200x, **<pthread.h>** |

32548 **CHANGE HISTORY**

32549 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32550 **Issue 6**

32551 The *pthread_cond_destroy*( ) and *pthread_cond_init*( ) functions are marked as part of the Threads |
32552 option. |

32553 IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION. |

32554 The **restrict** keyword is added to the *pthread_cond_init*( ) prototype for alignment with the |
32555 ISO / IEC 9899: 1999 standard. |

32556 **NAME**

32557 pthread_cond_init — initialize condition variables

32558 **SYNOPSIS**

32559 THR
```
#include <pthread.h>
```

32560
```
int pthread_cond_init(pthread_cond_t *restrict cond,
    const pthread_condattr_t *restrict attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```
32563

32564 **DESCRIPTION**

32565 Refer to *pthread_cond_destroy*( ).

32566 **NAME**

32567 pthread_cond_signal — signal a condition

32568 **SYNOPSIS**

32569 THR
```
#include <pthread.h>
```

32570
```
int pthread_cond_signal(pthread_cond_t *cond);
```

32571

32572 **DESCRIPTION**

32573 Refer to *pthread_cond_broadcast*( ).

NAME

pthread_cond_timedwait, pthread_cond_wait — wait on a condition

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
```

DESCRIPTION

The *pthread_cond_timedwait*( ) and *pthread_cond_wait*( ) functions are used to block on a condition variable. They shall be called with *mutex* locked by the calling thread or undefined behavior results.

These functions atomically release *mutex* and cause the calling thread to block on the condition variable *cond*; atomically here means ''atomically with respect to access by another thread to the mutex and then the condition variable''. That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ) in that thread shall behave as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex has been locked and is owned by the calling thread.

When using condition variables there is always a Boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) functions may occur. Since the return from *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) does not imply anything about the value of this predicate, the predicate should be re-evaluated upon such return.

The effect of using more than one mutex for concurrent *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) operations on the same condition variable is undefined; that is, a condition variable becomes bound to a unique mutex when a thread waits on the condition variable, and this (dynamic) binding ends when the wait returns.

A condition wait (whether timed or not) is a cancelation point. When the cancelability enable state of a thread is set to PTHREAD_CANCEL_DEFERRED, a side effect of acting upon a cancelation request while in a condition wait is that the mutex is (in effect) re-acquired before calling the first cancelation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up to the point of returning from the call to *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ), but at that point notices the cancelation request and instead of returning to the caller of *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ), starts the thread cancelation activities, which includes calling cancelation cleanup handlers.

A thread that has been unblocked because it has been canceled while blocked in a call to *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) shall not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

The *pthread_cond_timedwait*( ) function is the same as *pthread_cond_wait*( ) except that an error is returned if the absolute time specified by *abstime* passes (that is, system time equals or exceeds *abstime*) before the condition *cond* is signaled or broadcasted, or if the absolute time specified by *abstime* has already been passed at the time of the call. CS If the Clock Selection option is supported, the condition variable shall have a clock attribute which specifies the clock that shall be used to

measure the time specified by the *abstime* argument. When such timeouts occur, *pthread_cond_timedwait*( ) shall nonetheless release and re-acquire the mutex referenced by *mutex*. The *pthread_cond_timedwait*( ) function is also a cancelation point.

If a signal is delivered to a thread waiting for a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it was not interrupted, or it shall return zero due to spurious wakeup.

RETURN VALUE

Except in the case of [ETIMEDOUT], all these error checks shall act as if they were performed immediately at the beginning of processing for the function and shall cause an error return, in effect, prior to modifying the state of the mutex specified by *mutex* or the condition variable specified by *cond*.

Upon successful completion, a value of zero shall be returned; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_cond_timedwait*( ) function shall fail if:

[ETIMEDOUT] The time specified by *abstime* to *pthread_cond_timedwait*( ) has passed.

The *pthread_cond_timedwait*( ) and *pthread_cond_wait*( ) functions may fail if:

[EINVAL] The value specified by *cond*, *mutex*, or *abstime* is invalid.

[EINVAL] Different mutexes were supplied for concurrent *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) operations on the same condition variable.

[EPERM] The mutex was not owned by the current thread at the time of the call.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Condition Wait Semantics

It is important to note that when *pthread_cond_wait*( ) and *pthread_cond_timedwait*( ) return without error, the associated predicate may still be false. Similarly, when *pthread_cond_timedwait*( ) returns with the timeout error, the associated predicate may be true due to an unavoidable race between the expiration of the timeout and the predicate state change.

Some implementations, particularly on a multiprocessor, may sometimes cause multiple threads to wake up when the condition variable is signaled simultaneously on different processors.

In general, whenever a condition wait returns, the thread has to re-evaluate the predicate associated with the condition wait to determine whether it can safely proceed, should wait again, or should declare a timeout. A return from the wait does not imply that the associated predicate is either true or false.

It is thus recommended that a condition wait be enclosed in the equivalent of a ‘‘while loop’’ that checks the predicate.

32661 **Timed Wait Semantics**

32662 An absolute time measure was chosen for specifying the timeout parameter for two reasons.
32663 First, a relative time measure can be easily implemented on top of a function that specifies
32664 absolute time, but there is a race condition associated with specifying an absolute timeout on top
32665 of a function that specifies relative timeouts. For example, assume that *clock_gettime*( ) returns
32666 the current time and *cond_relative_timed_wait*( ) uses relative timeouts:

32667 ```
clock_gettime(CLOCK_REALTIME, &now)
32668 reltime = sleep_til_this_absolute_time -now;
32669 cond_relative_timed_wait(c, m, &reltime);
```

32670 If the thread is preempted between the first statement and the last statement, the thread blocks
32671 for too long. Blocking, however, is irrelevant if an absolute timeout is used. An absolute timeout
32672 also need not be recomputed if it is used multiple times in a loop, such as that enclosing a
32673 condition wait.

32674 For cases when the system clock is advanced discontinuously by an operator, it is expected that
32675 implementations process any timed wait expiring at an intervening time as if that time had
32676 actually occurred.

32677 **Cancelation and Condition Wait**

32678 A condition wait, whether timed or not, is a cancelation point. That is, the functions
32679 *pthread_cond_wait*( ) or *pthread_cond_timedwait*( ) are points where a pending (or concurrent)
32680 cancelation request is noticed. The reason for this is that an indefinite wait is possible at these
32681 points—whatever event is being waited for, even if the program is totally correct, might never
32682 occur; for example, some input data being awaited might never be sent. By making condition
32683 wait a cancelation point, the thread can be canceled and perform its cancelation cleanup handler
32684 even though it may be stuck in some indefinite wait.

32685 A side effect of acting on a cancelation request while a thread is blocked on a condition variable
32686 is to re-acquire the mutex before calling any of the cancelation cleanup handlers. This is done in
32687 order to ensure that the cancelation cleanup handler is executed in the same state as the critical
32688 code that lies both before and after the call to the condition wait function. This rule is also
32689 required when interfacing to POSIX threads from languages, such as Ada or C++, which may
32690 choose to map cancelation onto a language exception; this rule ensures that each exception
32691 handler guarding a critical section can always safely depend upon the fact that the associated
32692 mutex has already been locked regardless of exactly where within the critical section the
32693 exception was raised. Without this rule, there would not be a uniform rule that exception
32694 handlers could follow regarding the lock, and so coding would become very cumbersome.

32695 Therefore, since *some* statement has to be made regarding the state of the lock when a
32696 cancelation is delivered during a wait, a definition has been chosen that makes application
32697 coding most convenient and error free.

32698 When acting on a cancelation request while a thread is blocked on a condition variable, the
32699 implementation is required to ensure that the thread does not consume any condition signals
32700 directed at that condition variable if there are any other threads waiting on that condition
32701 variable. This rule is specified in order to avoid deadlock conditions that could occur if these two
32702 independent requests (one acting on a thread and the other acting on the condition variable)
32703 were not processed independently.
```

32704 **Performance of Mutexes and Condition Variables**

32705 Mutexes are expected to be locked only for a few instructions. This practice is almost
32706 automatically enforced by the desire of programmers to avoid long serial regions of execution
32707 (which would reduce total effective parallelism).

32708 When using mutexes and condition variables, one tries to ensure that the usual case is to lock the
32709 mutex, access shared data, and unlock the mutex. Waiting on a condition variable should be a
32710 relatively rare situation. For example, when implementing a read-write lock, code that acquires a
32711 read-lock typically needs only to increment the count of readers (under mutual-exclusion) and
32712 return. The calling thread would actually wait on the condition variable only when there is
32713 already an active writer. So the efficiency of a synchronization operation is bounded by the cost
32714 of mutex lock/unlock and not by condition wait. Note that in the usual case there is no context
32715 switch.

32716 This is not to say that the efficiency of condition waiting is unimportant. Since there needs to be
32717 at least one context switch per Ada rendezvous, the efficiency of waiting on a condition variable
32718 is important. The cost of waiting on a condition variable should be little more than the minimal
32719 cost for a context switch plus the time to unlock and lock the mutex.

32720 **Features of Mutexes and Condition Variables**

32721 It had been suggested that the mutex acquisition and release be decoupled from condition wait.
32722 This was rejected because it is the combined nature of the operation that, in fact, facilitates
32723 realtime implementations. Those implementations can atomically move a high-priority thread
32724 between the condition variable and the mutex in a manner that is transparent to the caller. This
32725 can prevent extra context switches and provide more deterministic acquisition of a mutex when
32726 the waiting thread is signaled. Thus, fairness and priority issues can be dealt with directly by the
32727 scheduling discipline. Furthermore, the current condition wait operation matches existing
32728 practice.

32729 **Scheduling Behavior of Mutexes and Condition Variables**

32730 Synchronization primitives that attempt to interfere with scheduling policy by specifying an
32731 ordering rule are considered undesirable. Threads waiting on mutexes and condition variables
32732 are selected to proceed in an order dependent upon the scheduling policy rather than in some
32733 fixed order (for example, FIFO or priority). Thus, the scheduling policy determines which
32734 thread(s) are awakened and allowed to proceed.

32735 **Timed Condition Wait**

32736 The *pthread_cond_timedwait*( ) function allows an application to give up waiting for a particular
32737 condition after a given amount of time. An example of its use follows:

```
32738 (void) pthread_mutex_lock(&t.mn);
32739         t.waiters++;
32740     clock_gettime(CLOCK_REALTIME, &ts);
32741     ts.tv_sec += 5;
32742     rc = 0;
32743     while (! mypredicate(&t) && rc == 0)
32744         rc = pthread_cond_timedwait(&t.cond, &t.mn, &ts);
32745     t.waiters--;
32746     if (rc == 0) setmystate(&t);
32747 (void) pthread_mutex_unlock(&t.mn);
```

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.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_signal( ), *pthread_cond_broadcast*( ), 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_cond_timedwait*( ) and *pthread_cond_wait*( ) functions are marked as part of the Threads option.

The Open Group corrigenda item U021/9 has been applied, correcting the prototype for the *pthread_cond_wait*( ) function.

The DESCRIPTION is updated for alignment with IEEE Std. 1003.1j-2000 by adding semantics for the Clock Selection option.

The ERRORS section has an additional case for [EPERM] in response to IEEE PASC Interpretation 1003.1c #28.

The **restrict** keyword is added to the *pthread_cond_timedwait*( ) and *pthread_cond_wait*( ) prototypes for alignment with the ISO/IEC 9899: 1999 standard.

32771 **NAME**

32772 pthread_cond_wait — wait on a condition

32773 **SYNOPSIS**

32774 THR
```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
```
32775 |
32776 |
32777 |

32778 **DESCRIPTION**

32779 Refer to *pthread_cond_timedwait*( ).

NAME

pthread_condattr_destroy, pthread_condattr_init — destroy and initialize condition variable attributes object

SYNOPSIS

THR

```
#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 destroys 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 condition variable attributes object can be re-initialized using *pthread_condattr_init*( ); the results of otherwise referencing the object after it has been destroyed are undefined.

The *pthread_condattr_init*( ) function initializes a condition variable attributes object *attr* with the default value for all of the attributes defined by the implementation.

Attempting to initialize an already initialized condition variable attributes object results in undefined behavior.

After a condition variable attributes object has been used to initialize one or more condition variables, any function affecting the attributes object (including destruction) does not affect any previously initialized condition variables.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined. |

RETURN VALUE

If successful, the *pthread_condattr_destroy*( ) and *pthread_condattr_init*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_condattr_destroy*( ) function may fail if:

[EINVAL] The value specified by *attr* is invalid. |

The *pthread_condattr_init*( ) function shall fail if:

[ENOMEM] Insufficient memory exists to initialize the condition variable attributes object. |

These functions shall not return an error code of [EINTR]. |

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

See *pthread_attr_init*( ) and *pthread_mutex_init*( ).

A *process-shared* attribute has been defined for condition variables for the same reason it has been defined for mutexes.

32820 **FUTURE DIRECTIONS**

32821 None.

32822 **SEE ALSO**

32823 *pthread_cond_destroy*( ), *pthread_condattr_getpshared*( ), *pthread_create*( ), *pthread_mutex_destroy*( ), |
32824 the Base Definitions volume of IEEE Std. 1003.1-200x, **<pthread.h>** |

32825 **CHANGE HISTORY**

32826 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32827 **Issue 6**

32828 The *pthread_condattr_destroy*( ) and *pthread_condattr_init*( ) functions are marked as part of the |
32829 Threads option. |

NAME

pthread_condattr_getclock, pthread_condattr_setclock — get and set the clock selection condition variable attribute

SYNOPSIS

THR CS

```
#include <pthread.h>

int pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
    clockid_t *restrict clock_id);
int pthread_condattr_setclock(pthread_condattr_t *attr,
    clockid_t clock_id);
```

DESCRIPTION

The *pthread_condattr_getclock*( ) function obtains the value of the clock attribute from the attributes object referenced by *attr*. The *pthread_condattr_setclock*( ) function is used to set the clock attribute in an initialized attributes object referenced by *attr*. If *pthread_condattr_setclock*( ) is called with a *clock_id* argument that refers to a CPU-time clock, the call shall fail.

The clock attribute is the clock ID of the clock that shall be used to measure the timeout service of *pthread_cond_timedwait*( ). The default value of the clock attribute shall refer to the system clock.

RETURN VALUE

If successful, the *pthread_condattr_getclock*( ) function shall return zero and store the value of the clock attribute of *attr* into the object referenced by the *clock_id* argument. Otherwise, an error number shall be returned to indicate the error.

If successful, the *pthread_condattr_setclock*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EINVAL] The value specified by *attr* is invalid.

The *pthread_condattr_setclock*( ) function may fail if:

[EINVAL] The value specified by *clock_id* does not refer to a known clock, or is a CPU-time clock.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_init( ), *pthread_cond_timedwait*( ), *pthread_condattr_destroy*( ), *pthread_condattr_getpshared*( ) (on page 1553),1 *pthread_condattr_init*( ), *pthread_condattr_setpshared*( ) (on page 1557),1 *pthread_create*( ), *pthread_mutex_init*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<pthread.h>**

32874 **CHANGE HISTORY**

32875 First released in Issue 6. Derived from IEEE Std. 1003.1j-2000. |

32876 **NAME**

32877 pthread_condattr_getpshared, pthread_condattr_setpshared — get and set the process-shared
32878 condition variable attributes

32879 **SYNOPSIS**

32880 THR TSH
```
#include <pthread.h>

int pthread_condattr_getpshared(const pthread_condattr_t *restrict attr,
    int *restrict pshared);
int pthread_condattr_setpshared(pthread_condattr_t *attr,
    int pshared);
```
32885

32886 **DESCRIPTION**

32887 The *pthread_condattr_getpshared*( ) function obtains the value of the *process-shared* attribute from
32888 the attributes object referenced by *attr*. The *pthread_condattr_setpshared*( ) function is used to set
32889 the *process-shared* attribute in an initialized attributes object referenced by *attr*.

32890 The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a condition
32891 variable to be operated upon by any thread that has access to the memory where the condition
32892 variable is allocated, even if the condition variable is allocated in memory that is shared by
32893 multiple processes. If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the
32894 condition variable is only operated upon by threads created within the same process as the
32895 thread that initialized the condition variable; if threads of differing processes attempt to operate
32896 on such a condition variable, the behavior is undefined. The default value of the attribute is
32897 PTHREAD_PROCESS_PRIVATE.

32898 Additional attributes, their default values, and the names of the associated functions to get and
32899 set those attribute values are implementation-defined.

32900 **RETURN VALUE**

32901 If successful, the *pthread_condattr_setpshared*( ) function shall return zero; otherwise, an error
32902 number shall be returned to indicate the error.

32903 If successful, the *pthread_condattr_getpshared*( ) function shall return zero and store the value of
32904 the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise,
32905 an error number shall be returned to indicate the error.

32906 **ERRORS**

32907 The *pthread_condattr_getpshared*( ) and *pthread_condattr_setpshared*( ) functions may fail if:

32908 [EINVAL] The value specified by *attr* is invalid.

32909 The *pthread_condattr_setpshared*( ) function may fail if:

32910 [EINVAL] The new value specified for the attribute is outside the range of legal values
32911 for that attribute.

32912 These functions shall not return an error code of [EINTR].

32913 **EXAMPLES**

32914 None.

32915 **APPLICATION USAGE**

32916 None.

32917 **RATIONALE**

32918 None.

32919 **FUTURE DIRECTIONS**

32920 None.

32921 **SEE ALSO**

32922 *pthread_create*( ), *pthread_cond_destroy*( ), *pthread_condattr_destroy*( ), *pthread_mutex_destroy*( ), the |
32923 Base Definitions volume of IEEE Std. 1003.1-200x, **<pthread.h>** |

32924 **CHANGE HISTORY**

32925 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32926 **Issue 6**

32927 The *pthread_condattr_getpshared*( ) and *pthread_condattr_setpshared*( ) functions are marked as part |
32928 of the Threads and Thread Process-Shared Synchronization options. |

32929 The **restrict** keyword is added to the *pthread_condattr_getpshared*( ) prototype for alignment with |
32930 the ISO / IEC 9899: 1999 standard. |

32931 **NAME**

32932 pthread_condattr_init — initialize condition variable attributes object

32933 **SYNOPSIS**

32934 THR
```
#include <pthread.h>
```

32935
```
int pthread_condattr_init(pthread_condattr_t *attr);
```

32936

32937 **DESCRIPTION**

32938 Refer to *pthread_condattr_destroy*( ).

32939 **NAME**

32940 pthread_condattr_setclock — set the clock selection condition variable attribute

32941 **SYNOPSIS**

32942 THR CS
```
#include <pthread.h>

int pthread_condattr_setclock(pthread_condattr_t *attr,
    clockid_t clock_id);
```

32943 |
32944 |
32945 |

32946 **DESCRIPTION**

32947 Refer to *pthread_condattr_getclock*( ).

32948 **NAME**

32949 pthread_condattr_setpshared — set the process-shared condition variable attributes

32950 **SYNOPSIS**

32951 THR TSH
```
#include <pthread.h>

int pthread_condattr_setpshared(pthread_condattr_t *attr,
    int pshared);
```
32952

32953

32954

32955 **DESCRIPTION**

32956 Refer to *pthread_condattr_getpshared*( ).

32957 **NAME**

32958 pthread_create — thread creation

32959 **SYNOPSIS**

32960 THR
```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
    const pthread_attr_t *restrict attr,
    void *(*start_routine)(void*), void *arg);
```
32964

32965 **DESCRIPTION**

32966 The *pthread_create*( ) function is used to create a new thread, with attributes specified by *attr*,
32967 within a process. If *attr* is NULL, the default attributes are used. If the attributes specified by *attr*
32968 are modified later, the thread's attributes are not affected. Upon successful completion,
32969 *pthread_create*( ) shall store the ID of the created thread in the location referenced by *thread*.

32970 The thread is created executing *start_routine* with *arg* as its sole argument. If the *start_routine*
32971 returns, the effect shall be as if there was an implicit call to *pthread_exit*( ) using the return value
32972 of *start_routine* as the exit status. Note that the thread in which *main*( ) was originally invoked
32973 differs from this. When it returns from *main*( ), the effect shall be as if there was an implicit call
32974 to *exit*( ) using the return value of *main*( ) as the exit status.

32975 The signal state of the new thread shall be initialized as follows:

32976 - The signal mask shall be inherited from the creating thread.

32977 - The set of signals pending for the new thread shall be empty.

32978 If *pthread_create*( ) fails, no new thread is created and the contents of the location referenced by
32979 *thread* are undefined.

32980 TCT If _POSIX_THREAD_CPUTIME is defined, the new thread shall have a CPU-time clock
32981 accessible, and the initial value of this clock shall be set to zero.

32982 **RETURN VALUE**

32983 If successful, the *pthread_create*( ) function shall return zero; otherwise, an error number shall be
32984 returned to indicate the error.

32985 **ERRORS**

32986 The *pthread_create*( ) function shall fail if:

32987 [EAGAIN] The system lacked the necessary resources to create another thread, or the
32988 system-imposed limit on the total number of threads in a process
32989 PTHREAD_THREADS_MAX would be exceeded.

32990 [EINVAL] The value specified by *attr* is invalid.

32991 [EPERM] The caller does not have appropriate permission to set the required
32992 scheduling parameters or scheduling policy.

32993 The *pthread_create*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

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 to begin ''activation''. The second approach requires no such condition variable or extra synchronization. In either approach, a separate Ada task control block would need to be created when the task object is created to hold rendezvous queues, and so on.

An extension of the preceding model would be to allow the state of the thread to be modified between the create and start. This would allow the thread attributes object to be eliminated. This has been rejected because:

- All state in the thread attributes object has to be able to be set for the thread. This would require the definition of functions to modify thread attributes. There would be no reduction in the number of function calls required to set up the thread. In fact, for an application that creates all threads using identical attributes, the number of function calls required to set up the threads would be dramatically increased. Use of a thread attributes object permits the application to make one set of attribute setting function calls. Otherwise, the set of attribute setting function calls needs to be made for each thread creation.
- Depending on the implementation architecture, functions to set thread state would require kernel calls, or for other implementation reasons would not be able to be implemented as macros, thereby increasing the cost of thread creation.
- The ability for applications to segregate threads by class would be lost.

Another suggested alternative uses a model similar to that for process creation, such as ''thread fork''. The fork semantics would provide more flexibility and the ''create'' function can be implemented simply by doing a thread fork followed immediately by a call to the desired ''start routine'' for the thread. This alternative has these problems:

- 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.

- 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.

FUTURE DIRECTIONS

None.

SEE ALSO

fork( ), *pthread_exit*( ), *pthread_join*( ), 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_create*( ) function is marked as part of the Threads option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EPERM] mandatory error condition is added.

The thread CPU-time clock semantics are added for alignment with IEEE Std. 1003.1d-1999.

The **restrict** keyword is added to the *pthread_create*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

pthread_detach — detach a thread

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

DESCRIPTION

The *pthread_detach*( ) function is used to indicate to the implementation that storage for the thread *thread* can be reclaimed when that thread terminates. If *thread* has not terminated, *pthread_detach*( ) shall not cause it to terminate. The effect of multiple *pthread_detach*( ) calls on the same target thread is unspecified.

RETURN VALUE

If the call succeeds, *pthread_detach*( ) shall return 0; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_detach*( ) function shall fail if:

[EINVAL] The implementation has detected that the value specified by *thread* does not refer to a joinable thread.

[ESRCH] No thread could be found corresponding to that specified by the given thread ID.

The *pthread_detach*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *pthread_join*( ) or *pthread_detach*( ) functions should eventually be called for every thread that is created so that storage associated with the thread may be reclaimed.

It has been suggested that a ‘‘detach’’ function is not necessary; the *detachstate* thread creation attribute is sufficient, since a thread need never be dynamically detached. However, need arises in at least two cases:

1. In a cancelation handler for a *pthread_join*( ) it is nearly essential to have a *pthread_detach*( ) function in order to detach the thread on which *pthread_join*( ) was waiting. Without it, it would be necessary to have the handler do another *pthread_join*( ) to attempt to detach the thread, which would both delay the cancelation processing for an unbounded period and introduce a new call to *pthread_join*( ), which might itself need a cancelation handler. A dynamic detach is nearly essential in this case.
2. In order to detach the ‘‘initial thread’’ (as may be desirable in processes that set up server threads).

FUTURE DIRECTIONS

None.

33099 **SEE ALSO**

33100 *pthread_join*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<pthread.h>**

33101 **CHANGE HISTORY**

33102 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33103 **Issue 6**

33104 The *pthread_detach*( ) function is marked as part of the Threads option.

33105 **NAME**

33106 pthread_equal — compare thread IDs

33107 **SYNOPSIS**

33108 THR
```
#include <pthread.h>
```
33109
```
int pthread_equal(pthread_t t1, pthread_t t2);
```
33110

33111 **DESCRIPTION**

33112 This function compares the thread IDs *t1* and *t2*.

33113 **RETURN VALUE**

33114 The *pthread_equal*( ) function shall return a non-zero value if *t1* and *t2* are equal; otherwise, zero
33115 shall be returned.

33116 If either *t1* or *t2* are not valid thread IDs, the behavior is undefined.

33117 **ERRORS**

33118 No errors are defined.

33119 The *pthread_equal*( ) function shall not return an error code of [EINTR]. |

33120 **EXAMPLES**

33121 None.

33122 **APPLICATION USAGE**

33123 None.

33124 **RATIONALE**

33125 Implementations may choose to define a thread ID as a structure. This allows additional
33126 flexibility and robustness over using an **int**. For example, a thread ID could include a sequence
33127 number that allows detection of ‘‘dangling IDs’’ (copies of a thread ID that has been detached).
33128 Because the C language does not support comparison on structure types, the *pthread_equal*( )
33129 function is provided to compare thread IDs.

33130 **FUTURE DIRECTIONS**

33131 None.

33132 **SEE ALSO**

33133 *pthread_create*( ), *pthread_self*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
33134 **<pthread.h>**

33135 **CHANGE HISTORY**

33136 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33137 **Issue 6**

33138 The *pthread_equal*( ) function is marked as part of the Threads option. |

33139 **NAME**

33140 pthread_exit — thread termination

33141 **SYNOPSIS**

33142 THR
```
#include <pthread.h>
```

33143
```
void pthread_exit(void *value_ptr);
```

33144

33145 **DESCRIPTION**

33146 The *pthread_exit*( ) function shall terminate the calling thread and make the value *value_ptr*
33147 available to any successful join with the terminating thread. Any cancelation cleanup handlers
33148 that have been pushed and not yet popped shall be popped in the reverse order that they were
33149 pushed and then executed. After all cancelation cleanup handlers have been executed, if the
33150 thread has any thread-specific data, appropriate destructor functions shall be called in an
33151 unspecified order. Thread termination does not release any application visible process resources,
33152 including, but not limited to, mutexes and file descriptors, nor does it perform any process-level
33153 cleanup actions, including, but not limited to, calling any *atexit*( ) routines that may exist.

33154 An implicit call to *pthread_exit*( ) is made when a thread other than the thread in which *main*( )
33155 was first invoked returns from the start routine that was used to create it. The function's return
33156 value serves as the thread's exit status.

33157 The behavior of *pthread_exit*( ) is undefined if called from a cancelation cleanup handler or
33158 destructor function that was invoked as a result of either an implicit or explicit call to
33159 *pthread_exit*( ).

33160 After a thread has terminated, the result of access to local (auto) variables of the thread is
33161 undefined. Thus, references to local variables of the exiting thread should not be used for the
33162 *pthread_exit*( ) *value_ptr* parameter value.

33163 The process shall exit with an exit status of 0 after the last thread has been terminated. The
33164 behavior shall be as if the implementation called *exit*( ) with a zero argument at thread
33165 termination time.

33166 **RETURN VALUE**

33167 The *pthread_exit*( ) function cannot return to its caller.

33168 **ERRORS**

33169 No errors are defined.

33170 The *pthread_exit*( ) function shall not return an error code of [EINTR]. |

33171 **EXAMPLES**

33172 None.

33173 **APPLICATION USAGE**

33174 None.

33175 **RATIONALE**

33176 The normal mechanism by which a thread terminates is to return from the routine that was
33177 specified in the *pthread_create*( ) call that started it. The *pthread_exit*( ) function provides the
33178 capability for a thread to terminate without requiring a return from the start routine of that
33179 thread, thereby providing a function analogous to *exit*( ).

33180 Regardless of the method of thread termination, any cancelation cleanup handlers that have
33181 been pushed and not yet popped are executed, and the destructors for any existing thread-
33182 specific data are executed. This volume of IEEE Std. 1003.1-200x requires that cancelation
33183 cleanup handlers be popped and called in order. After all cancelation cleanup handlers have

been executed, thread-specific data destructors are called, in an unspecified order, for each item of thread-specific data that exists in the thread. This ordering is necessary because cancelation cleanup handlers may rely on thread-specific data.

As the meaning of the status is determined by the application (except when the thread has been canceled, in which case it is PTHREAD_CANCELED), the implementation has no idea what an illegal status value is, which is why no address error checking is done.

FUTURE DIRECTIONS

None.

SEE ALSO

exit( ), *pthread_create*( ), *pthread_join*( ), 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_exit*( ) function is marked as part of the Threads option.

33199 **NAME**

33200 pthread_getconcurrency, pthread_setconcurrency — get or set level of concurrency

33201 **SYNOPSIS**

33202 XSI
```
#include <pthread.h>

int pthread_getconcurrency(void);
int pthread_setconcurrency(int new_level);
```
33205

33206 **DESCRIPTION**

33207 Unbound threads in a process may or may not be required to be simultaneously active. By
33208 default, the threads implementation ensures that a sufficient number of threads are active so that
33209 the process can continue to make progress. While this conserves system resources, it may not
33210 produce the most effective level of concurrency.

33211 The *pthread_setconcurrency*( ) function allows an application to inform the threads
33212 implementation of its desired concurrency level, *new_level*. The actual level of concurrency
33213 provided by the implementation as a result of this function call is unspecified.

33214 If *new_level* is zero, it causes the implementation to maintain the concurrency level at its
33215 discretion as if *pthread_setconcurrency*( ) had never been called.

33216 The *pthread_getconcurrency*( ) function shall return the value set by a previous call to the
33217 *pthread_setconcurrency*( ) function. If the *pthread_setconcurrency*( ) function was not previously
33218 called, this function shall return zero to indicate that the implementation is maintaining the
33219 concurrency level.

33220 When an application calls *pthread_setconcurrency*( ) it is informing the implementation of its
33221 desired concurrency level. The implementation uses this as a hint, not a requirement.

33222 If an implementation does not support multiplexing of user threads on top of several kernel-
33223 scheduled entities, the *pthread_setconcurrency*( ) and *pthread_getconcurrency*( ) functions are
33224 provided for source code compatibility but they have no effect when called. To maintain the
33225 function semantics, the *new_level* parameter is saved when *pthread_setconcurrency*( ) is called so
33226 that a subsequent call to *pthread_getconcurrency*( ) returns the same value.

33227 **RETURN VALUE**

33228 If successful, the *pthread_setconcurrency*( ) function shall return zero; otherwise, an error number
33229 shall be returned to indicate the error.

33230 The *pthread_getconcurrency*( ) function shall always return the concurrency level set by a previous
33231 call to *pthread_setconcurrency*( ). If the *pthread_setconcurrency*( ) function has never been called,
33232 *pthread_getconcurrency*( ) shall return zero.

33233 **ERRORS**

33234 The *pthread_setconcurrency*( ) function shall fail if:

33235 [EINVAL] The value specified by *new_level* is negative.

33236 [EAGAIN] The value specific by *new_level* would cause a system resource to be exceeded.

33237 These functions shall not return an error code of [EINTR].

33238 **EXAMPLES**

33239 None.

33240 **APPLICATION USAGE**

33241 Use of these functions changes the state of the underlying concurrency upon which the
33242 application depends. Library developers are advised to not use the *pthread_getconcurrency*( ) and
33243 *pthread_setconcurrency*( ) functions since their use may conflict with an applications use of these
33244 functions.

33245 **RATIONALE**

33246 None.

33247 **FUTURE DIRECTIONS**

33248 None.

33249 **SEE ALSO**

33250 The Base Definitions volume of IEEE Std. 1003.1-200x, **<pthread.h>** |

33251 **CHANGE HISTORY**

33252 First released in Issue 5.

33253 **NAME**

33254 pthread_getcpuclockid — access a thread CPU-time clock (**REALTIME**)

33255 **SYNOPSIS**

33256 TCT
```
#include <pthread.h>
#include <time.h>

int pthread_getcpuclockid(pthread_t thread_id, clockid_t *clock_id);
```
33257

33258

33259

33260 **DESCRIPTION**

33261 The *pthread_getcpuclockid*( ) function shall return in *clock_id* the clock ID of the CPU-time clock of
33262 the thread specified by *thread_id*, if the thread specified by *thread_id* exists.

33263 **RETURN VALUE**

33264 Upon successful completion, *pthread_getcpuclockid*( ) shall return zero; otherwise, an error
33265 number shall be returned to indicate the error.

33266 **ERRORS**

33267 The *pthread_getcpuclockid*( ) function may fail if:

33268 [ESRCH] The value specified by *thread_id* does not refer to an existing thread.

33269 **EXAMPLES**

33270 None.

33271 **APPLICATION USAGE**

33272 The *pthread_getcpuclockid*( ) function is part of the Thread CPU-Time Clocks option and need not |
33273 be provided on all implementations. |

33274 **RATIONALE**

33275 None.

33276 **FUTURE DIRECTIONS**

33277 None.

33278 **SEE ALSO**

33279 *clock_getcpuclockid*( ), *clock_getres*( ), *timer_create*( ), the Base Definitions volume of |
33280 IEEE Std. 1003.1-200x, **<pthread.h>**, **<time.h>** |

33281 **CHANGE HISTORY**

33282 First released in Issue 6. Derived from IEEE Std. 1003.1d-1999.

33283 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

33284 **NAME**

33285 pthread_getschedparam, pthread_setschedparam — dynamic thread scheduling parameters
33286 access (**REALTIME THREADS**)

33287 **SYNOPSIS**

33288 TPS
```
#include <pthread.h>

int pthread_getschedparam(pthread_t thread, int *restrict policy,
    struct sched_param *restrict param);
int pthread_setschedparam(pthread_t thread, int policy,
    const struct sched_param *param);
```
33293

33294 **DESCRIPTION**

33295 The *pthread_getschedparam*( ) and *pthread_setschedparam*( ) functions allow the scheduling policy
33296 and scheduling parameters of individual threads within a multi-threaded process to be retrieved
33297 and set. For SCHED_FIFO and SCHED_RR, the only required member of the **sched_param**
33298 structure is the priority *sched_priority*. For SCHED_OTHER, the affected scheduling parameters
33299 are implementation-defined.

33300 The *pthread_getschedparam*( ) function shall retrieve the scheduling policy and scheduling
33301 parameters for the thread whose thread ID is given by *thread* and shall store those values in
33302 *policy* and *param*, respectively. The priority value returned from *pthread_getschedparam*( ) shall be
33303 the value specified by the most recent *pthread_setschedparam*( ) or *pthread_create*( ) call affecting
33304 the target thread. It shall not reflect any temporary adjustments to its priority as a result of any
33305 priority inheritance or ceiling functions. The *pthread_setschedparam*( ) function sets the scheduling
33306 policy and associated scheduling parameters for the thread whose thread ID is given by *thread* to
33307 the policy and associated parameters provided in *policy* and *param*, respectively.

33308 The *policy* parameter may have the value SCHED_OTHER, SCHED_FIFO, or SCHED_RR. The
33309 scheduling parameters for the SCHED_OTHER policy are implementation-defined. The
33310 SCHED_FIFO and SCHED_RR policies shall have a single scheduling parameter, *priority*.

33311 TSP If _POSIX_THREAD_SPORADIC_SERVER is defined, then the *policy* argument may have the
33312 value SCHED_SPORADIC, with the exception for the *pthread_setschedparam*( ) function that if the
33313 scheduling policy was not SCHED_SPORADIC at the time of the call, it is implementation-
33314 defined whether the function is supported; in other words, the implementation need not allow
33315 the application to dynamically change the scheduling policy to SCHED_SPORADIC. The
33316 sporadic server scheduling policy has the associated parameters *sched_ss_low_priority*,
33317 *sched_ss_repl_period*, *sched_ss_init_budget*, *sched_priority*, and *sched_ss_max_repl*. The specified
33318 *sched_ss_repl_period* shall be greater than or equal to the specified *sched_ss_init_budget* for the
33319 function to succeed; if it is not, then the function shall fail. The value of *sched_ss_max_repl* shall
33320 be within the inclusive range [1,{SS_REPL_MAX}] for the function to succeed; if not, the function
33321 shall fail.

33322 If the *pthread_setschedparam*( ) function fails, no scheduling parameters are changed for the target
33323 thread.

33324 **RETURN VALUE**

33325 If successful, the *pthread_getschedparam*( ) and *pthread_setschedparam*( ) functions shall return zero;
33326 otherwise, an error number shall be returned to indicate the error.

33327 **ERRORS**

33328 The *pthread_getschedparam*( ) function may fail if:

33329 [ESRCH] The value specified by *thread* does not refer to a 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 a existing thread.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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 Thread Execution Scheduling option.

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 corrigenda item U026/2 has been 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 corrigenda item U047/1 has been applied.

33367 **NAME**

33368 pthread_getspecific, pthread_setspecific — thread-specific data management

33369 **SYNOPSIS**

33370 THR
```
#include <pthread.h>

void *pthread_getspecific(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void *value);
```
33373

33374 **DESCRIPTION**

33375 The *pthread_getspecific*( ) function shall return the value currently bound to the specified *key* on
33376 behalf of the calling thread.

33377 The *pthread_setspecific*( ) function shall associate a thread-specific *value* with a *key* obtained via a
33378 previous call to *pthread_key_create*( ). Different threads may bind different values to the same
33379 key. These values are typically pointers to blocks of dynamically allocated memory that have
33380 been reserved for use by the calling thread.

33381 The effect of calling *pthread_getspecific*( ) or *pthread_setspecific*( ) with a *key* value not obtained
33382 from *pthread_key_create*( ) or after *key* has been deleted with *pthread_key_delete*( ) is undefined.

33383 Both *pthread_getspecific*( ) and *pthread_setspecific*( ) may be called from a thread-specific data |
33384 destructor function. A call to *pthread_getspecific*( ) for the thread-specific data key being |
33385 destroyed shall return the value NULL, unless the value is changed (after the destructor starts) |
33386 by a call to *pthread_setspecific.*( ) Calling *pthread_setspecific*( ) thread-specific data destructor |
33387 routine may result either in lost storage (after at least PTHREAD_DESTRUCTOR_ITERATIONS) |
33388 or an infinite loop. |

33389 Both functions may be implemented as macros.

33390 **RETURN VALUE**

33391 The *pthread_getspecific*( ) function shall return the thread-specific data value associated with the
33392 given *key*. If no thread-specific data value is associated with *key*, then the value NULL shall be
33393 returned.

33394 If successful, the *pthread_setspecific*( ) function shall return zero; otherwise, an error number shall
33395 be returned to indicate the error.

33396 **ERRORS**

33397 No errors are returned from *pthread_getspecific*( ).

33398 The *pthread_setspecific*( ) function shall fail if:

33399 [ENOMEM] Insufficient memory exists to associate the value with the key. |

33400 The *pthread_setspecific*( ) function may fail if:

33401 [EINVAL] The key value is invalid. |

33402 These functions shall not return an error code of [EINTR]. |

33403 **EXAMPLES**

33404 None.

33405 **APPLICATION USAGE**

33406 None.

33407 **RATIONALE**

33408 Performance and ease-of-use of *pthread_getspecific*( ) is critical for functions that rely on
33409 maintaining state in thread-specific data. Since no errors are required to be detected by it, and
33410 since the only error that could be detected is the use of an invalid key, the function to
33411 *pthread_getspecific*( ) has been designed to favor speed and simplicity over error reporting.

33412 **FUTURE DIRECTIONS**

33413 None.

33414 **SEE ALSO**

33415 *pthread_key_create*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<pthread.h>** |

33416 **CHANGE HISTORY**

33417 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33418 **Issue 6**

33419 The *pthread_getspecific*( ) and *pthread_setspecific*( ) functions are marked as part of the Threads |
33420 option. |

NAME

pthread_join — wait for thread termination

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

DESCRIPTION

The *pthread_join*( ) function shall suspend execution of the calling thread until the target *thread* terminates, unless the target *thread* has already terminated. On return from a successful *pthread_join*( ) call with a non-NULL *value_ptr* argument, the value passed to *pthread_exit*( ) by the terminating thread shall be made available in the location referenced by *value_ptr*. When a *pthread_join*( ) returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to *pthread_join*( ) specifying the same target thread are undefined. If the thread calling *pthread_join*( ) is canceled, then the target thread shall not be detached.

It is unspecified whether a thread that has exited but remains unjoined counts against _PTHREAD_THREADS_MAX.

RETURN VALUE

If successful, the *pthread_join*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_join*( ) function shall fail if:

[EINVAL] The implementation has detected that the value specified by *thread* does not refer to a joinable thread. |

[ESRCH] No thread could be found corresponding to that specified by the given thread ID. |

The *pthread_join*( ) function may fail if:

[EDEADLK] A deadlock was detected or the value of *thread* specifies the calling thread.

The *pthread_join*( ) function shall not return an error code of [EINTR]. |

EXAMPLES

An example of thread creation and deletion follows:

```
typedef struct {
    int *ar;
    long n;
} subarray;

void *
incer(void *arg)
{
    long i;

    for (i = 0; i < ((subarray *)arg)->n; i++)
        ((subarray *)arg)->ar[i]++;
}

main()
{
    int        ar[1000000];
```

```
        pthread_t  th1, th2;
        subarray   sb1, sb2;

        sb1.ar = &ar[0];
        sb1.n  = 500000;
        (void) pthread_create(&th1, NULL, incer, &sb1);

        sb2.ar = &ar[500000];
        sb2.n  = 500000;
        (void) pthread_create(&th2, NULL, incer, &sb2);

        (void) pthread_join(th1, NULL);
        (void) pthread_join(th2, NULL);
    }
```

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 cancelation 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.
- Cancelation 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 cancelation 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, **<pthread.h>**

33508 **CHANGE HISTORY**

33509 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33510 **Issue 6**

33511 The *pthread_join*( ) function is marked as part of the Threads option.

33512 **NAME**

33513 pthread_key_create — thread-specific data key creation

33514 **SYNOPSIS**

33515 THR
```
#include <pthread.h>

int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));
```
33516

33517

33518 **DESCRIPTION**

33519 The *pthread_key_create*( ) function shall create a thread-specific data key visible to all threads in
33520 the process. Key values provided by *pthread_key_create*( ) are opaque objects used to locate
33521 thread-specific data. Although the same key value may be used by different threads, the values
33522 bound to the key by *pthread_setspecific*( ) are maintained on a per-thread basis and persist for the
33523 life of the calling thread.

33524 Upon key creation, the value NULL shall be associated with the new key in all active threads.
33525 Upon thread creation, the value NULL shall be associated with all defined keys in the new
33526 thread.

33527 An optional destructor function may be associated with each key value. At thread exit, if a key
33528 value has a non-NULL destructor pointer, and the thread has a non-NULL value associated with |
33529 that key, the value of the key is set to NULL, and then the function pointed to is called with the |
33530 previously associated value as its sole argument. The order of destructor calls is unspecified if |
33531 more than one destructor exists for a thread when it exits. |

33532 If, after all the destructors have been called for all non-NULL values with associated destructors,
33533 there are still some non-NULL values with associated destructors, then the process is repeated.
33534 If, after at least {PTHREAD_DESTRUCTOR_ITERATIONS} iterations of destructor calls for
33535 outstanding non-NULL values, there are still some non-NULL values with associated
33536 destructors, implementations may stop calling destructors, or they may continue calling
33537 destructors until no non-NULL values with associated destructors exist, even though this might
33538 result in an infinite loop.

33539 **RETURN VALUE**

33540 If successful, the *pthread_key_create*( ) function shall store the newly created key value at **key* and
33541 shall return zero. Otherwise, an error number shall be returned to indicate the error.

33542 **ERRORS**

33543 The *pthread_key_create*( ) function shall fail if:

33544 [EAGAIN] The system lacked the necessary resources to create another thread-specific |
33545 data key, or the system-imposed limit on the total number of keys per process
33546 PTHREAD_KEYS_MAX has been exceeded.

33547 [ENOMEM] Insufficient memory exists to create the key. |

33548 The *pthread_key_create*( ) function shall not return an error code of [EINTR]. |

EXAMPLES

The following example demonstrates a function that initializes a thread-specific data key when it is first called, and associates a thread-specific object with each calling thread, initializing this object when necessary.

```
static pthread_key_t key;
static pthread_once_t key_once = PTHREAD_ONCE_INIT;

static void
make_key()
{
    (void) pthread_key_create(&key, NULL);
}

func()
{
    void *ptr;

    (void) pthread_once(&key_once, make_key);
    if ((ptr = pthread_getspecific(key)) == NULL) {
        ptr = malloc(OBJECT_SIZE);
        ...
        (void) pthread_setspecific(key, ptr);
    }
    ...
}
```

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.

```
static pthread_key_t key;

func()
{
    void *ptr;

   /* KEY NOT INITIALIZED!!!  THIS WON'T WORK!!! */
    if ((ptr = pthread_getspecific(key)) == NULL &&
        pthread_setspecific(key, NULL) != 0) {
        pthread_key_create(&key, NULL);
        ...
    }
}
```

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 cancelation cleanup handlers cannot be used for this purpose because thread-specific data may persist outside the lexical scope in which the cancelation 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:

```
pthread_getspecific(key, &old);
new = allocate();
destructor(old);
pthread_setspecific(key, new);
```

Note: The above example could leak storage if run with asynchronous cancelation enabled. No such problems occur in the default cancelation state if no cancelation 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 it 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.

33631 **SEE ALSO**

33632 *pthread_getspecific*( ), *pthread_key_delete*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
33633 **<pthread.h>**

33634 **CHANGE HISTORY**

33635 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33636 **Issue 6**

33637 The *pthread_key_create*( ) function is marked as part of the Threads option. |

33638 IEEE PASC Interpretation 1003.1c #8 is applied, updating the DESCRIPTION. |

33639 **NAME**

33640 pthread_key_delete — thread-specific data key deletion

33641 **SYNOPSIS**

33642 THR
```
#include <pthread.h>
```

33643
```
int pthread_key_delete(pthread_key_t key);
```

33644

33645 **DESCRIPTION**

33646 The *pthread_key_delete*( ) function shall delete a thread-specific data key previously returned by
33647 *pthread_key_create*( ). The thread-specific data values associated with *key* need not be NULL at
33648 the time *pthread_key_delete*( ) is called. It is the responsibility of the application to free any
33649 application storage or perform any cleanup actions for data structures related to the deleted key
33650 or associated thread-specific data in any threads; this cleanup can be done either before or after
33651 *pthread_key_delete*( ) is called. Any attempt to use *key* following the call to *pthread_key_delete*( )
33652 results in undefined behavior.

33653 The *pthread_key_delete*( ) function shall be callable from within destructor functions. No
33654 destructor functions shall be invoked by *pthread_key_delete*( ). Any destructor function that may
33655 have been associated with *key* shall no longer be called upon thread exit.

33656 **RETURN VALUE**

33657 If successful, the *pthread_key_delete*( ) function shall return zero; otherwise, an error number shall
33658 be returned to indicate the error.

33659 **ERRORS**

33660 The *pthread_key_delete*( ) function may fail if:

33661 [EINVAL] The *key* value is invalid. |

33662 The *pthread_key_delete*( ) function shall not return an error code of [EINTR]. |

33663 **EXAMPLES**

33664 None.

33665 **APPLICATION USAGE**

33666 None.

33667 **RATIONALE**

33668 A thread-specific data key deletion function has been included in order to allow the resources
33669 associated with an unused thread-specific data key to be freed. Unused thread-specific data keys
33670 can arise, among other scenarios, when a dynamically loaded module that allocated a key is
33671 unloaded.

33672 Portable applications are responsible for performing any cleanup actions needed for data
33673 structures associated with the key to be deleted, including data referenced by thread-specific
33674 data values. No such cleanup is done by *pthread_key_delete*( ). In particular, destructor functions
33675 are not called. There are several reasons for this division of responsibility:

33676 1. The associated destructor functions used to free thread-specific data at thread exit time are
33677 only guaranteed to work correctly when called in the thread that allocated the thread-
33678 specific data. (Destructors themselves may utilize thread-specific data.) Thus, they cannot
33679 be used to free thread-specific data in other threads at key deletion time. Attempting to
33680 have them called by other threads at key deletion time would require other threads to be
33681 asynchronously interrupted. But since interrupted threads could be in an arbitrary state,
33682 including holding locks necessary for the destructor to run, this approach would fail. In
33683 general, there is no safe mechanism whereby an implementation could free thread-specific
33684 data at key deletion time.

2. Even if there were a means of safely freeing thread-specific data associated with keys to be deleted, doing so would require that implementations be able to enumerate the threads with non-NULL data and potentially keep them from creating more thread-specific data while the key deletion is occurring. This special case could cause extra synchronization in the normal case, which would otherwise be unnecessary.

For an application to know that it is safe to delete a key, it has to know that all the threads that might potentially ever use the key do not attempt to use it again. For example, it could know this if all the client threads have called a cleanup procedure declaring that they are through with the module that is being shut down, perhaps by zero'ing a reference count.

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_key_delete*( ) function is marked as part of the Threads option.

33702 **NAME**

33703 pthread_kill — send a signal to a thread

33704 **SYNOPSIS**

33705 THR
```
#include <signal.h>
```

33706
```
int pthread_kill(pthread_t thread, int sig);
```

33707

33708 **DESCRIPTION**

33709 The *pthread_kill*( ) function is used to request that a signal be delivered to the specified thread.

33710 As in *kill*( ), if *sig* is zero, error checking is performed but no signal is actually sent.

33711 **RETURN VALUE**

33712 Upon successful completion, the function shall return a value of zero. Otherwise, the function
33713 shall return an error number. If the *pthread_kill*( ) function fails, no signal shall be sent.

33714 **ERRORS**

33715 The *pthread_kill*( ) function shall fail if:

33716 [ESRCH] No thread could be found corresponding to that specified by the given thread |
33717 ID.

33718 [EINVAL] The value of the *sig* argument is an invalid or unsupported signal number. |

33719 The *pthread_kill*( ) function shall not return an error code of [EINTR]. |

33720 **EXAMPLES**

33721 None.

33722 **APPLICATION USAGE**

33723 The *pthread_kill*( ) function provides a mechanism for asynchronously directing a signal at a
33724 thread in the calling process. This could be used, for example, by one thread to affect broadcast
33725 delivery of a signal to a set of threads.

33726 Note that *pthread_kill*( ) only causes the signal to be handled in the context of the given thread;
33727 the signal action (termination or stopping) affects the process as a whole.

33728 **RATIONALE**

33729 None.

33730 **FUTURE DIRECTIONS**

33731 None.

33732 **SEE ALSO**

33733 *kill*( ), *pthread_self*( ), *raise*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<signal.h>** |

33734 **CHANGE HISTORY**

33735 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33736 **Issue 6**

33737 The *pthread_kill*( ) function is marked as part of the Threads option. |

33738 The APPLICATION USAGE section is added.

NAME

pthread_mutex_destroy, pthread_mutex_init — destroy and initialize a mutex

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

DESCRIPTION

The *pthread_mutex_destroy*( ) function destroys the mutex object referenced by *mutex*; the mutex object becomes, in effect, uninitialized. An implementation may cause *pthread_mutex_destroy*( ) to set the object referenced by *mutex* to an invalid value. A destroyed mutex object can be re-initialized using *pthread_mutex_init*( ); the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behavior.

The *pthread_mutex_init*( ) function initializes the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

Only *mutex* itself may be used for performing synchronization. The result of referring to copies of *mutex* in calls to *pthread_mutex_lock*( ), *pthread_mutex_trylock*( ), *pthread_mutex_unlock*( ), and *pthread_mutex_destroy*( ) is undefined.

Attempting to initialize an already initialized mutex results in undefined behavior.

In cases where default mutex attributes are appropriate, the macro PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes that are statically allocated. The effect shall be equivalent to dynamic initialization by a call to *pthread_mutex_init*( ) with parameter *attr* specified as NULL, except that no error checks are performed.

RETURN VALUE

If successful, the *pthread_mutex_destroy*( ) and *pthread_mutex_init*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

The [EBUSY] and [EINVAL] error checks, if implemented, act as if they were performed immediately at the beginning of processing for the function and shall cause an error return prior to modifying the state of the mutex specified by *mutex*.

ERRORS

The *pthread_mutex_destroy*( ) function may fail if:

[EBUSY] The implementation has detected an attempt to destroy the object referenced by *mutex* while it is locked or referenced (for example, while being used in a *pthread_cond_timedwait*( ) or *pthread_cond_wait*( )) by another thread.

[EINVAL] The value specified by *mutex* is invalid.

The *pthread_mutex_init*( ) function shall fail if:

[EAGAIN] The system lacked the necessary resources (other than memory) to initialize another mutex.

33783 [ENOMEM] Insufficient memory exists to initialize the mutex. |

33784 [EPERM] The caller does not have the privilege to perform the operation. |

33785 The *pthread_mutex_init*( ) function may fail if:

33786 [EBUSY] The implementation has detected an attempt to re-initialize the object |
33787 referenced by *mutex*, a previously initialized, but not yet destroyed, mutex.

33788 [EINVAL] The value specified by *attr* is invalid. |

33789 These functions shall not return an error code of [EINTR]. |

33790 **EXAMPLES**

33791 None.

33792 **APPLICATION USAGE**

33793 None.

33794 **RATIONALE**

33795 **Alternate Implementations Possible**

33796 This volume of IEEE Std. 1003.1-200x supports several alternative implementations of mutexes.
33797 An implementation may store the lock directly in the object of type **pthread_mutex_t**.
33798 Alternatively, an implementation may store the lock in the heap and merely store a pointer,
33799 handle, or unique ID in the mutex object. Either implementation has advantages or may be
33800 required on certain hardware configurations. So that portable code can be written that is
33801 invariant to this choice, this volume of IEEE Std. 1003.1-200x does not define assignment or
33802 equality for this type, and it uses the term ‘‘initialize’’ to reinforce the (more restrictive) notion
33803 that the lock may actually reside in the mutex object itself.

33804 Note that this precludes an over-specification of the type of the mutex or condition variable and
33805 motivates the opacity of the type.

33806 An implementation is permitted, but not required, to have *pthread_mutex_destroy*( ) store an
33807 illegal value into the mutex. This may help detect erroneous programs that try to lock (or
33808 otherwise reference) a mutex that has already been destroyed.

33809 **Tradeoff Between Error Checks and Performance Supported**

33810 Many of the error checks were made optional in order to let implementations trade off
33811 performance *versus* degree of error checking according to the needs of their specific applications
33812 and execution environment. As a general rule, errors or conditions caused by the system (such as
33813 insufficient memory) always need to be reported, but errors due to an erroneously coded
33814 application (such as failing to provide adequate synchronization to prevent a mutex from being
33815 deleted while in use) are made optional.

33816 A wide range of implementations is thus made possible. For example, an implementation
33817 intended for application debugging may implement all of the error checks, but an
33818 implementation running a single, provably correct application under very tight performance
33819 constraints in an embedded computer might implement minimal checks. An implementation
33820 might even be provided in two versions, similar to the options that compilers provide: a full-
33821 checking, but slower version; and a limited-checking, but faster version. To forbid this
33822 optionality would be a disservice to users.

33823 By carefully limiting the use of ‘‘undefined behavior’’ only to things that an erroneous (badly
33824 coded) application might do, and by defining that resource-not-available errors are mandatory,
33825 this volume of IEEE Std. 1003.1-200x ensures that a fully-conforming application is portable

33826 across the full range of implementations, while not forcing all implementations to add overhead
33827 to check for numerous things that a correct program never does.

33828 **Why No Limits Defined**

33829 Defining symbols for the maximum number of mutexes and condition variables was considered
33830 but rejected because the number of these objects may change dynamically. Furthermore, many
33831 implementations place these objects into application memory; thus, there is no explicit
33832 maximum.

33833 **Static Initializers for Mutexes and Condition Variables**

33834 Providing for static initialization of statically allocated synchronization objects allows modules
33835 with private static synchronization variables to avoid runtime initialization tests and overhead.
33836 Furthermore, it simplifies the coding of self-initializing modules. Such modules are common in
33837 C libraries, where for various reasons the design calls for self-initialization instead of requiring
33838 an explicit module initialization function to be called. An example use of static initialization
33839 follows.

33840 Without static initialization, a self-initializing routine *foo*( ) might look as follows:

```
33841 static pthread_once_t foo_once = PTHREAD_ONCE_INIT;
33842 static pthread_mutex_t foo_mutex;

33843 void foo_init()
33844 {
33845     pthread_mutex_init(&foo_mutex, NULL);
33846 }

33847 void foo()
33848 {
33849     pthread_once(&foo_once, foo_init);
33850     pthread_mutex_lock(&foo_mutex);
33851    /* Do work. */
33852     pthread_mutex_unlock(&foo_mutex);
33853 }
```

33854 With static initialization, the same routine could be coded as follows:

```
33855 static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;

33856 void foo()
33857 {
33858     pthread_mutex_lock(&foo_mutex);
33859    /* Do work. */
33860     pthread_mutex_unlock(&foo_mutex);
33861 }
```

33862 Note that the static initialization both eliminates the need for the initialization test inside
33863 *pthread_once*( ) and the fetch of &*foo_mutex* to learn the address to be passed to
33864 *pthread_mutex_lock*( ) or *pthread_mutex_unlock*( ).

33865 Thus, the C code written to initialize static objects is simpler on all systems and is also faster on a
33866 large class of systems; those where the (entire) synchronization object can be stored in
33867 application memory.

33868 Yet the locking performance question is likely to be raised for machines that require mutexes to
33869 be allocated out of special memory. Such machines actually have to have mutexes and possibly

condition variables contain pointers to the actual hardware locks. For static initialization to work on such machines, *pthread_mutex_lock*( ) also has to test whether or not the pointer to the actual lock has been allocated. If it has not, *pthread_mutex_lock*( ) has to initialize it before use. The reservation of such resources can be made when the program is loaded, and hence return codes have not been added to mutex locking and condition variable waiting to indicate failure to complete initialization.

This runtime test in *pthread_mutex_lock*( ) would at first seem to be extra work; an extra test is required to see whether the pointer has been initialized. On most machines this would actually be implemented as a fetch of the pointer, testing the pointer against zero, and then using the pointer if it has already been initialized. While the test might seem to add extra work, the extra effort of testing a register is usually negligible since no extra memory references are actually done. As more and more machines provide caches, the real expenses are memory references, not instructions executed.

Alternatively, depending on the machine architecture, there are often ways to eliminate *all* overhead in the most important case: on the lock operations that occur *after* the lock has been initialized. This can be done by shifting more overhead to the less frequent operation: initialization. Since out-of-line mutex allocation also means that an address has to be dereferenced to find the actual lock, one technique that is widely applicable is to have static initialization store a bogus value for that address; in particular, an address that causes a machine fault to occur. When such a fault occurs upon the first attempt to lock such a mutex, validity checks can be done, and then the correct address for the actual lock can be filled in. Subsequent lock operations incur no extra overhead since they do not ‘‘fault’’. This is merely one technique that can be used to support static initialization, while not adversely affecting the performance of lock acquisition. No doubt there are other techniques that are highly machine-dependent.

The locking overhead for machines doing out-of-line mutex allocation is thus similar for modules being implicitly initialized, where it is improved for those doing mutex allocation entirely inline. The inline case is thus made much faster, and the out-of-line case is not significantly worse.

Besides the issue of locking performance for such machines, a concern is raised that it is possible that threads would serialize contending for initialization locks when attempting to finish initializing statically allocated mutexes. (Such finishing would typically involve taking an internal lock, allocating a structure, storing a pointer to the structure in the mutex, and releasing the internal lock.) First, many implementations would reduce such serialization by hashing on the mutex address. Second, such serialization can only occur a bounded number of times. In particular, it can happen at most as many times as there are statically allocated synchronization objects. Dynamically allocated objects would still be initialized via *pthread_mutex_init*( ) or *pthread_cond_init*( ).

Finally, if none of the above optimization techniques for out-of-line allocation yields sufficient performance for an application on some implementation, the application can avoid static initialization altogether by explicitly initializing all synchronization objects with the corresponding *pthread_*_init*( ) functions, which are supported by all implementations. An implementation can also document the tradeoffs and advise which initialization technique is more efficient for that particular implementation.

Destroying Mutexes

A mutex can be destroyed immediately after it is unlocked. For example, consider the following code:

```
struct obj {
pthread_mutex_t om;
    int refcnt;
    ...
};

obj_done(struct obj *op)
{
    pthread_mutex_lock(&op->om);
    if (--op->refcnt == 0) {
        pthread_mutex_unlock(&op->om);
(A)     pthread_mutex_destroy(&op->om);
(B)     free(op);
    } else
(C)     pthread_mutex_unlock(&op->om);
}
```

In this case *obj* is reference counted and *obj_done*( ) is called whenever a reference to the object is dropped. Implementations are required to allow an object to be destroyed and freed and potentially unmapped (for example, lines A and B) immediately after the object is unlocked (line C).

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_getprioceiling( ), *pthread_mutex_lock*( ), *pthread_mutex_timedlock*( ), *pthread_mutexattr_getpshared*( ), 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_mutex_destroy*( ) and *pthread_mutex_init*( ) functions are marked as part of the Threads option.

The *pthread_mutex_timedlock*( ) function is added to the SEE ALSO section for alignment with IEEE Std. 1003.1d-1999.

IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION.

The **restrict** keyword is added to the *pthread_mutex_init*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

33951 **NAME**

33952 pthread_mutex_getprioceiling, pthread_mutex_setprioceiling — change the priority ceiling of a
33953 mutex (**REALTIME THREADS**)

33954 **SYNOPSIS**

33955 TPP
```
#include <pthread.h>

int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict mutex,
    int *restrict prioceiling);
int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
    int prioceiling, int *restrict old_ceiling);
```

33961 **DESCRIPTION**

33962 The *pthread_mutex_getprioceiling*( ) function shall return the current priority ceiling of the mutex.

33963 The *pthread_mutex_setprioceiling*( ) function either locks the mutex if it is unlocked, or blocks until
33964 it can successfully lock the mutex, then it changes the mutex's priority ceiling and releases the
33965 mutex. When the change is successful, the previous value of the priority ceiling is returned in
33966 *old_ceiling*. The process of locking the mutex need not adhere to the priority protect protocol.

33967 If the *pthread_mutex_setprioceiling*( ) function fails, the mutex priority ceiling shall not be
33968 changed.

33969 **RETURN VALUE**

33970 If successful, the *pthread_mutex_getprioceiling*( ) and *pthread_mutex_setprioceiling*( ) functions shall
33971 return zero; otherwise, an error number shall be returned to indicate the error.

33972 **ERRORS**

33973 The *pthread_mutex_getprioceiling*( ) and *pthread_mutex_setprioceiling*( ) functions may fail if:

33974 [EINVAL] The priority requested by *prioceiling* is out of range.

33975 [EINVAL] The value specified by *mutex* does not refer to a currently existing mutex.

33976 [EPERM] The caller does not have the privilege to perform the operation.

33977 These functions shall not return an error code of [EINTR].

33978 **EXAMPLES**

33979 None.

33980 **APPLICATION USAGE**

33981 None.

33982 **RATIONALE**

33983 None.

33984 **FUTURE DIRECTIONS**

33985 None.

33986 **SEE ALSO**

33987 *pthread_mutex_destroy*( ), *pthread_mutex_lock*( ), *pthread_mutex_timedlock*( ), the Base Definitions
33988 volume of IEEE Std. 1003.1-200x, **<pthread.h>**

33989 **CHANGE HISTORY**

33990 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33991 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 Thread Priority Protection option.

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 [ENOSYS] error denoting non-support of the priority ceiling protocol for mutexes has been removed. This is since if the implementation provides the functions (regardless of whether _POSIX_PTHREAD_PRIO_PROTECT is defined), they must function as in the DESCRIPTION and therefore the priority ceiling protocol for mutexes is supported.

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.

34005 **NAME**

34006 pthread_mutex_init — initialize a mutex

34007 **SYNOPSIS**

34008 THR
```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```
34012

34013 **DESCRIPTION**

34014 Refer to *pthread_mutex_destroy*( ).

34015 **NAME**

34016 pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a
34017 mutex

34018 **SYNOPSIS**

34019 THR
```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```
34023

34024 **DESCRIPTION**

34025 The mutex object referenced by *mutex* shall be locked by calling *pthread_mutex_lock*( ). If the
34026 mutex is already locked, the calling thread shall block until the mutex becomes available. This
34027 operation shall return with the mutex object referenced by *mutex* in the locked state with the
34028 calling thread as its owner.

34029 XSI If the mutex type is PTHREAD_MUTEX_NORMAL, deadlock detection shall not be provided.
34030 Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it
34031 has not locked or a mutex which is unlocked, undefined behavior results.

34032 If the mutex type is PTHREAD_MUTEX_ERRORCHECK, then error checking shall be provided.
34033 If a thread attempts to relock a mutex that it has already locked, an error is returned. If a thread
34034 attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error is
34035 returned.

34036 If the mutex type is PTHREAD_MUTEX_RECURSIVE, then the mutex shall maintain the
34037 concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock
34038 count is set to one. Every time a thread relocks this mutex, the lock count is incremented by one.
34039 Each time the thread unlocks the mutex, the lock count is decremented by one. When the lock
34040 count reaches zero, the mutex becomes available for other threads to acquire. If a thread
34041 attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error is
34042 returned.

34043 If the mutex type is PTHREAD_MUTEX_DEFAULT, attempting to recursively lock the mutex
34044 results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling
34045 thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in
34046 undefined behavior.

34047 The *pthread_mutex_trylock*( ) function is identical to *pthread_mutex_lock*( ) except that if the mutex
34048 object referenced by *mutex* is currently locked (by any thread, including the current thread), the
34049 call shall return immediately.

34050 XSI The *pthread_mutex_unlock*( ) function releases the mutex object referenced by *mutex*. The manner
34051 in which a mutex is released is dependent upon the mutex's type attribute. If there are threads
34052 blocked on the mutex object referenced by *mutex* when *pthread_mutex_unlock*( ) is called,
34053 resulting in the mutex becoming available, the scheduling policy is used to determine which
34054 thread shall acquire the mutex.

34055 XSI (In the case of PTHREAD_MUTEX_RECURSIVE mutexes, the mutex shall become available
34056 when the count reaches zero and the calling thread no longer has any locks on this mutex).

34057 If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the
34058 thread shall resume waiting for the mutex as if it was not interrupted.

34059 **RETURN VALUE**

34060 If successful, the *pthread_mutex_lock*( ) and *pthread_mutex_unlock*( ) functions shall return zero;
34061 otherwise, an error number shall be returned to indicate the error.

34062 The *pthread_mutex_trylock*( ) function shall return zero if a lock on the mutex object referenced by
34063 *mutex* is acquired. Otherwise, an error number is returned to indicate the error.

34064 **ERRORS**

34065 The *pthread_mutex_lock*( ) and *pthread_mutex_trylock*( ) functions shall fail if:

34066 [EINVAL] The *mutex* was created with the protocol attribute having the value |
34067 PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than
34068 the mutex's current priority ceiling.

34069 The *pthread_mutex_trylock*( ) function shall fail if:

34070 [EBUSY] The *mutex* could not be acquired because it was already locked. |

34071 The *pthread_mutex_lock*( ), *pthread_mutex_trylock*( ), and *pthread_mutex_unlock*( ) functions may
34072 fail if:

34073 [EINVAL] The value specified by *mutex* does not refer to an initialized mutex object. |

34074 XSI [EAGAIN] The mutex could not be acquired because the maximum number of recursive |
34075 locks for *mutex* has been exceeded.

34076 The *pthread_mutex_lock*( ) function may fail if:

34077 [EDEADLK] The current thread already owns the mutex. |

34078 The *pthread_mutex_unlock*( ) function may fail if:

34079 [EPERM] The current thread does not own the mutex. |

34080 These functions shall not return an error code of [EINTR]. |

34081 **EXAMPLES**

34082 None.

34083 **APPLICATION USAGE**

34084 None.

34085 **RATIONALE**

34086 Mutex objects are intended to serve as a low-level primitive from which other thread
34087 synchronization functions can be built. As such, the implementation of mutexes should be as
34088 efficient as possible, and this has ramifications on the features available at the interface.

34089 The mutex functions and the particular default settings of the mutex attributes have been
34090 motivated by the desire to not preclude fast, inlined implementations of mutex locking and
34091 unlocking.

34092 For example, deadlocking on a double-lock is explicitly allowed behavior in order to avoid
34093 requiring more overhead in the basic mechanism than is absolutely necessary. (More ''friendly''
34094 mutexes that detect deadlock or that allow multiple locking by the same thread are easily
34095 constructed by the user via the other mechanisms provided. For example, *pthread_self*( ) can be
34096 used to record mutex ownership.) Implementations might also choose to provide such extended
34097 features as options via special mutex attributes.

34098 Since most attributes only need to be checked when a thread is going to be blocked, the use of
34099 attributes does not slow the (common) mutex-locking case.

Likewise, while being able to extract the thread ID of the owner of a mutex might be desirable, it would require storing the current thread ID when each mutex is locked, and this could incur unacceptable levels of overhead. Similar arguments apply to a *mutex_tryunlock* operation.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_destroy( ), *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.

Issue 6

The *pthread_mutex_lock*( ), *pthread_mutex_trylock*( ), and *pthread_mutex_unlock*( ) functions are marked as part of the Threads option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The behavior when attempting to relock a mutex is defined.

The *pthread_mutex_timedlock*( ) function is added to the SEE ALSO section for alignment with IEEE Std. 1003.1d-1999.

34118 **NAME**

34119 pthread_mutex_setprioceiling — change the priority ceiling of a mutex (**REALTIME**
34120 **THREADS**)

34121 **SYNOPSIS**

34122 TPP
```
#include <pthread.h>
```

34123
```
int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
    int prioceiling, int *restrict old_ceiling);
```
34124
34125

34126 **DESCRIPTION**

34127 Refer to *pthread_mutex_getprioceiling*( ).

34128 **NAME**

34129 pthread_mutex_timedlock — lock a mutex (**REALTIME THREADS**)

34130 **SYNOPSIS**

34131 THR TMO
```
#include <pthread.h>
#include <time.h>

int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abs_timeout);
```

34136 **DESCRIPTION**

34137 The *pthread_mutex_timedlock*( ) function is called to lock the mutex object referenced by *mutex*. If
34138 the mutex is already locked, the calling thread blocks until the mutex becomes available as in the
34139 *pthread_mutex_lock*( ) function. If the mutex cannot be locked without waiting for another thread
34140 to unlock the mutex, this wait shall be terminated when the specified timeout expires.

34141 The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the
34142 clock on which timeouts are based (that is, when the value of that clock equals or exceeds
34143 *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time
34144 of the call. If the Timers option is supported, the timeout is based on the CLOCK_REALTIME
34145 clock; if the Timers option is not supported, the timeout is based on the system clock as returned
34146 by the *time*( ) function. The resolution of the timeout is the resolution of the clock on which it is
34147 based. The **timespec** datatype is defined as a structure in the **<time.h>** header.

34148 Under no circumstance will the function fail with a timeout if the mutex can be locked
34149 immediately. The validity of the *abs_timeout* parameter need not be checked if the mutex can be
34150 locked immediately.

34151 As a consequence of the priority inheritance rules (for mutexes initialized with the
34152 PRIO_INHERIT protocol), if a timed mutex wait is terminated because its timeout expires, the
34153 priority of the owner of the mutex will be adjusted as necessary to reflect the fact that this thread
34154 is no longer among the threads waiting for the mutex.

34155 **RETURN VALUE**

34156 If successful, the *pthread_mutex_timedlock*( ) function shall return zero; otherwise, an error
34157 number shall be returned to indicate the error.

34158 **ERRORS**

34159 The *pthread_mutex_timedlock*( ) function shall fail if:

34160 [EINVAL] The mutex was created with the protocol attribute having the value
34161 PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than
34162 the mutex' current priority ceiling.

34163 [EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter
34164 specified a nanoseconds field value less than zero or greater than or equal to
34165 1 000 million.

34166 [ETIMEDOUT] The mutex could not be locked before the specified timeout expired.

34167 The *pthread_mutex_timedlock*( ) function may fail if:

34168 [EINVAL] The value specified by mutex does not refer to an initialized mutex object.

34169 XSI [EAGAIN] The mutex could not be acquired because the maximum number of recursive
34170 locks for mutex has been exceeded.

34171 [EDEADLK] The current thread already owns the mutex.

34172 This function shall not return an error code of [EINTR].

34173 **EXAMPLES**

34174 None.

34175 **APPLICATION USAGE**

34176 The *pthread_mutex_timedlock*( ) function is part of the Threads and Timeouts options and need
34177 not be provided on all implementations.

34178 **RATIONALE**

34179 None.

34180 **FUTURE DIRECTIONS**

34181 None.

34182 **SEE ALSO**

34183 *pthread_mutex_destroy*( ), *pthread_mutex_lock*( ), *pthread_mutex_trylock*( ), *time*( ), the Base
34184 Definitions volume of IEEE Std. 1003.1-200x, **<pthread.h>**, **<time.h>**

34185 **CHANGE HISTORY**

34186 First released in Issue 6. Derived from IEEE Std. 1003.1d-1999.

34187 **NAME**

34188 pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a mutex

34189 **SYNOPSIS**

34190 THR
```
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```
34193

34194 **DESCRIPTION**

34195 Refer to *pthread_mutex_lock*( ).

34196 **NAME**

34197 pthread_mutexattr_destroy, pthread_mutexattr_init — destroy and initialize mutex attributes
34198 object

34199 **SYNOPSIS**

34200 THR
```
#include <pthread.h>

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```
34203

34204 **DESCRIPTION**

34205 The *pthread_mutexattr_destroy*( ) function destroys a mutex attributes object; the object becomes,
34206 in effect, uninitialized. An implementation may cause *pthread_mutexattr_destroy*( ) to set the
34207 object referenced by *attr* to an invalid value. A destroyed mutex attributes object can be re-
34208 initialized using *pthread_mutexattr_init*( ); the results of otherwise referencing the object after it
34209 has been destroyed are undefined.

34210 The *pthread_mutexattr_init*( ) function initializes a mutex attributes object *attr* with the default
34211 value for all of the attributes defined by the implementation.

34212 The effect of initializing an already initialized mutex attributes object is undefined.

34213 After a mutex attributes object has been used to initialize one or more mutexes, any function
34214 affecting the attributes object (including destruction) does not affect any previously initialized
34215 mutexes.

34216 **RETURN VALUE**

34217 Upon successful completion, *pthread_mutexattr_destroy*( ) and *pthread_mutexattr_init*( ) shall
34218 return zero; otherwise, an error number shall be returned to indicate the error.

34219 **ERRORS**

34220 The *pthread_mutexattr_destroy*( ) function may fail if:

34221 [EINVAL] The value specified by *attr* is invalid. |

34222 The *pthread_mutexattr_init*( ) function shall fail if: |

34223 [ENOMEM] Insufficient memory exists to initialize the mutex attributes object. |

34224 These functions shall not return an error code of [EINTR]. |

34225 **EXAMPLES**

34226 None.

34227 **APPLICATION USAGE**

34228 None.

34229 **RATIONALE**

34230 See *pthread_attr_init*( ) for a general explanation of attributes. Attributes objects allow
34231 implementations to experiment with useful extensions and permit extension of this volume of
34232 IEEE Std. 1003.1-200x without changing the existing functions. Thus, they provide for future
34233 extensibility of this volume of IEEE Std. 1003.1-200x and reduce the temptation to standardize
34234 prematurely on semantics that are not yet widely implemented or understood.

34235 Examples of possible additional mutex attributes that have been discussed are *spin_only*,
34236 *limited_spin*, *no_spin*, *recursive*, and *metered*. (To explain what the latter attributes might mean:
34237 recursive mutexes would allow for multiple re-locking by the current owner; metered mutexes
34238 would transparently keep records of queue length, wait time, and so on.) Since there is not yet
34239 wide agreement on the usefulness of these resulting from shared implementation and usage

34240 experience, they are not yet specified in this volume of IEEE Std. 1003.1-200x. Mutex attributes
34241 objects, however, make it possible to test out these concepts for possible standardization at a
34242 later time.

34243 **Mutex Attributes and Performance**

34244 Care has been taken to ensure that the default values of the mutex attributes have been defined
34245 such that mutexes initialized with the defaults have simple enough semantics so that the locking
34246 and unlocking can be done with the equivalent of a test-and-set instruction (plus possibly a few
34247 other basic instructions).

34248 There is at least one implementation method that can be used to reduce the cost of testing at
34249 lock-time if a mutex has non-default attributes. One such method that an implementation can
34250 employ (and this can be made fully transparent to fully conforming POSIX applications) is to
34251 secretly pre-lock any mutexes that are initialized to non-default attributes. Any later attempt to
34252 lock such a mutex causes the implementation to branch to the ‘‘slow path’’ as if the mutex were
34253 unavailable; then, on the slow path, the implementation can do the ‘‘real work’’ to lock a non-
34254 default mutex. The underlying unlock operation is more complicated since the implementation
34255 never really wants to release the pre-lock on this kind of mutex. This illustrates that, depending
34256 on the hardware, there may be certain optimizations that can be used so that whatever mutex
34257 attributes are considered ‘‘most frequently used’’ can be processed most efficiently.

34258 **Process Shared Memory and Synchronization**

34259 The existence of memory mapping functions in this volume of IEEE Std. 1003.1-200x leads to the
34260 possibility that an application may allocate the synchronization objects from this section in
34261 memory that is accessed by multiple processes (and therefore, by threads of multiple processes).

34262 In order to permit such usage, while at the same time keeping the usual case (that is, usage
34263 within a single process) efficient, a process-shared option has been defined.

34264 If an implementation supports the _POSIX_THREAD_PROCESS_SHARED option, then the
34265 *process-shared* attribute can be used to indicate that mutexes or condition variables may be
34266 accessed by threads of multiple processes.

34267 The default setting of PTHREAD_PROCESS_PRIVATE has been chosen for the *process-shared*
34268 attribute so that the most efficient forms of these synchronization objects are created by default.

34269 Synchronization variables that are initialized with the PTHREAD_PROCESS_PRIVATE *process-*
34270 *shared* attribute may only be operated on by threads in the process that initialized them.
34271 Synchronization variables that are initialized with the PTHREAD_PROCESS_SHARED *process-*
34272 *shared* attribute may be operated on by any thread in any process that has access to it. In
34273 particular, these processes may exist beyond the lifetime of the initializing process. For example,
34274 the following code implements a simple counting semaphore in a mapped file that may be used
34275 by many processes.

```
34276 /* sem.h */
34277 struct semaphore {
34278     pthread_mutex_t lock;
34279     pthread_cond_t nonzero;
34280     unsigned count;
34281 };
34282 typedef struct semaphore semaphore_t;

34283 semaphore_t *semaphore_create(char *semaphore_name);
34284 semaphore_t *semaphore_open(char *semaphore_name);
34285 void semaphore_post(semaphore_t *semap);
```

```
34286       void semaphore_wait(semaphore_t *semap);
34287       void semaphore_close(semaphore_t *semap);

34288       /* sem.c */
34289       #include <sys/types.h>
34290       #include <sys/stat.h>
34291       #include <sys/mman.h>
34292       #include <fcntl.h>
34293       #include <pthread.h>
34294       #include "sem.h"

34295       semaphore_t *
34296       semaphore_create(char *semaphore_name)
34297       {
34298       int fd;
34299           semaphore_t *semap;
34300           pthread_mutexattr_t psharedm;
34301           pthread_condattr_t psharedc;

34302           fd = open(semaphore_name, O_RDWR | O_CREAT | O_EXCL, 0666);
34303           if (fd < 0)
34304               return (NULL);
34305           (void) ftruncate(fd, sizeof(semaphore_t));
34306           (void) pthread_mutexattr_init(&psharedm);
34307           (void) pthread_mutexattr_setpshared(&psharedm,
34308               PTHREAD_PROCESS_SHARED);
34309           (void) pthread_condattr_init(&psharedc);
34310           (void) pthread_condattr_setpshared(&psharedc,
34311               PTHREAD_PROCESS_SHARED);
34312           semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
34313                   PROT_READ | PROT_WRITE, MAP_SHARED,
34314                   fd, 0);
34315           close (fd);
34316           (void) pthread_mutex_init(&semap->lock, &psharedm);
34317           (void) pthread_cond_init(&semap->nonzero, &psharedc);
34318           semap->count = 0;
34319           return (semap);
34320       }

34321       semaphore_t *
34322       semaphore_open(char *semaphore_name)
34323       {
34324           int fd;
34325           semaphore_t *semap;

34326           fd = open(semaphore_name, O_RDWR, 0666);
34327           if (fd < 0)
34328               return (NULL);
34329           semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
34330                   PROT_READ | PROT_WRITE, MAP_SHARED,
34331                   fd, 0);
34332           close (fd);
34333           return (semap);
34334       }
```

```
34335       void
34336       semaphore_post(semaphore_t *semap)
34337       {
34338           pthread_mutex_lock(&semap->lock);
34339           if (semap->count == 0)
34340               pthread_cond_signal(&semapx->nonzero);
34341           semap->count++;
34342           pthread_mutex_unlock(&semap->lock);
34343       }

34344       void
34345       semaphore_wait(semaphore_t *semap)
34346       {
34347           pthread_mutex_lock(&semap->lock);
34348           while (semap->count == 0)
34349               pthread_cond_wait(&semap->nonzero, &semap->lock);
34350           semap->count--;
34351           pthread_mutex_unlock(&semap->lock);
34352       }

34353       void
34354       semaphore_close(semaphore_t *semap)
34355       {
34356           munmap((void *) semap, sizeof(semaphore_t));
34357       }
```

34358 The following code is for three separate processes that create, post, and wait on a semphore in
34359 the file **/tmp/semaphore**. Once the file is created, the post and wait programs increment and
34360 decrement the counting semaphore (waiting and waking as required) even though they did not
34361 initialize the semaphore.

```
34362       /* create.c */
34363       #include "pthread.h"
34364       #include "sem.h"

34365       int
34366       main()
34367       {
34368           semaphore_t *semap;

34369           semap = semaphore_create("/tmp/semaphore");
34370           if (semap == NULL)
34371               exit(1);
34372           semaphore_close(semap);
34373           return (0);
34374       }

34375       /* post */
34376       #include "pthread.h"
34377       #include "sem.h"

34378       int
34379       main()
34380       {
34381           semaphore_t *semap;
```

```
34382           semap = semaphore_open("/tmp/semaphore");
34383           if (semap == NULL)
34384               exit(1);
34385           semaphore_post(semap);
34386           semaphore_close(semap);
34387           return (0);
34388       }

34389       /* wait */
34390       #include "pthread.h"
34391       #include "sem.h"

34392       int
34393       main()
34394       {
34395           semaphore_t *semap;

34396           semap = semaphore_open("/tmp/semaphore");
34397           if (semap == NULL)
34398               exit(1);
34399           semaphore_wait(semap);
34400           semaphore_close(semap);
34401           return (0);
34402       }
```

34403 **FUTURE DIRECTIONS**

34404 None.

34405 **SEE ALSO**

34406 *pthread_cond_destroy*( ), *pthread_create*( ), *pthread_mutex_destroy*( ), *pthread_mutexattr_destroy*( ), the |
34407 Base Definitions volume of IEEE Std. 1003.1-200x, **<pthread.h>** |

34408 **CHANGE HISTORY**

34409 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34410 **Issue 6**

34411 The *pthread_mutexattr_destroy*( ) and *pthread_mutexattr_init*( ) functions are marked as part of the |
34412 Threads option. |

34413 IEEE PASC Interpretation 1003.1c #27 is applied, updating the ERRORS section. |

NAME

pthread_mutexattr_getprioceiling, pthread_mutexattr_setprioceiling — get and set prioceiling attribute of mutex attributes object (**REALTIME THREADS**)

SYNOPSIS

TPP
```
#include <pthread.h>

int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *restrict attr,|
    int *restrict prioceiling);                                               |
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,               |
    int prioceiling);
```

DESCRIPTION

The *pthread_mutexattr_getprioceiling*( ) and *pthread_mutexattr_setprioceiling*( ) functions, respectively, get and set the priority ceiling attribute of a mutex attributes object pointed to by *attr* which was previously created by the function *pthread_mutexattr_init*( ).

The *prioceiling* attribute contains the priority ceiling of initialized mutexes. The values of *prioceiling* are within the maximum range of priorities defined by SCHED_FIFO.

The *prioceiling* attribute defines the priority ceiling of initialized mutexes, which is the minimum priority level at which the critical section guarded by the mutex is executed. In order to avoid priority inversion, the priority ceiling of the mutex is set to a priority higher than or equal to the highest priority of all the threads that may lock that mutex. The values of *prioceiling* are within the maximum range of priorities defined under the SCHED_FIFO scheduling policy.

RETURN VALUE

Upon successful completion, the *pthread_mutexattr_getprioceiling*( ) and *pthread_mutexattr_setprioceiling*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutexattr_getprioceiling*( ) and *pthread_mutexattr_setprioceiling*( ) functions may fail if:

[EINVAL] The value specified by *attr* or *prioceiling* is invalid. |

[EPERM] The caller does not have the privilege to perform the operation. |

These functions shall not return an error code of [EINTR]. |

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_destroy( ), *pthread_create*( ), *pthread_mutex_destroy*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<pthread.h>** |

34455 **CHANGE HISTORY**

34456 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34457 Marked as part of the Realtime Threads Feature Group.

34458 **Issue 6**

34459 The *pthread_mutexattr_getprioceiling*() and *pthread_mutexattr_setprioceiling*() functions are |
34460 marked as part of the Thread Priority Protection option. |

34461 The [ENOSYS] error condition has been removed as stubs need not be provided if an
34462 implementation does not support the Thread Priority Protection option. |

34463 The [ENOTSUP] error condition has been removed since these functions do not have a *protocol*
34464 argument. |

34465 The **restrict** keyword is added to the *pthread_mutexattr_getprioceiling*() prototype for alignment |
34466 with the ISO/IEC 9899:1999 standard. |

NAME

pthread_mutexattr_getprotocol, pthread_mutexattr_setprotocol — get and set protocol attribute of mutex attributes object (**REALTIME THREADS**)

SYNOPSIS

TPP|TPI
```
#include <pthread.h>

int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *restrict attr,|
    int *restrict protocol);                                               |
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,               |
    int protocol);
```

DESCRIPTION

The *pthread_mutexattr_getprotocol*( ) and *pthread_mutexattr_setprotocol*( ) functions, respectively, get and set the protocol attribute of a mutex attributes object pointed to by *attr* which was previously created by the function *pthread_mutexattr_init*( ).

The *protocol* attribute defines the protocol to be followed in utilizing mutexes. The value of
TPP *protocol* may be one of PTHREAD_PRIO_NONE, PTHREAD_PRIO_INHERIT, or PTHREAD_PRIO_PROTECT, which are defined by the header **<pthread.h>**.

When a thread owns a mutex with the PTHREAD_PRIO_NONE *protocol* attribute, its priority and scheduling are not affected by its mutex ownership.

TPI When a thread is blocking higher priority threads because of owning one or more mutexes with the PTHREAD_PRIO_INHERIT protocol attribute, it executes at the higher of its priority or the priority of the highest priority thread waiting on any of the mutexes owned by this thread and initialized with this protocol.

TPP When a thread owns one or more mutexes initialized with the PTHREAD_PRIO_PROTECT protocol, it executes at the higher of its priority or the highest of the priority ceilings of all the mutexes owned by this thread and initialized with this attribute, regardless of whether other threads are blocked on any of these mutexes or not.

While a thread is holding a mutex which has been initialized with the | PTHREAD_PRIO_INHERIT or PTHREAD_PRIO_PROTECT protocol attributes, it shall not be subject to being moved to the tail of the scheduling queue at its priority in the event that its original priority is changed, such as by a call to *sched_setparam*( ). Likewise, when a thread unlocks a mutex that has been initialized with the PTHREAD_PRIO_INHERIT or PTHREAD_PRIO_PROTECT protocol attributes, it shall not be subject to being moved to the tail of the scheduling queue at its priority in the event that its original priority is changed. |

If a thread simultaneously owns several mutexes initialized with different protocols, it shall execute at the highest of the priorities that it would have obtained by each of these protocols.

TPI When a thread makes a call to *pthread_mutex_lock*( ), the mutex was initialized with the protocol attribute having the value PTHREAD_PRIO_INHERIT, when the calling thread is blocked because the mutex is owned by another thread, that owner thread shall inherit the priority level of the calling thread as long as it continues to own the mutex. The implementation shall update its execution priority to the maximum of its assigned priority and all its inherited priorities. Furthermore, if this owner thread itself becomes blocked on another mutex, the same priority inheritance effect shall be propagated to this other owner thread, in a recursive manner.

34510 **RETURN VALUE**

34511 Upon successful completion, the *pthread_mutexattr_getprotocol*( ) and
34512 *pthread_mutexattr_setprotocol*( ) functions shall return zero; otherwise, an error number shall be
34513 returned to indicate the error.

34514 **ERRORS**

34515 The *pthread_mutexattr_setprotocol*( ) function shall fail if:

34516 [ENOTSUP] The value specified by *protocol* is an unsupported value. |

34517 The *pthread_mutexattr_getprotocol*( ) and *pthread_mutexattr_setprotocol*( ) functions may fail if:

34518 [EINVAL] The value specified by *attr* or *protocol* is invalid. |

34519 [EPERM] The caller does not have the privilege to perform the operation. |

34520 These functions shall not return an error code of [EINTR]. |

34521 **EXAMPLES**

34522 None.

34523 **APPLICATION USAGE**

34524 None.

34525 **RATIONALE**

34526 None.

34527 **FUTURE DIRECTIONS**

34528 None.

34529 **SEE ALSO**

34530 *pthread_cond_destroy*( ), *pthread_create*( ), *pthread_mutex_destroy*( ), the Base Definitions volume of |
34531 IEEE Std. 1003.1-200x, **<pthread.h>** |

34532 **CHANGE HISTORY**

34533 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34534 Marked as part of the Realtime Threads Feature Group.

34535 **Issue 6**

34536 The *pthread_mutexattr_getprotocol*( ) and *pthread_mutexattr_setprotocol*( ) functions are marked as |
34537 part of either the Thread Priority Protection or Threads Priority Inheritance options. |

34538 The [ENOSYS] error condition has been removed as stubs need not be provided if an
34539 implementation does not support the Thread Priority Protection or Threads Priority Inheritance |
34540 options. |

34541 The **restrict** keyword is added to the *pthread_mutexattr_getprotocol*( ) prototype for alignment |
34542 with the ISO / IEC 9899: 1999 standard. |

NAME

pthread_mutexattr_getpshared, pthread_mutexattr_setpshared — get and set process-shared attribute

SYNOPSIS

THR TSH

```
#include <pthread.h>

int pthread_mutexattr_getpshared(const pthread_mutexattr_t *restrict attr,
    int *restrict pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
    int pshared);
```

DESCRIPTION

The *pthread_mutexattr_getpshared*( ) function obtains the value of the *process-shared* attribute from the attributes object referenced by *attr*. The *pthread_mutexattr_setpshared*( ) function is used to set the *process-shared* attribute in an initialized attributes object referenced by *attr*.

The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes. If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the mutex shall only be operated upon by threads created within the same process as the thread that initialized the mutex; if threads of differing processes attempt to operate on such a mutex, the behavior is undefined. The default value of the attribute shall be PTHREAD_PROCESS_PRIVATE.

RETURN VALUE

Upon successful completion, *pthread_mutexattr_setpshared*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *pthread_mutexattr_getpshared*( ) shall return zero and stores the value of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutexattr_getpshared*( ) and *pthread_mutexattr_setpshared*( ) functions may fail if:

[EINVAL] The value specified by *attr* is invalid.

The *pthread_mutexattr_setpshared*( ) function may fail if:

[EINVAL] The new value specified for the attribute is outside the range of legal values for that attribute.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

34585 **SEE ALSO**

34586 *pthread_cond_destroy*( ), *pthread_create*( ), *pthread_mutex_destroy*( ), *pthread_mutexattr_destroy*( ), the |
34587 Base Definitions volume of IEEE Std. 1003.1-200x, **<pthread.h>** |

34588 **CHANGE HISTORY**

34589 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34590 **Issue 6**

34591 The *pthread_mutexattr_getpshared*( ) and *pthread_mutexattr_setpshared*( ) functions are marked as |
34592 part of the Threads and Thread Process-Shared Synchronization options. |

34593 The **restrict** keyword is added to the *pthread_mutexattr_getpshared*( ) prototype for alignment |
34594 with the ISO / IEC 9899: 1999 standard. |

34595 **NAME**

34596 pthread_mutexattr_gettype, pthread_mutexattr_settype — get or set a mutex type |

34597 **SYNOPSIS**

34598 XSI
```
#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);      |
```
34602

34603 **DESCRIPTION**

34604 The *pthread_mutexattr_gettype*( ) and *pthread_mutexattr_settype*( ) functions, respectively, get and
34605 set the mutex *type* attribute. This attribute is set in the *type* parameter to these functions. The
34606 default value of the *type* attribute is PTHREAD_MUTEX_DEFAULT.

34607 The type of mutex is contained in the *type* attribute of the mutex attributes. Valid mutex types
34608 include:

34609 PTHREAD_MUTEX_NORMAL
34610 This type of mutex does not detect deadlock. A thread attempting to relock this mutex
34611 without first unlocking it shall deadlock. Attempting to unlock a mutex locked by a
34612 different thread results in undefined behavior. Attempting to unlock an unlocked mutex
34613 results in undefined behavior.

34614 PTHREAD_MUTEX_ERRORCHECK
34615 This type of mutex provides error checking. A thread attempting to relock this mutex
34616 without first unlocking it shall return with an error. A thread attempting to unlock a mutex
34617 which another thread has locked shall return with an error. A thread attempting to unlock
34618 an unlocked mutex shall return with an error.

34619 PTHREAD_MUTEX_RECURSIVE
34620 A thread attempting to relock this mutex without first unlocking it shall succeed in locking
34621 the mutex. The relocking deadlock which can occur with mutexes of type
34622 PTHREAD_MUTEX_NORMAL cannot occur with this type of mutex. Multiple locks of this
34623 mutex require the same number of unlocks to release the mutex before another thread can
34624 acquire the mutex. A thread attempting to unlock a mutex which another thread has locked
34625 shall return with an error. A thread attempting to unlock an unlocked mutex shall return
34626 with an error.

34627 PTHREAD_MUTEX_DEFAULT
34628 Attempting to recursively lock a mutex of this type results in undefined behavior.
34629 Attempting to unlock a mutex of this type which was not locked by the calling thread
34630 results in undefined behavior. Attempting to unlock a mutex of this type which is not
34631 locked results in undefined behavior. An implementation is allowed to map this mutex to
34632 one of the other mutex types.

34633 **RETURN VALUE**

34634 Upon successful completion, the *pthread_mutexattr_gettype*( ) function shall return zero and store
34635 the value of the *type* attribute of *attr* into the object referenced by the *type* parameter. Otherwise,
34636 an error shall be returned to indicate the error.

34637 If successful, the *pthread_mutexattr_settype*( ) function shall return zero; otherwise, an error
34638 number shall be returned to indicate the error.

34639 **ERRORS**

34640 The *pthread_mutexattr_settype*( ) function shall fail if:

34641 [EINVAL] The value *type* is invalid.

34642 The *pthread_mutexattr_gettype*( ) and *pthread_mutexattr_settype*( ) functions may fail if:

34643 [EINVAL] The value specified by *attr* is invalid.

34644 These functions shall not return an error code of [EINTR].

34645 **EXAMPLES**

34646 None.

34647 **APPLICATION USAGE**

34648 It is advised that an application should not use a PTHREAD_MUTEX_RECURSIVE mutex with
34649 condition variables because the implicit unlock performed for a *pthread_cond_timedwait*( ) or
34650 *pthread_cond_wait*( ) may not actually release the mutex (if it had been locked multiple times). If
34651 this happens, no other thread can satisfy the condition of the predicate.

34652 **RATIONALE**

34653 None.

34654 **FUTURE DIRECTIONS**

34655 None.

34656 **SEE ALSO**

34657 *pthread_cond_timedwait*( ), *pthread_cond_wait*( ), the Base Definitions volume of
34658 IEEE Std. 1003.1-200x, **<pthread.h>**

34659 **CHANGE HISTORY**

34660 First released in Issue 5.

34661 **Issue 6**

34662 The Open Group corrigenda item U033/3 has been applied. The SYNOPSIS for
34663 *pthread_mutexattr_gettype*( ) is updated so that the first argument is of type
34664 **const**pthread_mutexattr_t***.**

34665 The **restrict** keyword is added to the *pthread_mutexattr_gettype*( ) prototype for alignment with
34666 the ISO/IEC 9899:1999 standard.

34667 **NAME**

34668 pthread_mutexattr_init — initialize mutex attributes object

34669 **SYNOPSIS**

34670 THR
```
#include <pthread.h>
```

34671
```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

34672

34673 **DESCRIPTION**

34674 Refer to *pthread_mutexattr_destroy*( ).

34675 **NAME**

34676 pthread_mutexattr_setprioceiling — set prioceiling attribute of mutex attributes object
34677 (**REALTIME THREADS**)

34678 **SYNOPSIS**

34679 TPP
```
#include <pthread.h>
```

34680
```
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
    int prioceiling);
```
34681
34682

34683 **DESCRIPTION**

34684 Refer to *pthread_mutexattr_getprioceiling*( ).

34685 **NAME**

34686 pthread_mutexattr_setprotocol — set protocol attribute of mutex attributes object (**REALTIME**
34687 **THREADS**)

34688 **SYNOPSIS**

34689 TPP | TPI
```
#include <pthread.h>

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
    int protocol);
```
34692

34693 **DESCRIPTION**

34694 Refer to *pthread_mutexattr_setprotocol*( ).

34695 **NAME**

34696 pthread_mutexattr_setpshared — set process-shared attribute

34697 **SYNOPSIS**

34698 THR TSH
```
#include <pthread.h>

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
    int pshared);
```
34699
34700
34701

34702 **DESCRIPTION**

34703 Refer to *pthread_mutexattr_getpshared*( ).

34704 **NAME**

34705 pthread_mutexattr_settype — set a mutex type

34706 **SYNOPSIS**

34707 XSI
```
#include <pthread.h>
```

34708
```
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

34709

34710 **DESCRIPTION**

34711 Refer to *pthread_mutexattr_gettype*( ).

34712 **NAME**

34713 pthread_once — dynamic package initialization

34714 **SYNOPSIS**

34715 THR
```
#include <pthread.h>

int pthread_once(pthread_once_t *once_control,
    void (*init_routine)(void));
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```
34719

34720 **DESCRIPTION**

34721 The first call to *pthread_once*( ) by any thread in a process, with a given *once_control*, shall call the
34722 *init_routine* with no arguments. Subsequent calls of *pthread_once*( ) with the same *once_control*
34723 shall not call the *init_routine*. On return from *pthread_once*( ), it is guaranteed that *init_routine* has
34724 completed. The *once_control* parameter is used to determine whether the associated initialization
34725 routine has been called.

34726 The *pthread_once*( ) function is not a cancelation point. However, if *init_routine* is a cancelation
34727 point and is canceled, the effect on *once_control* shall be as if *pthread_once*( ) was never called.

34728 The constant PTHREAD_ONCE_INIT is defined by the header **<pthread.h>**.

34729 The behavior of *pthread_once*( ) is undefined if *once_control* has automatic storage duration or is
34730 not initialized by PTHREAD_ONCE_INIT.

34731 **RETURN VALUE**

34732 Upon successful completion, *pthread_once*( ) shall return zero; otherwise, an error number shall
34733 be returned to indicate the error.

34734 **ERRORS**

34735 ***Notes to Reviewers***

34736 *This section with side shading will not appear in the final copy. - Ed.*

34737 D1, XSH, ERN 255 notes that no error is returned for invalid parameters and proposes the
34738 following:

34739 [EINVAL] If either *once_control* or *init_routine* is invalid.

34740 No errors are defined.

34741 The *pthread_once*( ) function shall not return an error code of [EINTR]. |

34742 **EXAMPLES**

34743 None.

34744 **APPLICATION USAGE**

34745 None.

34746 **RATIONALE**

34747 Some C libraries are designed for dynamic initialization. That is, the global initialization for the
34748 library is performed when the first procedure in the library is called. In a single-threaded
34749 program, this is normally implemented using a static variable whose value is checked on entry
34750 to a routine, as follows:

```
34751 static int random_is_initialized = 0;
34752 extern int initialize_random();
```

```
int random_function()
{
    if (random_is_initialized == 0) {
        initialize_random();
        random_is_initialized = 1;
    }
    ... /* Operations performed after initialization. */
}
```

To keep the same structure in a multi-threaded program, a new primitive is needed. Otherwise, library initialization has to be accomplished by an explicit call to a library-exported initialization function prior to any use of the library.

For dynamic library initialization in a multi-threaded process, a simple initialization flag is not sufficient; the flag needs to be protected against modification by multiple threads simultaneously calling into the library. Protecting the flag requires the use of a mutex; however, mutexes have to be initialized before they are used. Ensuring that the mutex is only initialized once requires a recursive solution to this problem.

The use of *pthread_once*( ) not only supplies an implementation-guaranteed means of dynamic initialization, it provides an aid to the reliable construction of multi-threaded and realtime systems. The preceding example then becomes:

```
#include <pthread.h>
static pthread_once_t random_is_initialized = PTHREAD_ONCE_INIT;
extern int initialize_random();

int random_function()
{
    (void) pthread_once(&random_is_initialized, initialize_random);
    ... /* Operations performed after initialization. */
}
```

Note that a **pthread_once_t** cannot be an array because some compilers do not accept the construct **&<array_name>**.

FUTURE DIRECTIONS

None.

SEE ALSO

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_once*( ) function is marked as part of the Threads option.

34790 **NAME**

34791 pthread_rwlock_destroy, pthread_rwlock_init — destroy and initialize a read-write lock object

34792 **SYNOPSIS**

34793 THR
```
#include <pthread.h>

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
    const pthread_rwlockattr_t *restrict attr);
```

34798 **DESCRIPTION**

34799 The *pthread_rwlock_destroy*( ) function shall destroy the read-write lock object referenced by
34800 *rwlock* and release any resources used by the lock. The effect of subsequent use of the lock is
34801 undefined until the lock is re-initialized by another call to *pthread_rwlock_init*( ). An
34802 implementation may cause *pthread_rwlock_destroy*( ) to set the object referenced by *rwlock* to an
34803 invalid value. Results are undefined if *pthread_rwlock_destroy*( ) is called when any thread holds
34804 *rwlock*. Attempting to destroy an uninitialized read-write lock results in undefined behavior.

34805 The *pthread_rwlock_init*( ) function shall allocate any resources required to use the read-write
34806 lock referenced by *rwlock* and initializes the lock to an unlocked state with attributes referenced
34807 by *attr*. If *attr* is NULL, the default read-write lock attributes are used; the effect is the same as
34808 passing the address of a default read-write lock attributes object. Once initialized, the lock can be
34809 used any number of times without being re-initialized. Results are undefined if
34810 *pthread_rwlock_init*( ) is called specifying an already initialized read-write lock. Results are
34811 undefined if a read-write lock is used without first being initialized.

34812 If the *pthread_rwlock_init*( ) function fails, *rwlock* is not initialized and the contents of *rwlock* are
34813 undefined.

34814 Only the object referenced by *rwlock* may be used for performing synchronization. The result of
34815 referring to copies of that object in calls to *pthread_rwlock_destroy*( ), *pthread_rwlock_rdlock*( ),
34816 *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_tryrdlock*( ),
34817 *pthread_rwlock_trywrlock*( ), *pthread_rwlock_unlock*( ), or *pthread_rwlock_wrlock*( ) is undefined.

34818 **RETURN VALUE**

34819 If successful, the *pthread_rwlock_destroy*( ) and *pthread_rwlock_init*( ) functions shall return zero;
34820 otherwise, an error number shall be returned to indicate the error.

34821 The [EBUSY] and [EINVAL] error checks, if implemented, act as if they were performed
34822 immediately at the beginning of processing for the function and caused an error return prior to
34823 modifying the state of the read-write lock specified by *rwlock*.

34824 **ERRORS**

34825 The *pthread_rwlock_destroy*( ) function may fail if:

34826 [EBUSY] The implementation has detected an attempt to destroy the object referenced
34827 by *rwlock* while it is locked.

34828 [EINVAL] The value specified by *rwlock* is invalid.

34829 The *pthread_rwlock_init*( ) function shall fail if:

34830 [EAGAIN] The system lacked the necessary resources (other than memory) to initialize
34831 another read-write lock.

34832 [ENOMEM] Insufficient memory exists to initialize the read-write lock.

34833 MAN [EPERM] The caller does not have the privilege to perform the operation.

The *pthread_rwlock_init*( ) function may fail if:

[EBUSY] The implementation has detected an attempt to re-initialize the object referenced by *rwlock*, a previously initialized but not yet destroyed read-write lock.

[EINVAL] The value specified by *attr* is invalid.

Notes to Reviewers

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

D1, XSH, ERN 259 noted that no error return is described for when the argument *rwlock* is invalid and proposes adding as a may fail case:

[EINVAL] The value specified by *rwlock* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_rdlock( ), *pthread_rwlock_timedrdlock*( ), *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, **<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 s changed to RWL and the initializer macro is deleted.
- The DESCRIPTION is updated as follows:
  - — It explicitly notes allocation of resources upon initialization of a read-write lock object.
  - — A paragraph is added specifying that copies of read-write lock objects may not be used.
- An [EINVAL] error is added to the ERRORS section for *pthread_rwlock_init*( ), indicating that the *rwlock* value is invalid.
- The SEE ALSO section is updated.

The **restrict** keyword is added to the *pthread_rwlock_init*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

34870 **NAME**

34871 pthread_rwlock_init — initialize a read-write lock object

34872 **SYNOPSIS**

34873 THR
```
#include <pthread.h>
```

34874
```
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
    const pthread_rwlockattr_t *restrict attr);
```
34875

34876

34877 **DESCRIPTION**

34878 Refer to *pthread_rwlock_destroy*( ).

NAME

pthread_rwlock_rdlock, pthread_rwlock_tryrdlock — lock a read-write lock object for reading

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

DESCRIPTION

The *pthread_rwlock_rdlock*( ) function shall apply a read lock to the read-write lock referenced by *rwlock*. The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock. TPS If the Thread Execution Scheduling option is supported, and the threads involved in the lock are executing with the scheduling policies SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC, the calling thread shall not acquire the lock if a writer holds the lock or if writers of higher or equal priority are blocked on the lock; otherwise, the calling thread shall acquire the lock. If the Thread Execution Scheduling option is not supported, it is implementation-defined whether the calling thread acquires the lock when a writer does not hold the lock and there are writers blocked on the lock. If a writer holds the lock, the calling thread shall not acquire the read lock. If the read lock is not acquired, the calling thread blocks (that is, it does not return from the *pthread_rwlock_rdlock*( ) call) until it can acquire the lock. The calling thread may deadlock if at the time the call is made it holds a write lock.

A thread may hold multiple concurrent read locks on *rwlock* (that is, successfully call the *pthread_rwlock_rdlock*( ) function *n* times). If so, the application shall ensure that the thread performs matching unlocks (that is, it calls the *pthread_rwlock_unlock*( ) function *n* times).

The maximum number of simultaneous read locks that an implementation guarantees can be applied to a read-write lock shall be implementation-defined. The *pthread_rwlock_rdlock*( ) function may fail if this maximum would be exceeded.

The *pthread_rwlock_tryrdlock*( ) function shall apply a read lock as in the *pthread_rwlock_rdlock*( ) function, with the exception that the function shall fail if the equivalent *pthread_rwlock_rdlock*( ) call would have blocked the calling thread. In no case does the *pthread_rwlock_tryrdlock*( ) function ever block; it always either acquires the lock or fails and returns immediately.

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 reading, upon return from the signal handler the thread resumes waiting for the read-write lock for reading as if it was not interrupted.

RETURN VALUE

If successful, the *pthread_rwlock_rdlock*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

The *pthread_rwlock_tryrdlock*( ) function shall return zero if the lock for reading on the read-write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_rwlock_tryrdlock*( ) function shall fail if:

[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.

The *pthread_rwlock_rdlock*( ) and *pthread_rwlock_tryrdlock*( ) functions may fail if:

[EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock object.

[EAGAIN] The read lock could not be acquired because the maximum number of read locks for *rwlock* has been exceeded.

The *pthread_rwlock_rdlock*( ) function may fail if:

[EDEADLK] The current thread already owns the read-write lock for writing.

These functions shall not return an error code of [EINTR].

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 3.287, Priority Inversion.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_destroy( ), *pthread_rwlock_init*( ), *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, **<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 RWL.
- The DESCRIPTION is updated as follows:
  - — Conditions under which writers have precedence over readers are specified.
  - — Failure of *pthread_rwlock_tryrdlock*( ) is clarified.
  - — A paragraph on the maximum number of read locks is added.
- In the ERRORS sections, [EBUSY] is modified to take into account write priority, and [EDEADLK] is deleted as a *pthread_rwlock_tryrdlock*( ) error.
- The SEE ALSO section is updated.

34956 **NAME**

34957 pthread_rwlock_timedrdlock — lock a read-write lock for reading

34958 **SYNOPSIS**

34959 THR TMO
```
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
    const struct timespec *restrict abs_timeout);
```

34964 **DESCRIPTION**

34965 The *pthread_rwlock_timedrdlock*( ) function applies a read lock to the read-write lock referenced
34966 by *rwlock* as in the *pthread_rwlock_rdlock*( ) function. However, if the lock cannot be acquired
34967 without waiting for other threads to unlock the lock, this wait shall be terminated when the
34968 specified timeout expires. The timeout expires when the absolute time specified by *abs_timeout*
34969 passes, as measured by the clock on which timeouts are based (that is, when the value of that
34970 clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already
34971 been passed at the time of the call.

34972 TMR If the Timers option is supported, the timeout is based on the CLOCK_REALTIME clock; if the
34973 Timers option is not supported, the timeout is based on the system clock as returned by the
34974 *time*( ) function. The resolution of the timeout is the resolution of the clock on which it is based.
34975 The **timespec** data type is defined as a structure in the **<time.h>** header. Under no circumstances
34976 shall the function fail with a timeout if the lock can be acquired immediately. The validity of the
34977 *abs_timeout* parameter need not be checked if the lock can be immediately acquired.

34978 If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-
34979 write lock via a call to *pthread_rwlock_timedrdlock*( ), upon return from the signal handler the
34980 thread shall resume waiting for the lock as if it was not interrupted.

34981 The calling thread may deadlock if at the time the call is made it holds a write lock on *rwlock*.
34982 The results are undefined if this function is called with an uninitialized read-write lock.

34983 **RETURN VALUE**

34984 The *pthread_rwlock_timedrdlock*( ) function shall return zero if the lock for reading on the read-
34985 write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned
34986 to indicate the error.

34987 **ERRORS**

34988 The *pthread_rwlock_timedrdlock*( ) function shall fail if:

34989 [ETIMEDOUT] The lock could not be acquired before the specified timeout expired.

34990 The *pthread_rwlock_timedrdlock*( ) function may fail if:

34991 [EAGAIN] The read lock could not be acquired because the maximum number of read
34992 locks for lock would be exceeded.

34993 [EDEADLK] The calling thread already holds a write lock on *rwlock*.

34994 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock
34995 object, or the *abs_timeout* nanosecond value is less than zero or greater than or
34996 equal to 1 000 million.

34997 This function shall not return an error code of [EINTR].

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 3.287, Priority Inversion.

The *pthread_rwlock_timedrdlock*( ) function is part of the Threads and Timeouts options and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_destroy( ), *pthread_rwlock_init*( ), *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, **<pthread.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std. 1003.1j-2000.

35016 **NAME**

35017 pthread_rwlock_timedwrlock — lock a read-write lock for writing

35018 **SYNOPSIS**

35019 THR TMO
```
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
    const struct timespec *restrict abs_timeout);
```

35024 **DESCRIPTION**

35025 The *pthread_rwlock_timedwrlock*( ) function applies a write lock to the read-write lock referenced
35026 by *rwlock* as in the *pthread_rwlock_wrlock*( ) function. However, if the lock cannot be acquired
35027 without waiting for other threads to unlock the lock, this wait shall be terminated when the
35028 specified timeout expires. The timeout expires when the absolute time specified by *abs_timeout*
35029 passes, as measured by the clock on which timeouts are based (that is, when the value of that
35030 clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already
35031 been passed at the time of the call.

35032 TMR If the Timers option is supported, the timeout is based on the CLOCK_REALTIME clock; if the
35033 Timers option is not supported, the timeout is based on the system clock as returned by the
35034 *time*( ) function. The resolution of the timeout is the resolution of the clock on which it is based.
35035 The **timespec** data type is defined as a structure in the **<time.h>** header. Under no circumstances
35036 shall the function fail with a timeout if the lock can be acquired immediately. The validity of the
35037 *abs_timeout* parameter need not be checked if the lock can be immediately acquired.

35038 If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-
35039 write lock via a call to *pthread_rwlock_timedwrlock*( ), upon return from the signal handler the
35040 thread shall resume waiting for the lock as if it was not interrupted.

35041 The calling thread may deadlock if at the time the call is made it holds the read-write lock. The
35042 results are undefined if this function is called with an uninitialized read-write lock.

35043 **RETURN VALUE**

35044 The *pthread_rwlock_timedwrlock*( ) function shall return zero if the lock for writing on the read-
35045 write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned
35046 to indicate the error.

35047 **ERRORS**

35048 The *pthread_rwlock_timedwrlock*( ) function shall fail if:

35049 [ETIMEDOUT] The lock could not be acquired before the specified timeout expired.

35050 The *pthread_rwlock_timedwrlock*( ) function may fail if:

35051 [EDEADLK] The calling thread already holds the *rwlock*.

35052 [EINVAL] The value specified by rwlock does not refer to an initialized read-write lock
35053 object, or the *abs_timeout* nanosecond value is less than zero or greater than or
35054 equal to 1 000 million.

35055 This function shall not return an error code of [EINTR].

35056 **EXAMPLES**

35057 None.

35058 **APPLICATION USAGE**

35059 Applications using this function may be subject to priority inversion, as discussed in the Base |
35060 Definitions volume of IEEE Std. 1003.1-200x, Section 3.287, Priority Inversion. |

35061 The *pthread_rwlock_timedwrlock*( ) function is part of the Threads and Timeouts options and need |
35062 not be provided on all implementations. |

35063 **RATIONALE**

35064 None.

35065 **FUTURE DIRECTIONS**

35066 None.

35067 **SEE ALSO**

35068 *pthread_rwlock_destroy*( ), *pthread_rwlock_init*( ), *pthread_rwlock_rdlock*( ),
35069 *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_tryrdlock*( ), *pthread_rwlock_trywrlock*( ),
35070 *pthread_rwlock_unlock*( ), *pthread_rwlock_wrlock*( ), the Base Definitions volume of |
35071 IEEE Std. 1003.1-200x, **<pthread.h>**, **<time.h>** |

35072 **CHANGE HISTORY**

35073 First released in Issue 6. Derived from IEEE Std. 1003.1j-2000. |

35074 **NAME**

35075 pthread_rwlock_tryrdlock — lock a read-write lock object for reading

35076 **SYNOPSIS**

35077 THR
```
#include <pthread.h>
```

35078
```
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

35079

35080 **DESCRIPTION**

35081 Refer to *pthread_rwlock_rdlock*( ).

35082 **NAME**

35083 pthread_rwlock_trywrlock, pthread_rwlock_wrlock — lock a read-write lock object for writing

35084 **SYNOPSIS**

35085 THR
```
#include <pthread.h>

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```
35086

35087

35088

35089 **DESCRIPTION**

35090 The *pthread_rwlock_trywrlock*( ) function shall apply a write lock like the *pthread_rwlock_wrlock*( )
35091 function, with the exception that the function shall fail if any thread currently holds *rwlock* (for
35092 reading or writing).

35093 The *pthread_rwlock_wrlock*( ) function shall apply a write lock to the read-write lock referenced
35094 by *rwlock*. The calling thread acquires the write lock if no other thread (reader or writer) holds
35095 the read-write lock *rwlock*. Otherwise, the thread blocks (that is, does not return from the
35096 *pthread_rwlock_wrlock*( ) call) until it can acquire the lock. The calling thread may deadlock if at
35097 the time the call is made it holds the read-write lock (whether a read or write lock).

35098 Implementations are allowed to favor writers over readers to avoid writer starvation.

35099 Results are undefined if any of these functions are called with an uninitialized read-write lock.

35100 If a signal is delivered to a thread waiting for a read-write lock for writing, upon return from the
35101 signal handler the thread resumes waiting for the read-write lock for writing as if it was not
35102 interrupted.

35103 **RETURN VALUE**

35104 The *pthread_rwlock_trywrlock*( ) function shall return zero if the lock for writing on the read-write
35105 lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned to
35106 indicate the error.

35107 If successful, the *pthread_rwlock_wrlock*( ) function shall return zero; otherwise, an error number
35108 shall be returned to indicate the error.

35109 **ERRORS**

35110 The *pthread_rwlock_trywrlock*( ) function shall fail if:

35111 [EBUSY] The read-write lock could not be acquired for writing because it was already
35112 locked for reading or writing.

35113 The *pthread_rwlock_trywrlock*( ) and *pthread_rwlock_wrlock*( ) functions may fail if:

35114 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock
35115 object.

35116 The *pthread_rwlock_wrlock*( ) function may fail if:

35117 [EDEADLK] The current thread already owns the read-write lock for writing or reading.

35118 These functions shall not return an error code of [EINTR].

35119 **EXAMPLES**

35120 None.

35121 **APPLICATION USAGE**

35122 Applications using these functions may be subject to priority inversion, as discussed in the Base |
35123 Definitions volume of IEEE Std. 1003.1-200x, Section 3.287, Priority Inversion. |

35124 **RATIONALE**

35125 None.

35126 **FUTURE DIRECTIONS**

35127 None.

35128 **SEE ALSO**

35129 *pthread_rwlock_destroy*( ), *pthread_rwlock_init*( ), *pthread_rwlock_rdlock*( ),
35130 *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_tryrdlock*( ),
35131 *pthread_rwlock_unlock*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<pthread.h>** |

35132 **CHANGE HISTORY**

35133 First released in Issue 5.

35134 **Issue 6**

35135 The following changes are made for alignment with IEEE Std. 1003.1j-2000:

35136 • The margin code in the SYNOPSIS is changed to RWL.

35137 • The [EDEADLK] error is deleted as a *pthread_rwlock_trywrlock*( ) error.

35138 • The SEE ALSO section is updated.

35139 **NAME**

35140 pthread_rwlock_unlock — unlock a read-write lock object

35141 **SYNOPSIS**

35142 THR
```
#include <pthread.h>

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```
35143

35144

35145 **DESCRIPTION**

35146 The *pthread_rwlock_unlock*( ) function is called to release a lock held on the read-write lock object
35147 referenced by *rwlock*. Results are undefined if the read-write lock *rwlock* is not held by the
35148 calling thread.

35149 If this function is called to release a read lock from the read-write lock object and there are other
35150 read locks currently held on this read-write lock object, the read-write lock object remains in the
35151 read locked state. If this function releases the last read lock for this read-write lock object, the
35152 read-write lock object shall be put in the unlocked state with no owners.

35153 If this function is called to release a write lock for this read-write lock object, the read-write lock
35154 object shall be put in the unlocked state.

35155 If there are threads blocked on the lock when it becomes available, the scheduling policy is used
35156 TPS to determine which thread(s) shall acquire the lock. If the Thread Execution Scheduling option is
35157 supported, when threads executing with the scheduling policies SCHED_FIFO, SCHED_RR, or
35158 SCHED_SPORADIC are waiting on the lock, they will acquire the lock in priority order when
35159 the lock becomes available. For equal priority threads, write locks take precedence over read
35160 locks. If the Thread Execution Scheduling option is not supported, it is implementation-defined |
35161 whether write locks take precedence over read locks. |

35162 Results are undefined if any of these functions are called with an uninitialized read-write lock.

35163 **RETURN VALUE**

35164 If successful, the *pthread_rwlock_unlock*( ) function shall return zero; otherwise, an error number
35165 shall be returned to indicate the error.

35166 **ERRORS**

35167 The *pthread_rwlock_unlock*( ) function may fail if:

35168 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock |
35169 object.

35170 [EPERM] The current thread does not hold a lock on the read-write lock. |

35171 The *pthread_rwlock_unlock*( ) function shall not return an error code of [EINTR]. |

35172 **EXAMPLES**

35173 None.

35174 **APPLICATION USAGE**

35175 None.

35176 **RATIONALE**

35177 None.

35178 **FUTURE DIRECTIONS**

35179 None.

35180 **SEE ALSO**

35181 *pthread_rwlock_destroy*( ), *pthread_rwlock_init*( ), *pthread_rwlock_rdlock*( ),
35182 *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_tryrdlock*( ),
35183 *pthread_rwlock_trywrlock*( ), *pthread_rwlock_wrlock*( ), the Base Definitions volume of |
35184 IEEE Std. 1003.1-200x, **<pthread.h>** |

35185 **CHANGE HISTORY**

35186 First released in Issue 5.

35187 **Issue 6**

35188 The following changes are made for alignment with IEEE Std. 1003.1j-2000:

35189 • The margin code in the SYNOPSIS is changed to RWL.

35190 • The DESCRIPTION is updated as follows:

35191 — The conditions under which writers have precedence over readers are specified.

35192 — The concept of read-write lock owner is deleted.

35193 • The SEE ALSO section is updated.

35194 **NAME**

35195 pthread_rwlock_wrlock — lock a read-write lock object for writing

35196 **SYNOPSIS**

35197 THR
```
#include <pthread.h>
```

35198
```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

35199

35200 **DESCRIPTION**

35201 Refer to *pthread_rwlock_trywrlock*( ).

35202 **NAME**

35203 pthread_rwlockattr_destroy, pthread_rwlockattr_init — destroy and initialize read-write lock
35204 attributes object

35205 **SYNOPSIS**

35206 THR
```
#include <pthread.h>

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```
35209

35210 **DESCRIPTION**

35211 The *pthread_rwlockattr_destroy*( ) function shall destroy a read-write lock attributes object. The
35212 effect of subsequent use of the object is undefined until the object is re-initialized by another call
35213 to *pthread_rwlockattr_init*( ). An implementation may cause *pthread_rwlockattr_destroy*( ) to set
35214 the object referenced by *attr* to an invalid value.

35215 The *pthread_rwlockattr_init*( ) function shall initialize a read-write lock attributes object *attr* with
35216 the default value for all of the attributes defined by the implementation.

35217 Results are undefined if *pthread_rwlockattr_init*( ) is called specifying an already initialized read-
35218 write lock attributes object.

35219 After a read-write lock attributes object has been used to initialize one or more read-write locks,
35220 any function affecting the attributes object (including destruction) does not affect any previously
35221 initialized read-write locks.

35222 **RETURN VALUE**

35223 If successful, the *pthread_rwlockattr_destroy*( ) and *pthread_rwlockattr_init*( ) functions shall return
35224 zero; otherwise, an error number shall be returned to indicate the error.

35225 **ERRORS**

35226 The *pthread_rwlockattr_destroy*( ) function may fail if:

35227 [EINVAL] The value specified by *attr* is invalid.

35228 The *pthread_rwlockattr_init*( ) function shall fail if:

35229 [ENOMEM] Insufficient memory exists to initialize the read-write lock attributes object.

35230 These functions shall not return an error code of [EINTR].

35231 **EXAMPLES**

35232 None.

35233 **APPLICATION USAGE**

35234 None.

35235 **RATIONALE**

35236 None.

35237 **FUTURE DIRECTIONS**

35238 None.

35239 **SEE ALSO**

35240 *pthread_rwlock_init*( ), *pthread_rwlockattr_getpshared*( ), *pthread_rwlockattr_setpshared*( ), the Base
35241 Definitions volume of IEEE Std. 1003.1-200x, **<pthread.h>**

35242 **CHANGE HISTORY**

35243 First released in Issue 5.

35244 **Issue 6**

35245 The following changes are made for alignment with IEEE Std. 1003.1j-2000:

35246 • The margin code in the SYNOPSIS is changed to RWL.

35247 • The SEE ALSO section is updated.

35248 **NAME**

35249 pthread_rwlockattr_getpshared, pthread_rwlockattr_setpshared — get and set process-shared
35250 attribute of read-write lock attributes object

35251 **SYNOPSIS**

35252 THR TSH
```
#include <pthread.h>

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *restrict attr,
    int *restrict pshared);
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
    int pshared);
```
35257

35258 **DESCRIPTION**

35259 The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a read-write lock
35260 to be operated upon by any thread that has access to the memory where the read-write lock is
35261 allocated, even if the read-write lock is allocated in memory that is shared by multiple processes.
35262 If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the read-write lock shall only
35263 be operated upon by threads created within the same process as the thread that initialized the
35264 read-write lock; if threads of differing processes attempt to operate on such a read-write lock,
35265 the behavior is undefined. The default value of the *process-shared* attribute is
35266 PTHREAD_PROCESS_PRIVATE.

35267 The *pthread_rwlockattr_getpshared*( ) function obtains the value of the *process-shared* attribute from
35268 the initialized attributes object referenced by *attr*. The *pthread_rwlockattr_setpshared*( ) function is
35269 used to set the *process-shared* attribute in an initialized attributes object referenced by *attr*.

35270 Additional attributes, their default values, and the names of the associated functions to get and
35271 set those attribute values are implementation-defined.

35272 **RETURN VALUE**

35273 Upon successful completion, the *pthread_rwlockattr_getpshared*( ) shall return zero and store the
35274 value of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter.
35275 Otherwise, an error number shall be returned to indicate the error.

35276 If successful, the *pthread_rwlockattr_setpshared*( ) function shall return zero; otherwise, an error
35277 number shall be returned to indicate the error.

35278 **ERRORS**

35279 The *pthread_rwlockattr_getpshared*( ) and *pthread_rwlockattr_setpshared*( ) functions may fail if:

35280 [EINVAL] The value specified by *attr* is invalid.

35281 The *pthread_rwlockattr_setpshared*( ) function may fail if:

35282 [EINVAL] The new value specified for the attribute is outside the range of legal values
35283 for that attribute.

35284 These functions shall not return an error code of [EINTR].

35285 **EXAMPLES**

35286 None.

35287 **APPLICATION USAGE**

35288 None.

35289 **RATIONALE**

35290 None.

35291 **FUTURE DIRECTIONS**

35292 None.

35293 **SEE ALSO**

35294 *pthread_rwlock_init*( ), *pthread_rwlockattr_destroy*( ), *pthread_rwlockattr_init*( ), the Base Definitions |
35295 volume of IEEE Std. 1003.1-200x, **<pthread.h>** |

35296 **CHANGE HISTORY**

35297 First released in Issue 5.

35298 **Issue 6**

35299 The following changes are made for alignment with IEEE Std. 1003.1j-2000:

35300 • The margin code in the SYNOPSIS is changed to RWL TSH.

35301 • The DESCRIPTION notes that additional attributes are implementation-defined. |

35302 • The SEE ALSO section is updated.

35303 The **restrict** keyword is added to the *pthread_rwlockattr_getpshared*( ) prototype for alignment |
35304 with the ISO / IEC 9899: 1999 standard. |

35305 **NAME**

35306 pthread_rwlockattr_init — initialize read-write lock attributes object

35307 **SYNOPSIS**

35308 XSI
```
#include <pthread.h>
```

35309
```
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

35310

35311 **DESCRIPTION**

35312 Refer to *pthread_rwlockattr_destroy*( ).

35313 **NAME**

35314 pthread_rwlockattr_setpshared — set process-shared attribute of read-write lock attributes
35315 object

35316 **SYNOPSIS**

35317 XSI
```
#include <pthread.h>

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
    int pshared);
```
35320

35321 **DESCRIPTION**

35322 Refer to *pthread_rwlockattr_getpshared*( ).

35323 **NAME**

35324 pthread_self — get calling thread's ID

35325 **SYNOPSIS**

35326 THR
```
#include <pthread.h>
```
35327
```
pthread_t pthread_self(void);
```
35328

35329 **DESCRIPTION**

35330 The *pthread_self*( ) function shall return the thread ID of the calling thread.

35331 **RETURN VALUE**

35332 Refer to the DESCRIPTION.

35333 **ERRORS**

35334 No errors are defined.

35335 The *pthread_self*( ) function shall not return an error code of [EINTR]. |

35336 **EXAMPLES**

35337 None.

35338 **APPLICATION USAGE**

35339 None.

35340 **RATIONALE**

35341 The *pthread_self*( ) function provides a capability similar to the *getpid*( ) function for processes
35342 and the rationale is the same: the creation call does not provide the thread ID to the created
35343 thread.

35344 **FUTURE DIRECTIONS**

35345 None.

35346 **SEE ALSO**

35347 *pthread_create*( ), *pthread_equal*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
35348 **<pthread.h>**

35349 **CHANGE HISTORY**

35350 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

35351 **Issue 6**

35352 The *pthread_self*( ) function is marked as part of the Threads option. |

35353 **NAME**

35354 pthread_setcancelstate, pthread_setcanceltype, pthread_testcancel — set cancelability state

35355 **SYNOPSIS**

35356 THR
```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```
35360

35361 **DESCRIPTION**

35362 The *pthread_setcancelstate*( ) function shall atomically both set the calling thread's cancelability
35363 state to the indicated *state* and return the previous cancelability state at the location referenced
35364 by *oldstate*. Legal values for *state* are PTHREAD_CANCEL_ENABLE and
35365 PTHREAD_CANCEL_DISABLE.

35366 The *pthread_setcanceltype*( ) function shall atomically both set the calling thread's cancelability
35367 type to the indicated *type* and return the previous cancelability type at the location referenced by
35368 *oldtype*. Legal values for *type* are PTHREAD_CANCEL_DEFERRED and
35369 PTHREAD_CANCEL_ASYNCHRONOUS.

35370 The cancelability state and type of any newly created threads, including the thread in which
35371 *main*( ) was first invoked, shall be PTHREAD_CANCEL_ENABLE and
35372 PTHREAD_CANCEL_DEFERRED respectively.

35373 The *pthread_testcancel*( ) function shall create a cancelation point in the calling thread. The
35374 *pthread_testcancel*( ) function shall have no effect if cancelability is disabled.

35375 **RETURN VALUE**

35376 If successful, the *pthread_setcancelstate*( ) and *pthread_setcanceltype*( ) functions shall return zero;
35377 otherwise, an error number shall be returned to indicate the error.

35378 **ERRORS**

35379 The *pthread_setcancelstate*( ) function may fail if:

35380 [EINVAL] The specified state is not PTHREAD_CANCEL_ENABLE or |
35381 PTHREAD_CANCEL_DISABLE.

35382 The *pthread_setcanceltype*( ) function may fail if:

35383 [EINVAL] The specified type is not PTHREAD_CANCEL_DEFERRED or
35384 PTHREAD_CANCEL_ASYNCHRONOUS.

35385 These functions shall not return an error code of [EINTR]. |

35386 **EXAMPLES**

35387 None.

35388 **APPLICATION USAGE**

35389 None.

35390 **RATIONALE**

35391 The *pthread_setcancelstate*( ) and *pthread_setcanceltype*( ) functions are used to control the points at
35392 which a thread may be asynchronously canceled. For cancelation control to be usable in modular
35393 fashion, some rules need to be followed.

35394 An object can be considered to be a generalization of a procedure. It is a set of procedures and
35395 global variables written as a unit and called by clients not known by the object. Objects may
35396 depend on other objects.

First, cancelability should only be disabled on entry to an object, never explicitly enabled. On exit from an object, the cancelability state should always be restored to its value on entry to the object.

This follows from a modularity argument: if the client of an object (or the client of an object that uses that object) has disabled cancelability, it is because the client does not want to be concerned about cleaning up if the thread is canceled while executing some sequence of actions. If an object is called in such a state and it enables cancelability and a cancelation request is pending for that thread, then the thread is canceled, contrary to the wish of the client that disabled.

Second, the cancelability type may be explicitly set to either *deferred* or *asynchronous* upon entry to an object. But as with the cancelability state, on exit from an object the cancelability type should always be restored to its value on entry to the object.

Finally, only functions that are cancel-safe may be called from a thread that is asynchronously cancelable.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cancel( ), 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_setcancelstate*( ), *pthread_setcanceltype*( ), and *pthread_testcancel*( ) functions are marked as part of the Threads option.

35419 **NAME**

35420 pthread_setconcurrency — set level of concurrency

35421 **SYNOPSIS**

35422 XSI
```
#include <pthread.h>
```

35423
```
int pthread_setconcurrency(int new_level);
```

35424

35425 **DESCRIPTION**

35426 Refer to *pthread_getconcurrency*( ).

35427 **NAME**

35428 pthread_setschedparam — dynamic thread scheduling parameters access (**REALTIME**
35429 **THREADS**)

35430 **SYNOPSIS**

35431 TPS
```
#include <pthread.h>
```

35432 35433
```
int pthread_setschedparam(pthread_t thread, int policy,
    const struct sched_param *param);
```

35434

35435 **DESCRIPTION**

35436 Refer to *pthread_getschedparam*( ).

35437 **NAME**

35438 pthread_setspecific — thread-specific data management

35439 **SYNOPSIS**

35440 THR
```
#include <pthread.h>
```

35441
```
int pthread_setspecific(pthread_key_t key, const void *value);
```

35442

35443 **DESCRIPTION**

35444 Refer to *pthread_getspecific*( ).

NAME

pthread_sigmask, sigprocmask — examine and change blocked signals

SYNOPSIS

```
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *restrict set,
    sigset_t *restrict oset);
int sigprocmask(int how, const sigset_t *restrict set,
    sigset_t *restrict oset);
```

DESCRIPTION

THR The *pthread_sigmask*( ) function is used to examine or change (or both) the calling thread's signal mask, regardless of the number of threads in the process. The effect shall be the same as described for *sigprocmask*( ), without the restriction that the call be made in a single-threaded process.

In a single-threaded process, the *sigprocmask*( ) function allows the calling process to examine or change (or both) the signal mask of the calling thread.

If the argument *set* is not a null pointer, it points to a set of signals to be used to change the currently blocked set.

The argument *how* indicates the way in which the set is changed, and the application shall ensure it consists of one of the following values:

SIG_BLOCK The resulting set shall be the union of the current set and the signal set pointed to by *set*.

SIG_SETMASK The resulting set shall be the signal set pointed to by *set*.

SIG_UNBLOCK The resulting set shall be the intersection of the current set and the complement of the signal set pointed to by *set*.

If the argument *oset* is not a null pointer, the previous mask is stored in the location pointed to by *oset*. If *set* is a null pointer, the value of the argument *how* is not significant and the process' signal mask is unchanged; thus the call can be used to enquire about currently blocked signals.

If there are any pending unblocked signals after the call to *sigprocmask*( ), at least one of those signals shall be delivered before the call to *sigprocmask*( ) returns.

It is not possible to block those signals which cannot be ignored. This shall be enforced by the system without causing an error to be indicated.

If any of the SIGFPE, SIGILL, SIGSEGV, or SIGBUS signals are generated while they are blocked, the result is undefined, unless the signal was generated by the *kill*( ) function, the *sigqueue*( ) function, or the *raise*( ) function.

If *sigprocmask*( ) fails, the thread's signal mask is not changed.

The use of the *sigprocmask*( ) function is unspecified in a multi-threaded process.

RETURN VALUE

THR Upon successful completion *pthread_sigmask*( ) shall return 0; otherwise, it shall return the corresponding error number.

Upon successful completion, *sigprocmask*( ) shall return 0; otherwise, –1 shall be returned, *errno* shall be set to indicate the error, and the process' signal mask shall be unchanged.

35486 **ERRORS**

35487 THR The *pthread_sigmask*( ) and *sigprocmask*( ) functions shall fail if:

35488 [EINVAL] The value of the *how* argument is not equal to one of the defined values.

35489 THR The *pthread_sigmask*( ) function shall not return an error code of [EINTR].

35490 **EXAMPLES**

35491 None.

35492 **APPLICATION USAGE**

35493 None.

35494 **RATIONALE**

35495 When a process' signal mask is changed in a signal-catching function that is installed by
35496 *sigaction*( ), the restoration of the signal mask on return from the signal-catching function
35497 overrides that change (see *sigaction*( )). If the signal-catching function was installed with
35498 *signal*( ), it is unspecified whether this occurs.

35499 See *kill*( ) for a discussion of the requirement on delivery of signals.

35500 **FUTURE DIRECTIONS**

35501 None.

35502 **SEE ALSO**

35503 *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ), *sigpending*( ),
35504 *sigqueue*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<signal.h>**

35505 **CHANGE HISTORY**

35506 First released in Issue 3.

35507 Entry included for alignment with the POSIX.1-1988 standard.

35508 **Issue 4**

35509 The DESCRIPTION is changed to indicate that signals can also be generated by *raise*( ).

35510 The following change is incorporated for alignment with the ISO POSIX-1 standard:

35511 • The type of the arguments *set* and *oset* are changed from **sigset_t*** to **const sigset_t***.

35512 **Issue 5**

35513 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

35514 The *pthread_sigmask*( ) function is added for alignment with the POSIX Threads Extension.

35515 **Issue 6**

35516 The *pthread_sigmask*( ) function is marked as part of the Threads option.

35517 The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

35518 • The DESCRIPTION is updated to explicitly state the functions which may generate the
35519 signal.

35520 The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

35521 The **restrict** keyword is added to the *pthread_sigmask*( ) and *sigprocmask*( ) prototypes for
35522 alignment with the ISO / IEC 9899: 1999 standard.

NAME

pthread_spin_destroy, pthread_spin_init — destroy or initialize a spin lock object

SYNOPSIS

SPI

```
#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 destroys the spin lock referenced by *lock* and releases 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 allocates any resources required to use the spin lock referenced by *lock* and initializes 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 operated upon by threads created within the same process as the thread that initialized the spin lock. If threads of differing processes attempt to operate on such a spin lock, the behavior is undefined.

The results are undefined if *pthread_spin_init*( ) is called specifying an already initialized spin lock. The results are undefined if a spin lock is used without first being initialized.

If the *pthread_spin_init*( ) function fails, the lock is not initialized and the contents of *lock* are undefined.

Only the object referenced by *lock* may be used for performing synchronization.

The result of referring to copies of that object in calls to *pthread_spin_destroy*( ), *pthread_spin_lock*( ), *pthread_spin_trylock*( ), or *pthread_spin_unlock*( ) is undefined.

RETURN VALUE

Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EBUSY] The implementation has detected an attempt to initialize or destroy a spin lock while it is in use (for example, while being used in a *pthread_spin_lock*( ) call) by another thread.

[EINVAL] The value specified by *lock* is invalid.

The *pthread_spin_init*( ) function shall fail if:

[EAGAIN] The system lacks the necessary resources to initialize another spin lock.

[ENOMEM] Insufficient memory exists to initialize the lock.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_spin_destroy*( ) and *pthread_spin_init*( ) functions are part of the Spin Locks option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_spin_lock( ), *pthread_spin_trylock*( ), *pthread_spin_unlock*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<<pthread.h>>**

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.

35582 **NAME**

35583 pthread_spin_init — initialize a spin lock object

35584 **SYNOPSIS**

35585 SPI
```
#include <pthread.h>
```

35586
```
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```

35587

35588 **DESCRIPTION**

35589 Refer to *pthread_spin_destroy*( ).

35590 **NAME**

35591 pthread_spin_lock, pthread_spin_trylock — lock a spin lock object

35592 **SYNOPSIS**

35593 SPI
```
#include <pthread.h>

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
```
35596

35597 **DESCRIPTION**

35598 The *pthread_spin_lock*( ) function locks the spin lock referenced by *lock*. The calling thread
35599 acquires the lock if it is not held by another thread. Otherwise, the thread spins (that is, does not
35600 return from the *pthread_spin_lock*( ) call) until the lock becomes available. The results are
35601 undefined if the calling thread holds the lock at the time the call is made. The
35602 *pthread_spin_trylock*( ) function locks the spin lock referenced by *lock* if it is not held by any
35603 thread. Otherwise, the function fails.

35604 The results are undefined if any of these functions is called with an uninitialized spin lock.

35605 **RETURN VALUE**

35606 Upon successful completion, these functions shall return zero; otherwise, an error number shall
35607 be returned to indicate the error.

35608 **ERRORS**

35609 These functions may fail if:

35610 [EINVAL] The value specified by *lock* does not refer to an initialized spin lock object.

35611 The *pthread_spin_lock*( ) function may fail if:

35612 [EDEADLK] The calling thread already holds the lock.

35613 The *pthread_spin_trylock*( ) function shall fail if:

35614 [EBUSY] A thread currently holds the lock.

35615 These functions shall not return an error code of [EINTR]. |

35616 **EXAMPLES**

35617 None.

35618 **APPLICATION USAGE**

35619 Applications using this function may be subject to priority inversion, as discussed in the Base |
35620 Definitions volume of IEEE Std. 1003.1-200x, Section 3.287, Priority Inversion. |

35621 The *pthread_spin_lock*( ) and *pthread_spin_trylock*( ) functions are part of the Spin Locks option |
35622 and need not be provided on all implementations. |

35623 **RATIONALE**

35624 None.

35625 **FUTURE DIRECTIONS**

35626 None.

35627 **SEE ALSO**

35628 *pthread_spin_init*( ), *pthread_spin_destroy*( ), *pthread_spin_unlock*( ), the Base Definitions volume of |
35629 IEEE Std. 1003.1-200x, **<pthread.h>** |

35630 **CHANGE HISTORY**

35631 First released in Issue 6. Derived from IEEE Std. 1003.1j-2000.

35632 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

35633 **NAME**

35634 pthread_spin_trylock — lock a spin lock object

35635 **SYNOPSIS**

35636 SPI
```
#include <pthread.h>
```

35637
```
int pthread_spin_trylock(pthread_spinlock_t *lock);
```

35638

35639 **DESCRIPTION**

35640 Refer to *pthread_spin_lock*( ).

NAME

pthread_spin_unlock — unlock a spin lock object

SYNOPSIS

SPI

```
#include <pthread.h>

int pthread_spin_unlock(pthread_spinlock_t *lock);
```

DESCRIPTION

The *pthread_spin_unlock*( ) function releases the spin lock referenced by *lock* which was locked via the *pthread_spin_lock*( ) or *pthread_spin_trylock*( ) functions. The results are undefined if the lock is not held by the calling thread. If there are threads spinning on the lock when *pthread_spin_unlock*( ) is called, the lock becomes available and an unspecified spinning thread shall acquire the lock.

The results are undefined if this function is called with an uninitialized thread spin lock.

RETURN VALUE

Upon successful completion, the *pthread_spin_unlock*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_spin_unlock*( ) function may fail if:

[EINVAL] An invalid argument was specified.

[EPERM] The calling thread does not hold the lock.

This function shall not return an error code of [EINTR]. |

EXAMPLES

None.

APPLICATION USAGE

The *pthread_spin_unlock*( ) function is part of the Spin Locks option and need not be provided on all implementations. |

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_spin_init( ), *pthread_spin_destroy*( ), *pthread_spin_lock*( ), *pthread_spin_trylock*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<pthread.h>** |

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.

35677 **NAME**

35678 pthread_testcancel — set cancelability state

35679 **SYNOPSIS**

35680 THR
```
#include <pthread.h>
```

35681
```
void pthread_testcancel(void);
```

35682

35683 **DESCRIPTION**

35684 Refer to *pthread_setcancelstate*( ).

35685 **NAME**

35686 ptsname — get name of the slave pseudo-terminal device

35687 **SYNOPSIS**

35688 XSI
```
#include <stdlib.h>
```

35689
```
char *ptsname(int fildes);
```

35690

35691 **DESCRIPTION**

35692 The *ptsname*( ) function shall return the name of the slave pseudo-terminal device associated
35693 with a master pseudo-terminal device. The *fildes* argument is a file descriptor that refers to the
35694 master device. The *ptsname*( ) function shall return a pointer to a string containing the path name
35695 of the corresponding slave device.

35696 The *ptsname*( ) function need not be reentrant. A function that is not required to be reentrant is
35697 not required to be thread-safe.

35698 **RETURN VALUE**

35699 Upon successful completion, *ptsname*( ) shall return a pointer to a string which is the name of the
35700 pseudo-terminal slave device. Upon failure, *ptsname*( ) shall return a null pointer. This could
35701 occur if *fildes* is an invalid file descriptor or if the slave device name does not exist in the file
35702 system.

35703 **ERRORS**

35704 No errors are defined.

35705 **EXAMPLES**

35706 None.

35707 **APPLICATION USAGE**

35708 The value returned may point to a static data area that is overwritten by each call to *ptsname*( ).

35709 **RATIONALE**

35710 None.

35711 **FUTURE DIRECTIONS**

35712 None.

35713 **SEE ALSO**

35714 *grantpt*( ), *open*( ), *ttyname*( ), *unlockpt*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
35715 **<stdlib.h>**

35716 **CHANGE HISTORY**

35717 First released in Issue 4, Version 2.

35718 **Issue 5**

35719 Moved from X/OPEN UNIX extension to BASE.

35720 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

35721 **NAME**

35722 putc — put byte on a stream

35723 **SYNOPSIS**

35724 `#include <stdio.h>`

35725 `int putc(int c, FILE *stream);`

35726 **DESCRIPTION**

35727 CX The functionality described on this reference page is aligned with the ISO C standard. Any
35728 conflict between the requirements described here and the ISO C standard is unintentional. This
35729 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

35730 The *putc*( ) function shall be equivalent to *fputc*( ), except that if it is implemented as a macro it
35731 may evaluate *stream* more than once, so the argument should never be an expression with side
35732 effects.

35733 **RETURN VALUE**

35734 Refer to *fputc*( ).

35735 **ERRORS**

35736 Refer to *fputc*( ).

35737 **EXAMPLES**

35738 None.

35739 **APPLICATION USAGE**

35740 Because it may be implemented as a macro, *putc*( ) may treat a *stream* argument with side effects
35741 incorrectly. In particular, *putc*(*c*,*f*++) does not necessarily work correctly. Therefore, use of this
35742 function is not recommended in such situations; *fputc*( ) should be used instead.

35743 **RATIONALE**

35744 None.

35745 **FUTURE DIRECTIONS**

35746 None.

35747 **SEE ALSO**

35748 *fputc*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>** |

35749 **CHANGE HISTORY**

35750 First released in Issue 1. Derived from Issue 1 of the SVID. |

35751 **Issue 4**

35752 The APPLICATION USAGE section now states that the use of this function is not recommended
35753 with a *stream* argument with side effects.

35754 The following change is incorporated for alignment with the ISO C standard:

35755 - The *c* argument is not allowed to be evaluated more than once.

35756 **NAME**

35757 putc_unlocked — stdio with explicit client locking

35758 **SYNOPSIS**

35759 TSF
```
#include <stdio.h>
```

35760
```
int putc_unlocked(int c, FILE *stream);
```

35761

35762 **DESCRIPTION**

35763 Refer to *getc_unlocked*( ).

35764 **NAME**

35765 putchar — put byte on stdout stream

35766 **SYNOPSIS**

35767 `#include <stdio.h>`

35768 `int putchar(int c);`

35769 **DESCRIPTION**

35770 CX The functionality described on this reference page is aligned with the ISO C standard. Any
35771 conflict between the requirements described here and the ISO C standard is unintentional. This
35772 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

35773 The function call *putchar*(*c*) shall be equivalent to *putc*(*c*,*stdout*).

35774 **RETURN VALUE**

35775 Refer to *fputc*( ).

35776 **ERRORS**

35777 Refer to *fputc*( ).

35778 **EXAMPLES**

35779 None.

35780 **APPLICATION USAGE**

35781 None.

35782 **RATIONALE**

35783 None.

35784 **FUTURE DIRECTIONS**

35785 None.

35786 **SEE ALSO**

35787 *putc*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>** |

35788 **CHANGE HISTORY**

35789 First released in Issue 1. Derived from Issue 1 of the SVID. |

35790 **NAME**

35791 putchar_unlocked — stdio with explicit client locking

35792 **SYNOPSIS**

35793 TSF
```
#include <stdio.h>
```

35794
```
int putchar_unlocked(int c);
```

35795

35796 **DESCRIPTION**

35797 Refer to *getc_unlocked*( ).

35798 **NAME**

35799 putenv — change or add a value to environment

35800 **SYNOPSIS**

35801 XSI
```
#include <stdlib.h>
```

35802
```
int putenv(char *string);
```

35803

35804 **DESCRIPTION**

35805 The *putenv*( ) function uses the *string* argument to set environment variable values. The *string*
35806 argument should point to a string of the form "*name=value*. The *putenv*( ) function makes the
35807 value of the environment variable *name* equal to *value* by altering an existing variable or creating
35808 a new one. In either case, the string pointed to by *string* becomes part of the environment, so
35809 altering the string shall change the environment. The space used by *string* is no longer used once
35810 a new string-defining *name* is passed to *putenv*( ).

35811 The *putenv*( ) function need not be reentrant. A function that is not required to be reentrant is not
35812 required to be thread-safe.

35813 **RETURN VALUE**

35814 Upon successful completion, *putenv*( ) shall return 0; otherwise, it shall return a non-zero value
35815 and set *errno* to indicate the error.

35816 **ERRORS**

35817 The *putenv*( ) function may fail if:

35818 [ENOMEM] Insufficient memory was available. |

35819 **EXAMPLES**

35820 **Changing the Value of an Environment Variable**

35821 The following example changes the value of the *HOME* environment variable to the value
35822 **/usr/home**.

35823
```
#include <stdlib.h>
```
35824
```
...
```
35825
```
static char *var = "HOME=/usr/home";
```
35826
```
int ret;
```

35827
```
ret = putenv(var);
```

35828 **APPLICATION USAGE**

35829 The *putenv*( ) function manipulates the environment pointed to by *environ*, and can be used in
35830 conjunction with *getenv*( ).

35831 This routine may use *malloc*( ) to enlarge the environment.

35832 A potential error is to call *putenv*( ) with an automatic variable as the argument, then return from
35833 the calling function while *string* is still part of the environment. |

35834 The *setenv*( ) function is preferred over this function. |

35835 **RATIONALE**

35836 None.

35837 **FUTURE DIRECTIONS**

35838 None.

35839 **SEE ALSO**

35840 *exec*, *getenv*( ), *malloc*( ), *setenv*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>**

35841 **CHANGE HISTORY**

35842 First released in Issue 1. Derived from Issue 1 of the SVID.

35843 **Issue 4**

35844 The **<stdlib.h>** header is added to the SYNOPSIS section.

35845 The type of argument *string* is changed from **char*** to **const char***.

35846 **Issue 5**

35847 The type of the argument to this function is changed from **const char*** to **char***. This was
35848 indicated as a FUTURE DIRECTION in previous issues.

35849 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

35850 **NAME**

35851 putmsg, putpmsg — send a message on a STREAM (**STREAMS**)

35852 **SYNOPSIS**

35853 XSR
```
#include <stropts.h>

int putmsg(int fildes, const struct strbuf *ctlptr,
    const struct strbuf *dataptr, int flags);
int putpmsg(int fildes, const struct strbuf *ctlptr,
    const struct strbuf *dataptr, int band, int flags);
```
35858

35859 **DESCRIPTION**

35860 The *putmsg*( ) function shall create a message from a process buffer(s) and send the message to a
35861 STREAMS file. The message may contain either a data part, a control part, or both. The data and
35862 control parts are distinguished by placement in separate buffers, as described below. The
35863 semantics of each part are defined by the STREAMS module that receives the message.

35864 The *putpmsg*( ) function does the same thing as *putmsg*( ), but the process can send messages in
35865 different priority bands. Except where noted, all requirements on *putmsg*( ) also pertain to
35866 *putpmsg*( ).

35867 The *fildes* argument specifies a file descriptor referencing an open STREAM. The *ctlptr* and
35868 *dataptr* arguments each point to a **strbuf** structure.

35869 The *ctlptr* argument points to the structure describing the control part, if any, to be included in
35870 the message. The *buf* member in the **strbuf** structure points to the buffer where the control
35871 information resides, and the *len* member indicates the number of bytes to be sent. The *maxlen*
35872 member is not used by *putmsg*( ). In a similar manner, the argument *dataptr* specifies the data, if
35873 any, to be included in the message. The *flags* argument indicates what type of message should be
35874 sent and is described further below.

35875 To send the data part of a message, the application shall ensure that *dataptr* is not a null pointer
35876 and the *len* member of *dataptr* is 0 or greater. To send the control part of a message, the
35877 application shall ensure that the corresponding values are set for *ctlptr*. No data (control) part
35878 shall be sent if either *dataptr*(*ctlptr*) is a null pointer or the *len* member of *dataptr*(*ctlptr*) is set to
35879 –1.

35880 For *putmsg*( ), if a control part is specified and *flags* is set to RS_HIPRI, a high priority message is
35881 sent. If no control part is specified, and *flags* is set to RS_HIPRI, *putmsg*( ) fails and sets *errno* to
35882 [EINVAL]. If *flags* is set to 0, a normal message (priority band equal to 0) is sent. If a control part
35883 and data part are not specified and *flags* is set to 0, no message is sent and 0 is returned.

35884 For *putpmsg*( ), the flags are different. The *flags* argument is a bitmask with the following
35885 mutually-exclusive flags defined: MSG_HIPRI and MSG_BAND. If *flags* is set to 0, *putpmsg*( )
35886 fails and sets *errno* to [EINVAL]. If a control part is specified and *flags* is set to MSG_HIPRI and
35887 *band* is set to 0, a high-priority message is sent. If *flags* is set to MSG_HIPRI and either no control
35888 part is specified or *band* is set to a non-zero value, *putpmsg*( ) fails and sets *errno* to [EINVAL]. If
35889 *flags* is set to MSG_BAND, then a message is sent in the priority band specified by *band*. If a
35890 control part and data part are not specified and *flags* is set to MSG_BAND, no message is sent
35891 and 0 is returned.

35892 The *putmsg*( ) function blocks if the STREAM write queue is full due to internal flow control
35893 conditions, with the following exceptions:

35894 • For high-priority messages, *putmsg*( ) does not block on this condition and continues
35895 processing the message.

- For other messages, *putmsg*( ) does not block but fails when the write queue is full and O_NONBLOCK is set.

The *putmsg*( ) function also blocks, unless prevented by lack of internal resources, while waiting for the availability of message blocks in the STREAM, regardless of priority or whether O_NONBLOCK has been specified. No partial message is sent.

RETURN VALUE

Upon successful completion, *putmsg*( ) and *putpmsg*( ) shall return 0; otherwise, they shall return –1 and set *errno* to indicate the error.

ERRORS

The *putmsg*( ) and *putpmsg*( ) functions shall fail if:

[EAGAIN] A non-priority message was specified, the O_NONBLOCK flag is set, and the STREAM write queue is full due to internal flow control conditions; or buffers could not be allocated for the message that was to be created.

[EBADF] *fildes* is not a valid file descriptor open for writing.

[EINTR] A signal was caught during *putmsg*( ).

[EINVAL] An undefined value is specified in *flags*, or *flags* is set to RS_HIPRI or MSG_HIPRI and no control part is supplied, or the STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer, or *flags* is set to MSG_HIPRI and *band* is non-zero (for *putpmsg*( ) only).

[ENOSR] Buffers could not be allocated for the message that was to be created due to insufficient STREAMS memory resources.

[ENOSTR] A STREAM is not associated with *fildes*.

[ENXIO] A hangup condition was generated downstream for the specified STREAM.

[EPIPE] or [EIO] The *fildes* argument refers to a STREAMS-based pipe and the other end of the pipe is closed. A SIGPIPE signal is generated for the calling thread.

[ERANGE] The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost STREAM module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of a message is larger than the maximum configured size of the data part of a message.

In addition, *putmsg*( ) and *putpmsg*( ) shall fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of *errno* does not reflect the result of *putmsg*( ) or *putpmsg*( ), but reflects the prior error.

EXAMPLES

Sending a High-Priority Message

The value of *fd* is assumed to refer to an open STREAMS file. This call to *putmsg*( ) does the following:

1. Creates a high-priority message with a control part and a data part, using the buffers pointed to by *ctrlbuf* and *databuf*, respectively.
2. Sends the message to the STREAMS file identified by *fd*.

```
#include <stropts.h>
#include <string.h>
...
int fd;
char *ctrlbuf = "This is the control part";
char *databuf = "This is the data part";
struct strbuf ctrl;
struct strbuf data;
int ret;

ctrl.buf = ctrlbuf;
ctrl.len = strlen(ctrlbuf);

data.buf = databuf;
data.len = strlen(databuf);

ret = putmsg(fd, &ctrl, &data, MSG_HIPRI);
```

Using putpmsg()

This example has the same effect as the previous example. In this example, however, the *putpmsg*( ) function is used to create and send the message to the STREAMS file.

```
#include <stropts.h>
#include <string.h>
...
int fd;
char *ctrlbuf = "This is the control part";
char *databuf = "This is the data part";
struct strbuf ctrl;
struct strbuf data;
int ret;

ctrl.buf = ctrlbuf;
ctrl.len = strlen(ctrlbuf);

data.buf = databuf;
data.len = strlen(databuf);

ret = putpmsg(fd, &ctrl, &data, 0, MSG_HIPRI);
```

APPLICATION USAGE

None.

35971 **RATIONALE**

35972 None.

35973 **FUTURE DIRECTIONS**

35974 None.

35975 **SEE ALSO**

35976 *getmsg*( ), *poll*( ), *read*( ), *write*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
35977 **<stropts.h>**, Section 2.6 (on page 539)

35978 **CHANGE HISTORY**

35979 First released in Issue 4, Version 2.

35980 **Issue 5**

35981 Moved from X/OPEN UNIX extension to BASE.

35982 The following text is removed from the DESCRIPTION: ‘‘The STREAM head guarantees that the
35983 control part of a message generated by *putmsg*( ) is at least 64 bytes in length’’.

35984 **Issue 6**

35985 This function is marked as part of the XSI STREAMS Option Group.

35986 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

35987 **NAME**

35988 puts — put a string on standard output

35989 **SYNOPSIS**

35990
```
#include <stdio.h>
```

35991
```
int puts(const char *s);
```

35992 **DESCRIPTION**

35993 CX The functionality described on this reference page is aligned with the ISO C standard. Any
35994 conflict between the requirements described here and the ISO C standard is unintentional. This
35995 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

35996 The *puts*( ) function shall write the string pointed to by *s*, followed by a <newline> character, to
35997 the standard output stream *stdout*. The terminating null byte shall not be written.

35998 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
35999 execution of *puts*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same
36000 stream or a call to *exit*( ) or *abort*( ).

36001 **RETURN VALUE**

36002 Upon successful completion, *puts*( ) shall return a non-negative number. Otherwise, it shall
36003 CX return EOF, shall set an error indicator for the stream, and *errno* shall be set to indicate the error.

36004 **ERRORS**

36005 Refer to *fputc*( ).

36006 **EXAMPLES**

36007 **Printing to Standard Output**

36008 The following example gets the current time, converts it to a string using *localtime*( ) and
36009 *asctime*( ), and prints it to standard output using *puts*( ). It then prints the number of minutes to
36010 an event for which it is waiting.

36011
```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
puts(asctime(localtime(&now)));
printf("There are %d minutes to the event.\n",
    minutes_to_event);
...
```

36023 **APPLICATION USAGE**

36024 The *puts*( ) function appends a <newline> character, while *fputs*( ) does not.

36025 **RATIONALE**

36026 None.

36027 **FUTURE DIRECTIONS**

36028 None.

36029 **SEE ALSO**

36030 *fopen*( ), *fputs*( ), *putc*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**

36031 **CHANGE HISTORY**

36032 First released in Issue 1. Derived from Issue 1 of the SVID.

36033 **Issue 4**

36034 In the DESCRIPTION, the words ‘‘null character’’ are replaced by ‘‘null byte’’.

36035 The following change is incorporated for alignment with the ISO C standard:

36036 • The type of argument *s* is changed from **char*** to **const char***.

36037 **Issue 6**

36038 Extensions beyond the ISO C standard are now marked.

36039 **NAME**

36040 pututxline — put an entry into user accounting database

36041 **SYNOPSIS**

36042 XSI
```
#include <utmpx.h>

struct utmpx *pututxline(const struct utmpx *utmpx);
```
36043

36044

36045 **DESCRIPTION**

36046 Refer to *endutxent*( ).

NAME

putwc — put a wide character on a stream

SYNOPSIS

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

wint_t putwc(wchar_t wc, 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 *putwc*( ) function shall be equivalent to *fputwc*( ), except that if it is implemented as a macro it may evaluate *stream* more than once, so the argument should never be an expression with side effects.

RETURN VALUE

Refer to *fputwc*( ).

ERRORS

Refer to *fputwc*( ).

EXAMPLES

None.

APPLICATION USAGE

Because it may be implemented as a macro, *putwc*( ) may treat a *stream* argument with side effects incorrectly. In particular, *putwc*(*wc*,*f*++) need not work correctly. Therefore, use of this function is not recommended; *fputwc*( ) should be used instead.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fputwc( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released as a World-wide Portability Interface in Issue 4.

Issue 5

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument *wc* is changed from **wint_t** to **wchar_t**.

The Optional Header (OH) marking is removed from **<stdio.h>**.

36082 **NAME**

36083 putwchar — put a wide character on stdout stream

36084 **SYNOPSIS**

36085 `#include <wchar.h>`

36086 `wint_t putwchar(wchar_t wc);`

36087 **DESCRIPTION**

36088 CX The functionality described on this reference page is aligned with the ISO C standard. Any
36089 conflict between the requirements described here and the ISO C standard is unintentional. This
36090 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

36091 The function call *putwchar*(*wc*) shall be equivalent to *putwc*(*wc*,*stdout*).

36092 **RETURN VALUE**

36093 Refer to *fputwc*( ).

36094 **ERRORS**

36095 Refer to *fputwc*( ).

36096 **EXAMPLES**

36097 None.

36098 **APPLICATION USAGE**

36099 None.

36100 **RATIONALE**

36101 None.

36102 **FUTURE DIRECTIONS**

36103 None.

36104 **SEE ALSO**

36105 *fputwc*( ), *putwc*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>**

36106 **CHANGE HISTORY**

36107 First released in Issue 4.

36108 **Issue 5**

36109 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument *wc*
36110 is changed from **wint_t** to **wchar_t**.

36111 **NAME**

36112 pwrite, — write on a file

36113 **SYNOPSIS**

36114 `#include <unistd.h>`

36115 XSI
```
ssize_t pwrite(int fildes, const void *buf, size_t nbyte,
    off_t offset);
```
36116

36117

36118 **DESCRIPTION**

36119 Refer to *write*( ).

36120 **NAME**

36121 qsort — sort a table of data

36122 **SYNOPSIS**

36123 `#include <stdlib.h>`

36124 `void qsort(void *`*`base`*`, size_t `*`nel`*`, size_t `*`width`*`,`
36125 `    int (*`*`compar`*`)(const void *, const void *));`

36126 **DESCRIPTION**

36127 CX The functionality described on this reference page is aligned with the ISO C standard. Any
36128 conflict between the requirements described here and the ISO C standard is unintentional. This
36129 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

36130 The *qsort*( ) function sorts an array of *nel* objects, the initial element of which is pointed to by
36131 *base*. The size of each object, in bytes, is specified by the *width* argument.

36132 The contents of the array are sorted in ascending order according to a comparison function. The
36133 *compar* argument is a pointer to the comparison function, which is called with two arguments
36134 that point to the elements being compared. The application shall ensure that the function returns
36135 an integer less than, equal to, or greater than 0, if the first argument is considered respectively
36136 less than, equal to, or greater than the second. If two members compare as equal, their order in
36137 the sorted array is unspecified.

36138 **RETURN VALUE**

36139 The *qsort*( ) function shall return no value.

36140 **ERRORS**

36141 No errors are defined.

36142 **EXAMPLES**

36143 None.

36144 **APPLICATION USAGE**

36145 The comparison function need not compare every byte, so arbitrary data may be contained in
36146 the elements in addition to the values being compared.

36147 **RATIONALE**

36148 None.

36149 **FUTURE DIRECTIONS**

36150 None.

36151 **SEE ALSO**

36152 The Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>**

36153 **CHANGE HISTORY**

36154 First released in Issue 1. Derived from Issue 1 of the SVID.

36155 **Issue 4**

36156 The following change is incorporated for alignment with the ISO C standard:

36157 • The arguments to *compar* are formally defined in the SYNOPSIS section.

36158 **Issue 6**

36159 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

36160 **NAME**

36161 raise — send a signal to the executing process

36162 **SYNOPSIS**

36163 `#include <signal.h>`

36164 `int raise(int sig);`

36165 **DESCRIPTION**

36166 CX The functionality described on this reference page is aligned with the ISO C standard. Any
36167 conflict between the requirements described here and the ISO C standard is unintentional. This
36168 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

36169 CX The *raise*( ) function shall send the signal *sig* to the executing thread or process. If a signal |
36170 handler is called, the *raise*( ) function shall not return until after the signal handler does. |

36171 THR If the implementation supports the Threads option, the effect of the *raise*( ) function is equivalent |
36172 to calling:

36173 `pthread_kill(pthread_self(), sig);`

36174 |

36175 CX Otherwise, the effect of the *raise*( ) function is equivalent to calling: |

36176 `kill(getpid(), sig);` |

36177 |

36178 **RETURN VALUE**

36179 CX Upon successful completion, 0 shall be returned. Otherwise, a non-zero value shall be returned
36180 and *errno* shall be set to indicate the error.

36181 **ERRORS**

36182 The *raise*( ) function shall fail if:

36183 CX [EINVAL] The value of the *sig* argument is an invalid signal number. |

36184 **EXAMPLES**

36185 None.

36186 **APPLICATION USAGE**

36187 None.

36188 **RATIONALE**

36189 The term ‘‘thread’’ is an extension to the ISO C standard.

36190 **FUTURE DIRECTIONS**

36191 None.

36192 **SEE ALSO**

36193 *kill*( ), *sigaction*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<signal.h>**, |
36194 **<sys/types.h>**

36195 **CHANGE HISTORY**

36196 First released in Issue 4. Derived from the ANSI C standard. |

36197 **Issue 5**

36198 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

Extensions beyond the ISO C standard are now 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* on error is added.
- The [EINVAL] error condition is added.

NAME

rand, rand_r, srand — pseudo-random number generator

SYNOPSIS

```
#include <stdlib.h>

int rand(void);
int rand_r(unsigned *seed);
void srand(unsigned seed);
```

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 *rand*( ) function shall compute a sequence of pseudo-random integers in the range 0 to XSI {RAND_MAX} with a period of at least 2^{32}.

CX The *rand*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *rand_r*( ) function shall compute a sequence of pseudo-random integers in the range 0 to {RAND_MAX}. (The value of the {RAND_MAX} macro shall be at least 32 767.)

If *rand_r*( ) is called with the same initial value for the object pointed to by *seed* and that object is not modified between successive returns and calls to *rand_r*( ), the same sequence shall be generated.

The *srand*( ) function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to *rand*( ). If *srand*( ) is then called with the same seed value, the sequence of pseudo-random numbers shall be repeated. If *rand*( ) is called before any calls to *srand*( ) are made, the same sequence shall be generated as when *srand*( ) is first called with a seed value of 1.

The implementation shall behave as if no function defined in this volume of IEEE Std. 1003.1-200x calls *rand*( ) or *srand*( ).

RETURN VALUE

The *rand*( ) function shall return the next pseudo-random number in the sequence.

TSF The *rand_r*( ) function shall return a pseudo-random integer.

The *srand*( ) function shall return no value.

ERRORS

No errors are defined.

EXAMPLES

Generating a Pseudo-Random Number Sequence

The following example demonstrates how to generate a sequence of pseudo-random numbers.

```
#include <stdio.h>
#include <stdlib.h>
...
    long count, i;
    char *keystr;
    int elementlen, len;
    char c;
```

```
...
/* Initial random number generator. */
    srand(1);

    /* Create keys using only lower case characters */
    len = 0;
    for (i=0; i<count; i++) {
        while (len < elementlen) {
            c = (char) (rand() % 128);
            if (islower(c))
                keystr[len++] = c;
        }

        keystr[len] = '\0';
        printf("%s Element%0*ld\n", keystr, elementlen, i);
        len = 0;
    }
```

Generating the Same Sequence on Different Machines

The following code defines a pair of functions that could be incorporated into applications wishing to ensure that the same sequence of numbers is generated across different machines.

```
static unsigned long next = 1;
int myrand(void)  /* RAND_MAX assumed to be 32767. */
{
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}

void mysrand(unsigned seed)
{
    next = seed;
}
```

APPLICATION USAGE

The *drand48*( ) function provides a much more elaborate random number generator.

RATIONALE

The ISO C standard *rand*( ) and *srand*( ) functions allow per-process pseudo-random streams shared by all threads. Those two functions need not change, but there has to be mutual-exclusion that prevents interference between two threads concurrently accessing the random number generator.

With regard to *rand*( ), there are two different behaviors that may be wanted in a multi-threaded program:

1. A single per-process sequence of pseudo-random numbers that is shared by all threads that call *rand*( )
2. A different sequence of pseudo-random numbers for each thread that calls *rand*( )

This is provided by the modified thread-safe function based on whether the seed value is global to the entire process or local to each thread.

This does not address the known deficiencies of the *rand*( ) function implementations, which have been approached by maintaining more state. In effect, this specifies new thread-safe forms of a deficient function. Since alternatives to *rand*( ) are not standardized, they are not modified as

part of this volume of IEEE Std. 1003.1-200x.

FUTURE DIRECTIONS

None.

SEE ALSO

drand48( ), 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 4

The definition of *srand*( ) is added to the SYNOPSIS section.

In the DESCRIPTION, the text referring to the period of pseudo-random numbers is marked as an extension.

The example in the APPLICATION USAGE section is updated as follows:

- To use ISO C standard syntax.
- To avoid name clashes with standard functions.

The following changes are incorporated for alignment with the ISO C standard:

- The argument list of *rand*( ) is explicitly defined as **void**.
- The argument *seed* is explicitly defined as **unsigned**.

Issue 5

The *rand_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *rand*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

Extensions beyond the ISO C standard are now marked.

The *rand_r*( ) function is marked as part of the Thread-Safe Functions option.

36316 **NAME**

36317 random — generate pseudorandom number

36318 **SYNOPSIS**

36319 XSI
```
#include <stdlib.h>
```

36320
```
long random(void);
```

36321

36322 **DESCRIPTION**

36323 Refer to *initstate*( ).

NAME

pread, read, readv — 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);

#include <sys/uio.h>

ssize_t readv(int fildes, const struct iovec *iov, int iovcnt);
```

DESCRIPTION

The *read*( ) function attempts 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.

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*( ) starts at a position in the file given by the file offset associated with *fildes*. The file offset is 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 greater than *nbyte*. The value returned may be less than *nbyte* if the number of bytes left in the file is less than *nbyte*, if the *read*( ) request was interrupted by a signal, or if the file is a pipe or FIFO or special file and has fewer than *nbyte* bytes immediately available for reading. For example, a *read*( ) from a file associated with a terminal may return one typed line of data.

If a *read*( ) is interrupted by a signal before it reads any data, it shall return –1 with *errno* set to [EINTR].

If a *read*( ) is interrupted by a signal after it has successfully read some data, it shall return the number of bytes read.

XSR A *read*( ) from a STREAMS file can read data in three different modes: *byte-stream* mode, *message-nondiscard* mode, and *message-discard* mode. The default is byte-stream mode. This can be changed using the I_SRDOPT *ioctl*( ) request, and can be tested with the I_GRDOPT *ioctl*( ). In byte-stream mode, *read*( ) retrieves data from the STREAM until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, *read*( ) retrieves data until as many bytes as were requested are transferred, or until a message boundary is reached. If *read*( ) does not retrieve all the data in a message, the remaining data is left on the STREAM, and can be retrieved by the next *read*( ) call. Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the *read*( ) returns is discarded, and is not available for a subsequent *read*( ), *readv*( ), or *getmsg*( ) call.

How *read*( ) handles zero-byte STREAMS messages is determined by the current read mode setting. In byte-stream mode, *read*( ) accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The *read*( ) function shall then return the number of bytes read, and place the zero-byte message back on the STREAM to be retrieved by the next *read*( ), *readv*( ), or *getmsg*( ). In message-nondiscard mode or message-discard mode, a zero-byte message shall return 0 and the message shall be removed from the STREAM. When a zero-byte message is read as the first message on a STREAM, the message shall be removed from the STREAM and 0 shall be returned, regardless of the read mode.

A *read*( ) from a STREAMS file shall return the data in the message at the front of the STREAM head read queue, regardless of the priority band of the message.

By default, STREAMs are in control-normal mode, in which a *read*( ) from a STREAMS file can only process messages that contain a data part but do not contain a control part. The *read*( ) shall fail if a message containing a control part is encountered at the STREAM head. This default action can be changed by placing the STREAM in either control-data mode or control-discard mode with the I_SRDOPT *ioctl*( ) command. In control-data mode, *read*( ) converts any control part to data and passes it to the application before passing any data part originally present in the same message. In control-discard mode, *read*( ) shall discard message control parts but return to the process any data part in the message.

In addition, *read*( ) and *readv*( ) fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of *errno* does not reflect the result of *read*( ) or *readv*( ), but reflects the prior error. If a hangup occurs on the STREAM being read, *read*( ) continues to operate normally until the STREAM head read queue is empty. Thereafter, it shall return 0.

XSI The *readv*( ) function is equivalent to *read*( ), but places the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt*–1]. The *iovcnt* argument is valid if greater than 0 and less than or equal to {IOV_MAX}.

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. The *readv*( ) function always fills an area completely before proceeding to the next.

Upon successful completion, *readv*( ) shall mark for update the *st_atime* field of the file.

SIO If the O_DSYNC and O_RSYNC bits have been set, read I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion. If the O_SYNC and O_RSYNC bits have been set, read I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.

SHM If *fildes* refers to a shared memory object, the result of the *read*( ) function is unspecified.

TYM If *fildes* refers to a typed memory object, the result of the *read*( ) function is unspecified.

For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with *fildes*.

XSI The *pread*( ) function performs the same action as *read*( ), except that it reads from a given position in the file without changing the file pointer. The first three arguments to *pread*( ) are the same as *read*( ) with the addition of a fourth argument offset for the desired position inside the file. An attempt to perform a *pread*( ) on a file that is incapable of seeking results in an error.

If *fildes* refers to a socket, *read*( ) is equivalent to *recv*( ) with no flags set.

RETURN VALUE

XSI Upon successful completion, *read*( ), *pread*( ) and *readv*( ) shall return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions shall return –1 and set *errno* to indicate the error.

ERRORS

XSI The *read*( ), *pread*( ) and *readv*( ) functions shall fail if:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the process would be delayed.

[EBADF] The *fildes* argument is not a valid file descriptor open for reading.

XSR [EBADMSG] The file is a STREAM file that is set to control-normal mode and the message waiting to be read includes a control part.

[EINTR] The read operation was terminated due to the receipt of a signal, and no data was transferred.

XSR [EINVAL] The STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

[EIO] The process is a member of a background process attempting to read from its controlling terminal, the process is ignoring or blocking the SIGTTIN signal, or the process group is orphaned. This error may also be generated for implementation-defined reasons.

XSI [EISDIR] The *fildes* argument refers to a directory and the implementation does not allow the directory to be read using *read*( ), *pread*( ), or *readv*( ). The *readdir*( ) function should be used instead.

[EOVERFLOW] The file is a regular file, *nbyte* is greater than 0, the starting position is before the end-of-file, and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fildes*.

MAN The *read*( ) function shall fail if:

[EAGAIN] or [EWOULDBLOCK]
The file descriptor is for a connection-made socket, is marked O_NONBLOCK, and no data is waiting to be received.

[ECONNRESET] A read was attempted on a connection-mode socket and the connection was forcibly closed by its peer.

[ENOTCONN] A read was attempted on a connection-made socket that is not connected.

[ETIMEDOUT] A read was attempted on a connection-mode socket and a transmission timeout occurred.

The *readv*( ) function shall fail if:

[EINVAL] The sum of the *iov_len* values in the *iov* array overflowed an **ssize_t**.

XSI The *read*( ),*pread*( ) and *readv*( )functions may fail if:

MAN [EIO] A physical I/O error has occurred.

MAN [ENOBUFS] Insufficient resources were available in the system to perform the operation.

MAN [ENOMEM] Insufficient memory was available to fulfill the request.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

The *readv*( ) function may fail if:

XSI [EINVAL] The *iovcnt* argument was less than or equal to 0, or greater than {IOV_MAX}.

XSI The *pread*( ) function shall fail, and the file pointer shall remain unchanged, if:

XSI [EINVAL] The *offset* argument is invalid. The value is negative.

XSI [EOVERFLOW] The file is a regular file and an attempt was made to read or write at or beyond the offset maximum associated with the file.

XSI [ENXIO] A request was outside the capabilities of the device.

XSI [ESPIPE] *fildes* 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*.

```
#include <sys/types.h>
#include <unistd.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_read;
int fd;
...
nbytes = sizeof(buf);
bytes_read = read(fd, buf, nbytes);
...
```

Reading Data into an Array

The following example reads data from the file associated with the file descriptor *fd* into the buffers specified by members of the *iov* array.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
...
ssize_t bytes_read;
int fd;
char buf0[20];
char buf1[30];
char buf2[40];
int iovcnt;
struct iovec iov[3];

iov[0].iov_base = buf0;
iov[0].iov_len = sizeof(buf0);
iov[1].iov_base = buf1;
iov[1].iov_len = sizeof(buf1);
iov[2].iov_base = buf2;
iov[2].iov_len = sizeof(buf2);
...
iovcnt = sizeof(iov) / sizeof(struct iovec);

bytes_read = readv(fd, iov, iovcnt);
...
```

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.

Input and Output

The use of I/O with large byte counts has always presented problems. Ideas such as *lread*( ) and *lwrite*( ) (using and returning **long**s) 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.

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.

This volume of IEEE Std. 1003.1-200x requires that no action be taken 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.

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).

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.

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.

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.

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.

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 device and impossible on a non-seekable device. Most new implementations support this behavior. The behavior required by IEEE Std. 1003.1-200x is to return the number of bytes transferred.

IEEE Std. 1003.1-200x does not specify when an implementation that buffers *read*( )s actually moves the data into the user-supplied buffer, so an implementation may chose to do this at the latest possible moment. Therefore, an interrupt arriving earlier may not cause *read*( ) to return a partial byte count, but rather to return –1 and set *errno* to [EINTR].

Consideration was also given to combining the two previous options, and setting *errno* to [EINTR] while returning a short count. However, not only is there no existing practice that

implements this, it is also contradictory to the idea that when *errno* is set, the function responsible shall return –1.

FUTURE DIRECTIONS

None.

SEE ALSO

fcntl( ), *ioctl*( ), *lseek*( ), *open*( ), *pipe*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stropts.h>**, **<sys/uio.h>**, **<unistd.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

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

Issue 4

The **<unistd.h>** header is added to the SYNOPSIS section.

The DESCRIPTION is rearranged for clarity and to align more closely with the ISO POSIX-1 standard. No functional changes are made other than as noted elsewhere in this CHANGE HISTORY section.

In the ERRORS section in previous issues, generation of the [EIO] error depended on whether or not an implementation supported Job Control. This functionality is now defined as mandatory.

The [ENXIO] error is marked as an extension.

The APPLICATION USAGE section is removed.

The description of [EINTR] is amended.

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The type of the argument *buf* is changed from **char*** to **void***, and the type of the argument *nbyte* is changed from **unsigned** to **size_t**.
- The DESCRIPTION now states that the result is implementation-defined if *nbyte* is greater than {SSIZE_MAX}. This limit was defined by the constant {INT_MAX} in Issue 3.

The following change is incorporated for alignment with the FIPS requirements:

- The last paragraph of the DESCRIPTION now states that if *read*( ) is interrupted by a signal after it has successfully read some data, it returns the number of bytes read. In Issue 3, it was optional whether *read*( ) returned the number of bytes read, or whether it returned –1 with *errno* set to [EINTR].

Issue 4, Version 2

The following changes are incorporated for X/OPEN UNIX conformance:

- The *readv*( ) function is added to the SYNOPSIS.
- The DESCRIPTION is updated to describe the reading of data from STREAMS files. An operational description of the *readv*( ) function is also added.
- References to the *readv*( ) function are added to the RETURN VALUE and ERRORS sections in appropriate places.
- The ERRORS section has been restructured to describe errors that apply generally (that is, to both *read*( ) and *readv*( )), and to describe those that apply to *readv*( ) specifically. The [EBADMSG], [EINVAL], and [EISDIR] errors are also added.

Issue 5

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

Large File Summit extensions are added.

The *pread*( ) function is added.

Issue 6

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:

- The DESCRIPTION now states that if *read*( ) is interrupted by a signal after it has successfully read some data, it returns the number of bytes read. In Issue 3, it was optional whether *read*( ) returned the number of bytes read, or whether it returned –1 with *errno* set to [EINTR]. This is a FIPS requirement.
- In the DESCRIPTION, text is added to indicate that for regular files, no data transfer occurs past the offset maximum established in the open file description associated with *fildes*. This change is to support large files.
- The [EOVERFLOW] mandatory error condition is added.
- 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 *read*( ) results are unspecified for typed memory objects.

New RATIONALE is added to explain the atomicity requirements for input and output operations.

The following error conditions are added for operations on sockets: [EAGAIN], [ECONNRESET], [ENOTCONN], and [ETIMEDOUT].

The [EIO] error is changed to ‘‘may fail’’.

The following error conditions are added for operations on sockets: [ENOBUFS] and [ENOMEM].

36662 **NAME**

36663 readdir, readdir_r — read directory

36664 **SYNOPSIS**

36665 ```
#include <dirent.h>
```

36666 ```
struct dirent *readdir(DIR *dirp);
```

36667 TSF ```
int readdir_r(DIR *restrict dirp, struct dirent *restrict entry,
```

36668 ```
    struct dirent **restrict result);
```

36669

36670 **DESCRIPTION**

36671 The type **DIR**, which is defined in the header **<dirent.h>**, represents a *directory stream*, which is
36672 an ordered sequence of all the directory entries in a particular directory. Directory entries
36673 represent files; files may be removed from a directory or added to a directory asynchronously to
36674 the operation of *readdir*( ).

36675 The *readdir*( ) function shall return a pointer to a structure representing the directory entry at the
36676 current position in the directory stream specified by the argument *dirp*, and position the
36677 directory stream at the next entry. It shall return a null pointer upon reaching the end of the
36678 directory stream. The structure **dirent** defined by the **<dirent.h>** header describes a directory
36679 entry.

36680 The *readdir*( ) function shall not return directory entries containing empty names. If entries for
36681 dot or dot-dot exist, one entry shall be returned for dot and one entry shall be returned for dot-
36682 dot; otherwise, they shall not be returned.

36683 The pointer returned by *readdir*( ) points to data which may be overwritten by another call to
36684 *readdir*( ) on the same directory stream. This data is not overwritten by another call to *readdir*( )
36685 on a different directory stream.

36686 If a file is removed from or added to the directory after the most recent call to *opendir*( ) or
36687 *rewinddir*( ), whether a subsequent call to *readdir*( ) returns an entry for that file is unspecified.

36688 The *readdir*( ) function may buffer several directory entries per actual read operation; *readdir*( )
36689 shall mark for update the *st_atime* field of the directory each time the directory is actually read.

36690 After a call to *fork*( ), either the parent or child (but not both) may continue processing the
36691 XSI directory stream using *readdir*( ), *rewinddir*( ), or *seekdir*( ). If both the parent and child processes
36692 use these functions, the result is undefined.

36693 If the entry names a symbolic link, the value of the *d_ino* member is unspecified.

36694 The *readdir*( ) function need not be reentrant. A function that is not required to be reentrant is not
36695 required to be thread-safe.

36696 TSF The *readdir_r*( ) function initializes the **dirent** structure referenced by *entry* to represent the
36697 directory entry at the current position in the directory stream referred to by *dirp*, store a pointer
36698 to this structure at the location referenced by *result*, and position the directory stream at the next
36699 entry.

36700 The storage pointed to by *entry* shall be large enough for a **dirent** with an array of **char** *d_name*
36701 members containing at least {NAME_MAX} plus one elements.

36702 Upon successful return, the pointer returned at **result* shall have the same value as the argument
36703 *entry*. Upon reaching the end of the directory stream, this pointer shall have the value NULL.

36704 The *readdir_r*( ) function shall not return directory entries containing empty names.

If a file is removed from or added to the directory after the most recent call to *opendir*() or *rewinddir*(), whether a subsequent call to *readdir_r*() returns an entry for that file is unspecified.

The *readdir_r*() function may buffer several directory entries per actual read operation; the *readdir_r*() function shall mark for update the *st_atime* field of the directory each time the directory is actually read.

Applications wishing to check for error situations should set *errno* to 0 before calling *readdir*(). If *errno* is set to non-zero on return, an error occurred.

RETURN VALUE

Upon successful completion, *readdir*() shall return a pointer to an object of type **struct dirent**. When an error is encountered, a null pointer shall be returned and *errno* shall be set to indicate the error. When the end of the directory is encountered, a null pointer shall be returned and *errno* is not changed.

TSF If successful, the *readdir_r*() function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *readdir*() function shall fail if:

[EOVERFLOW] One of the values in the structure to be returned cannot be represented correctly.

The *readdir*() function may fail if:

[EBADF] The *dirp* argument does not refer to an open directory stream.

[ENOENT] The current position of the directory stream is invalid.

The *readdir_r*() function may fail if:

[EBADF] The *dirp* argument does not refer to an open directory stream.

EXAMPLES

The following sample code searches the current directory for the entry *name*:

```
dirp = opendir(".");
while (dirp) {
    errno = 0;
    if ((dp = readdir(dirp)) != NULL) {
        if (strcmp(dp->d_name, name) == 0) {
            closedir(dirp);
            return FOUND;
        }
    } else {
        if (errno == 0) {
            closedir(dirp);
            return NOT_FOUND;
        }
        closedir(dirp);
        return READ_ERROR;
    }
}
return OPEN_ERROR;
```

APPLICATION USAGE

The *readdir*( ) function should be used in conjunction with *opendir*( ), *closedir*( ), and *rewinddir*( ) to examine the contents of the directory.

The *readdir_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

The returned value of *readdir*( ) merely *represents* a directory entry. No equivalence should be inferred.

Historical implementations of *readdir*( ) obtain multiple directory entries on a single read operation, which permits subsequent *readdir*( ) operations to operate from the buffered information. Any wording that required each successful *readdir*( ) operation to mark the directory *st_atime* field for update would militate against the historical performance-oriented implementations.

Since *readdir*( ) returns NULL when it detects an error and when the end of the directory is encountered, an application that needs to tell the difference must set *errno* to zero before the call and check it if NULL is returned. Because the function must not change *errno* in the second case and must set it to a non-zero value in the first case, a zero *errno* after a call returning NULL indicates end of directory; otherwise, an error.

Routines to deal with this problem more directly were proposed:

```
int derror (dirp)
DIR *dirp;
```

```
void clearderr (dirp)
DIR *dirp;
```

The first would indicate whether an error had occurred, and the second would clear the error indication. The simpler method involving *errno* was adopted instead by requiring that *readdir*( ) not change *errno* when end-of-directory is encountered.

An error or signal indicating that a directory has changed while open was considered but rejected.

The thread-safe version of the directory reading function shall return values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call. Either the {NAME_MAX} compile-time constant or the corresponding *pathconf*( ) option can be used to determine the maximum sizes of returned path names.

FUTURE DIRECTIONS

None.

SEE ALSO

closedir( ), *lstat*( ), *opendir*( ), *rewinddir*( ), *symlink*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<dirent.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 2.

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

In the DESCRIPTION, the fact that XSI-conformant systems return entries for dot and dot-dot is marked as an extension. This functionality is not specified in the ISO POSIX-1 standard.

There is some rewording of the DESCRIPTION and RETURN VALUE sections. No functional changes are made other than as noted elsewhere in this CHANGE HISTORY section.

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The last paragraph of the DESCRIPTION describing a restriction after *fork*( ) is added.

Issue 4, Version 2

The following changes are incorporated for X/OPEN UNIX conformance:

- A statement is added to the DESCRIPTION indicating the disposition of certain fields in **struct dirent** when an entry refers to a symbolic link.
- The [ENOENT] optional error condition is added.

Issue 5

Large File Summit extensions are added.

The *readdir_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *readdir*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *readdir_r*( ) function is marked as part of the Thread-Safe Functions option.

The Open Group corrigenda item U026/7 has been applied, correcting the prototype for *readdir_r*( ).

The Open Group corrigenda item U026/8 has been applied, clarifying the wording of the successful return for the *readdir_r*( ) function.

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 statement is added to the DESCRIPTION indicating the disposition of certain fields in **struct dirent** when an entry refers to a symbolic link.
- The [EOVERFLOW] mandatory error condition is added. This change is to support large files.
- The [ENOENT] optional error condition is 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.

The **restrict** keyword is added to the *readdir_r*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

36826 **NAME**

36827 readlink — read the contents of a symbolic link

36828 **SYNOPSIS**

36829 `#include <unistd.h>`

36830 `ssize_t readlink(const char *restrict path, char *restrict buf,` |
36831 `size_t bufsize);` |

36832 **DESCRIPTION** |

36833 The *readlink*( ) function shall place the contents of the symbolic link referred to by *path* in the
36834 buffer *buf* which has size *bufsize*. If the number of bytes in the symbolic link is less than *bufsize*,
36835 the contents of the remainder of *buf* are unspecified. If the *buf* argument is not large enough to
36836 contain the link content, the first *bufsize* bytes shall be placed in *buf*.

36837 If the value of *bufsize* is greater than {SSIZE_MAX}, the result is implementation-defined. |

36838 **RETURN VALUE**

36839 Upon successful completion, *readlink*( ) shall return the count of bytes placed in the buffer.
36840 Otherwise, it shall return a value of –1, leave the buffer unchanged, and set *errno* to indicate the
36841 error.

36842 **ERRORS**

36843 The *readlink*( ) function shall fail if:

36844 [EACCES] Search permission is denied for a component of the path prefix of *path*. |

36845 [EINVAL] The *path* argument names a file that is not a symbolic link. |

36846 [EIO] An I/O error occurred while reading from the file system. |

36847 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* |
36848 argument.

36849 [ENAMETOOLONG] |
36850 The length of the *path* argument exceeds {PATH_MAX} or a path name |
36851 component is longer than {NAME_MAX}. |

36852 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string. |

36853 [ENOTDIR] A component of the path prefix is not a directory. |

36854 The *readlink*( ) function may fail if:

36855 [EACCES] Read permission is denied for the directory. |

36856 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
36857 resolution of the *path* argument.

36858 [ENAMETOOLONG] |
36859 As a result of encountering a symbolic link in resolution of the *path* argument,
36860 the length of the substituted path name string exceeded {PATH_MAX}.

EXAMPLES

Reading the Name of a Symbolic Link

The following example shows how to read the name of a symbolic link named **/modules/pass1**.

```
#include <unistd.h>

char buf[1024];
int len;
...
if ((len = readlink("/modules/pass1", buf, sizeof(buf)-1)) != -1);
buf[len] = '\0';
```

APPLICATION USAGE

Portable 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.

FUTURE DIRECTIONS

None.

SEE ALSO

lstat( ), *stat*( ), *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 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 [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.
- 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.

36907 **NAME**

36908 readv — vectored read from file

36909 **SYNOPSIS**

36910 XSI
```
#include <sys/uio.h>
```

36911
```
ssize_t readv(int fildes, const struct iovec *iov, int iovcnt);
```

36912

36913 **DESCRIPTION**

36914 Refer to *read*( ).

36915 **NAME**

36916 realloc — memory reallocator

36917 **SYNOPSIS**

36918 `#include <stdlib.h>`

36919 `void *realloc(void *`*`ptr`*`, size_t `*`size`*`);`

36920 **DESCRIPTION**

36921 CX The functionality described on this reference page is aligned with the ISO C standard. Any
36922 conflict between the requirements described here and the ISO C standard is unintentional. This
36923 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

36924 The *realloc*( ) function shall change the size of the memory object pointed to by *ptr* to the size
36925 specified by *size*. The contents of the object shall remain unchanged up to the lesser of the new
36926 and old sizes. If the new size of the memory object would require movement of the object, the
36927 space for the previous instantiation of the object is freed. If the new size is larger, the contents of
36928 the newly allocated portion of the object are unspecified. If *size* is 0 and *ptr* is not a null pointer,
36929 the object pointed to is freed. If the space cannot be allocated, the object remains unchanged.

36930 If *ptr* is a null pointer, *realloc*( ) shall behave like *malloc*( ) for the specified size.

36931 If *ptr* does not match a pointer returned earlier by *calloc*( ), *malloc*( ), or *realloc*( ) or if the space has
36932 previously been deallocated by a call to *free*( ) or *realloc*( ), the behavior is undefined.

36933 The order and contiguity of storage allocated by successive calls to *realloc*( ) is unspecified. The |
36934 pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a
36935 pointer to any type of object and then used to access such an object in the space allocated (until
36936 the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object
36937 disjoint from any other object. The pointer returned points to the start (lowest byte address) of |
36938 the allocated space. If the space cannot be allocated, a null pointer shall be returned. |

36939 **RETURN VALUE**

36940 Upon successful completion with a size not equal to 0, *realloc*( ) shall return a pointer to the
36941 (possibly moved) allocated space. If *size* is 0, either a null pointer or a unique pointer that can be
36942 successfully passed to *free*( ) shall be returned. If there is not enough available memory, *realloc*( )
36943 CX shall return a null pointer and set *errno* to [ENOMEM].

36944 **ERRORS**

36945 The *realloc*( ) function shall fail if:

36946 CX [ENOMEM] Insufficient memory is available. |

36947 **EXAMPLES**

36948 None.

36949 **APPLICATION USAGE**

36950 None.

36951 **RATIONALE**

36952 None.

36953 **FUTURE DIRECTIONS**

36954 None.

36955 **SEE ALSO**

36956 *calloc*( ), *free*( ), *malloc*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>** |

36957 **CHANGE HISTORY**

36958 First released in Issue 1. Derived from Issue 1 of the SVID.

36959 **Issue 4**

36960 The setting of *errno* and the [ENOMEM] error are marked as extensions.

36961 The APPLICATION USAGE section is removed.

36962 The following changes are incorporated for alignment with the ISO C standard:

36963 • The DESCRIPTION is updated to indicate as follows:

36964 — The order and contiguity of storage allocated by successive calls to this function is
36965 unspecified.

36966 — Each allocation yields a pointer to an object disjoint from any other object.

36967 — The returned pointer points to the lowest byte address of the allocation.

36968 • The RETURN VALUE section is updated to indicate what is returned if *size* is 0.

36969 **Issue 6**

36970 Extensions beyond the ISO C standard are now marked.

36971 The following new requirements on POSIX implementations derive from alignment with the
36972 Single UNIX Specification:

36973 • In the RETURN VALUE section, if there is not enough available memory, the setting of *errno*
36974 to [ENOMEM] is added.

36975 • The [ENOMEM] error condition is added.

NAME

realpath — resolve a path name

SYNOPSIS

XSI

```
#include <stdlib.h>

char *realpath(const char *restrict file_name,
    char *restrict resolved_name);
```

DESCRIPTION

The *realpath*( ) function derives, from the path name pointed to by *file_name*, an absolute path name that names the same file, whose resolution does not involve `'.'`, `'..'`, or symbolic links. The generated path name is stored as a null-terminated string, up to a maximum of {PATH_MAX} bytes, in the buffer pointed to by *resolved_name*.

RETURN VALUE

Upon successful completion, *realpath*( ) shall return a pointer to the resolved name. Otherwise, *realpath*( ) shall return a null pointer and set *errno* to indicate the error, and the contents of the buffer pointed to by *resolved_name* are undefined.

ERRORS

The *realpath*( ) function shall fail if:

[EACCES] Read or search permission was denied for a component of *file_name*.

[EINVAL] Either the *file_name* or *resolved_name* argument is a null pointer.

[EIO] An error occurred while reading from the file system.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

[ENAMETOOLONG]
The length of the *file_name* argument exceeds {PATH_MAX} or a path name component is longer than {NAME_MAX}.

[ENOENT] A component of *file_name* does not name an existing file or *file_name* points to an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

The *realpath*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
Path name resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

[ENOMEM] Insufficient storage space is available.

EXAMPLES

Generating an Absolute Path Name

The following example generates an absolute path name for the file identified by the *symlinkpath* argument. The generated path name is stored in the *actualpath* array.

```
#include <stdlib.h>
...
char *symlinkpath = "/tmp/symlink/file";
char actualpath [PATH_MAX+1];
char *ptr;

ptr = realpath(symlinkpath, actualpath);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getcwd( ), *sysconf*( ), 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 6

The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

The **restrict** keyword is added to the *realpath*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

NAME

recv — receive a message from a connected socket

SYNOPSIS

```
#include <sys/socket.h>

ssize_t recv(int socket, void *buffer, size_t length, int flags);
```

DESCRIPTION

The *recv*( ) function receives a message from a connection-mode or connectionless-mode socket. It is normally used with connected sockets because it does not permit the application to retrieve the source address of received data.

The *recv*( ) function takes the following arguments:

socket Specifies the socket file descriptor.

buffer Points to a buffer where the message should be stored.

length Specifies the length in bytes of the buffer pointed to by the *buffer* argument.

flags Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:

MSG_PEEK Peeks at an incoming message. The data is treated as unread and the next *recv*( ) or similar function shall still return this data.

MSG_OOB Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.

MSG_WAITALL Requests that the function block until the full amount of data requested can be returned. The function may return a smaller amount of data 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 *recv*( ) function shall return the length of the message written to the buffer pointed to by the *buffer* argument. 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 buffer, and MSG_PEEK is not set in the *flags* argument, the excess bytes shall be discarded. For stream-based sockets, such as SOCK_STREAM, message boundaries shall be ignored. In this case, data is returned to the user as soon as it becomes available, and no data shall be discarded.

If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first message.

If no messages are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, *recv*( ) shall block until a message arrives. If no messages are available at the socket and O_NONBLOCK is set on the socket's file descriptor, *recv*( ) shall fail and set *errno* to [EAGAIN] or [EWOULDBLOCK].

RETURN VALUE

Upon successful completion, *recv*( ) shall return the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, *recv*( ) shall return 0. Otherwise, –1 shall be returned and *errno* set to indicate the error.

37082 **ERRORS**

37083 The *recv*( ) function shall fail if:

37084 [EAGAIN] or [EWOULDBLOCK]

37085 The socket's file descriptor is marked O_NONBLOCK and no data is waiting
37086 to be received; or MSG_OOB is set and no out-of-band data is available and
37087 either the socket's file descriptor is marked O_NONBLOCK or the socket does
37088 not support blocking to await out-of-band data.

37089 [EBADF] The *socket* argument is not a valid file descriptor.

37090 [ECONNRESET] A connection was forcibly closed by a peer.

37091 [EINTR] The *recv*( ) function was interrupted by a signal that was caught, before any
37092 data was available.

37093 [EINVAL] The MSG_OOB flag is set and no out-of-band data is available.

37094 [ENOTCONN] A receive is attempted on a connection-mode socket that is not connected.

37095 [ENOTSOCK] The *socket* argument does not refer to a socket.

37096 [EOPNOTSUPP] The specified flags are not supported for this socket type or protocol.

37097 [ETIMEDOUT] The connection timed out during connection establishment, or due to a
37098 transmission timeout on active connection.

37099 The *recv*( ) function may fail if:

37100 [EIO] An I/O error occurred while reading from or writing to the file system.

37101 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

37102 [ENOMEM] Insufficient memory was available to fulfill the request.

37103 **EXAMPLES**

37104 None.

37105 **APPLICATION USAGE**

37106 The *recv*( ) function is identical to *recvfrom*( ) with a zero *address_len* argument, and to *read*( ) if no
37107 flags are used.

37108 The *select*( ) and *poll*( ) functions can be used to determine when data is available to be received.

37109 **RATIONALE**

37110 None.

37111 **FUTURE DIRECTIONS**

37112 None.

37113 **SEE ALSO**

37114 *poll*( ), *read*( ), *recvmsg*( ), *recvfrom*( ), *select*( ), *send*( ), *sendmsg*( ), *sendto*( ), *shutdown*( ), *socket*( ),
37115 *write*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/socket.h>**

37116 **CHANGE HISTORY**

37117 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

37118 **NAME**

37119 recvfrom — receive a message from a socket

37120 **SYNOPSIS**

37121 ```
#include <sys/socket.h>
```

37122 ```
ssize_t recvfrom(int socket, void *restrict buffer, size_t length,
```
37123 ```
    int flags, struct sockaddr *restrict address,
```
37124 ```
    socklen_t *restrict address_len);
```

37125 **DESCRIPTION**

37126 The *recvfrom*( ) function receives a message from a connection-mode or connectionless-mode
37127 socket. It is normally used with connectionless-mode sockets because it permits the application
37128 to retrieve the source address of received data.

37129 The *recvfrom*( ) function takes the following arguments:

37130 *socket* Specifies the socket file descriptor.

37131 *buffer* Points to the buffer where the message should be stored.

37132 *length* Specifies the length in bytes of the buffer pointed to by the *buffer* argument.

37133 *flags* Specifies the type of message reception. Values of this argument are formed
37134 by logically OR'ing zero or more of the following values:

37135 MSG_PEEK Peeks at an incoming message. The data is treated as unread
37136 and the next *recvfrom*( ) or similar function shall still return
37137 this data.

37138 MSG_OOB Requests out-of-band data. The significance and semantics
37139 of out-of-band data are protocol-specific.

37140 MSG_WAITALL Requests that the function block until the full amount of
37141 data requested can be returned. The function may return a
37142 smaller amount of data if a signal is caught, if the
37143 connection is terminated, if MSG_PEEK was specified, or if
37144 an error is pending for the socket.

37145 *address* A null pointer, or points to a **sockaddr** structure in which the sending address
37146 is to be stored. The length and format of the address depend on the address
37147 family of the socket.

37148 *address_len* Specifies the length of the **sockaddr** structure pointed to by the *address*
37149 argument.

37150 The *recvfrom*( ) function shall return the length of the message written to the buffer pointed to by
37151 the *buffer* argument. For message-based sockets, such as SOCK_DGRAM and
37152 SOCK_SEQPACKET, the entire message shall be read in a single operation. If a message is too
37153 long to fit in the supplied buffer, and MSG_PEEK is not set in the *flags* argument, the excess
37154 bytes shall be discarded. For stream-based sockets, such as SOCK_STREAM, message
37155 boundaries shall be ignored. In this case, data is returned to the user as soon as it becomes
37156 available, and no data shall be discarded.

37157 If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first
37158 message.

37159 Not all protocols provide the source address for messages. If the *address* argument is not a null
37160 pointer and the protocol provides the source address of messages, the source address of the
37161 received message is stored in the **sockaddr** structure pointed to by the *address* argument, and the

length of this address is stored in the object pointed to by the *address_len* argument.

If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the stored address shall be truncated.

If the *address* argument is not a null pointer and the protocol does not provide the source address of messages, the value stored in the object pointed to by *address* is unspecified.

If no messages are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, *recvfrom*( ) blocks until a message arrives. If no messages are available at the socket and O_NONBLOCK is set on the socket's file descriptor, *recvfrom*( ) shall fail and set *errno* to [EAGAIN] or [EWOULDBLOCK].

RETURN VALUE

Upon successful completion, *recvfrom*( ) shall return the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, *recvfrom*( ) shall return 0. Otherwise, the function shall return –1 and set *errno* to indicate the error.

ERRORS

The *recvfrom*( ) 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]
: A signal interrupted *recvfrom*( ) 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.

[ETIMEDOUT]
: The connection timed out during connection establishment, or due to a transmission timeout on active connection.

The *recvfrom*( ) 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 *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*( ), *recv*( ), *recvmsg*( ), *select*( ) (on page 1753)1 *send*( ), *sendmsg*( ), *sendto*( ), *shutdown*( ), *socket*( ), *write*( ), 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.

37209 **NAME**

37210 recvmsg — receive a message from a socket

37211 **SYNOPSIS**

37212 `#include <sys/socket.h>`

37213 `ssize_t recvmsg(int socket, struct msghdr *message, int flags);`

37214 **DESCRIPTION**

37215 The *recvmsg*( ) function receives a message from a connection-mode or connectionless-mode
37216 socket. It is normally used with connectionless-mode sockets because it permits the application
37217 to retrieve the source address of received data.

37218 The *recvmsg*( ) function takes the following arguments:

37219 *socket* Specifies the socket file descriptor.

37220 *message* Points to a **msghdr** structure, containing both the buffer to store the source
37221 address and the buffers for the incoming message. The length and format of
37222 the address depend on the address family of the socket. The *msg_flags* member
37223 is ignored on input, but may contain meaningful values on output.

37224 *flags* Specifies the type of message reception. Values of this argument are formed
37225 by logically OR'ing zero or more of the following values:

37226 MSG_OOB Requests out-of-band data. The significance and semantics
37227 of out-of-band data are protocol-specific.

37228 MSG_PEEK Peeks at the incoming message.

37229 MSG_WAITALL Requests that the function block until the full amount of
37230 data requested can be returned. The function may return a
37231 smaller amount of data if a signal is caught, if the
37232 connection is terminated, if MSG_PEEK was specified, or if
37233 an error is pending for the socket.

37234 The *recvmsg*( ) function receives messages from unconnected or connected sockets and shall
37235 return the length of the message.

37236 The *recvmsg*( ) function shall return the total length of the message. For message-based sockets,
37237 such as SOCK_DGRAM and SOCK_SEQPACKET, the entire message shall be read in a single
37238 operation. If a message is too long to fit in the supplied buffers, and MSG_PEEK is not set in the
37239 *flags* argument, the excess bytes shall be discarded, and MSG_TRUNC is set in the *msg_flags*
37240 member of the **msghdr** structure. For stream-based sockets, such as SOCK_STREAM, message
37241 boundaries shall be ignored. In this case, data is returned to the user as soon as it becomes
37242 available, and no data shall be discarded.

37243 If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first
37244 message.

37245 If no messages are available at the socket and O_NONBLOCK is not set on the socket's file
37246 descriptor, *recvmsg*( ) shall block until a message arrives. If no messages are available at the
37247 socket and O_NONBLOCK is set on the socket's file descriptor, *recvmsg*( ) function shall fail and
37248 set *errno* to [EAGAIN] or [EWOULDBLOCK].

37249 In the **msghdr** structure, the *msg_name* and *msg_namelen* members specify the source address if
37250 the socket is unconnected. If the socket is connected, the *msg_name* and *msg_namelen* members
37251 are ignored. The *msg_name* member may be a null pointer if no names are desired or required.
37252 The *msg_iov* and *msg_iovlen* fields are used to specify where the received data shall be stored.
37253 *msg_iov* points to an array of **iovec** structures; *msg_iovlen* shall be set to the dimension of this

array. In each **iovec** structure, the *iov_base* field specifies a storage area and the *iov_len* field gives its size in bytes. Each storage area indicated by *msg_iov* is filled with received data in turn until all of the received data is stored or all of the areas have been filled.

Upon successful completion, the *msg_flags* member of the message header is the bitwise-inclusive OR of all of the following flags that indicate conditions detected for the received message:

MSG_EOR End of record was received (if supported by the protocol).

MSG_OOB Out-of-band data was received.

MSG_TRUNC Normal data was truncated.

MSG_CTRUNC Control data was truncated.

RETURN VALUE

Upon successful completion, *recvmsg*( ) shall return the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, *recvmsg*( ) shall return 0. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *recvmsg*( ) 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 open file descriptor.

[ECONNRESET] A connection was forcibly closed by a peer.

[EINTR] This function was interrupted by a signal before any data was available.

[EINVAL] The sum of the *iov_len* values overflows a **ssize_t**, or the MSG_OOB flag is set and no out-of-band data is available.

[EMSGSIZE] The *msg_iovlen* member of the **msghdr** structure pointed to by *message* is less than or equal to 0, or is greater than {IOV_MAX}.

[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.

[ETIMEDOUT] The connection timed out during connection establishment, or due to a transmission timeout on active connection.

The *recvmsg*( ) 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.

37291 **EXAMPLES**

37292 None.

37293 **APPLICATION USAGE**

37294 The *select*( ) and *poll*( ) functions can be used to determine when data is available to be received.

37295 **RATIONALE**

37296 None.

37297 **FUTURE DIRECTIONS**

37298 None.

37299 **SEE ALSO**

37300 *poll*( ), *recv*( ), *recvfrom*( ), *select*( ), *send*( ), *sendmsg*( ), *sendto*( ), *shutdown*( ), *socket*( ), the Base |
37301 Definitions volume of IEEE Std. 1003.1-200x, **<sys/socket.h>** |

37302 **CHANGE HISTORY**

37303 First released in Issue 6. Derived from the XNS, Issue 5.2 specification. |

37304 **NAME**

37305 regcomp, regerror, regexec, regfree — regular expression matching

37306 **SYNOPSIS**

37307
```
#include <regex.h>

int regcomp(regex_t *restrict preg, const char *restrict pattern, int cflags);|
size_t regerror(int errcode, const regex_t *restrict preg,                    |
    char *restrict errbuf, size_t errbuf_size);                               |
int regexec(const regex_t *restrict preg, const char *restrict string,        |
    size_t nmatch, regmatch_t pmatch[restrict], int eflags);                  |
void regfree(regex_t *preg);                                                  |
```

37314 **DESCRIPTION**

37315 These functions interpret *basic* and *extended* regular expressions as described in the Base |
37316 Definitions volume of IEEE Std. 1003.1-200x, Chapter 9, Regular Expressions. |

37317 The **regex_t** structure contains at least the following member:

37318

37319

Member Type	Member Name	Description
`size_t`	`re_nsub`	Number of parenthesized subexpressions.

37320

37321 The **regmatch_t** structure contains at least the following members:

37322

37323

Member Type	Member Name	Description
regoff_t	*rm_so*	Byte offset from start of *string* to start of substring.
regoff_t	*rm_eo*	Byte offset from start of *string* of the first character after the end of substring.

37324

37325

37326

37327 The *regcomp*( ) function shall compile the regular expression contained in the string pointed to by
37328 the *pattern* argument and places the results in the structure pointed to by *preg*. The *cflags*
37329 argument is the bitwise-inclusive OR of zero or more of the following flags, which are defined in
37330 the header **<regex.h>**:

37331 REG_EXTENDED Use Extended Regular Expressions.

37332 REG_ICASE Ignore case in match. (See the Base Definitions volume of |
37333 IEEE Std. 1003.1-200x, Chapter 9, Regular Expressions.) |

37334 REG_NOSUB Report only success / fail in *regexec*( ).

37335 REG_NEWLINE Change the handling of <newline> characters, as described in the text.

37336 The default regular expression type for *pattern* is a Basic Regular Expression. The application can
37337 specify Extended Regular Expressions using the REG_EXTENDED *cflags* flag.

37338 If the REG_NOSUB flag was not set in *cflags*, then *regcomp*( ) shall set *re_nsub* to the number of
37339 parenthesized subexpressions (delimited by `"\(\)"` in basic regular expressions or `"( )"` in
37340 extended regular expressions) found in *pattern*.

37341 The *regexec*( ) function compares the null-terminated string specified by *string* with the compiled
37342 regular expression *preg* initialized by a previous call to *regcomp*( ). If it finds a match, *regexec*( )
37343 shall return 0; otherwise, it shall return non-zero indicating either no match or an error. The
37344 *eflags* argument is the bitwise-inclusive OR of zero or more of the following flags, which are
37345 defined in the header **<regex.h>**:

REG_NOTBOL The first character of the string pointed to by *string* is not the beginning of the line. Therefore, the circumflex character (`'^'`), when taken as a special character, shall not match the beginning of *string*.

REG_NOTEOL The last character of the string pointed to by *string* is not the end of the line. Therefore, the dollar sign (`'$'`), when taken as a special character, shall not match the end of *string*.

If *nmatch* is 0 or REG_NOSUB was set in the *cflags* argument to *regcomp*( ), then *regexec*( ) shall ignore the *pmatch* argument. Otherwise, the application shall ensure that the *pmatch* argument points to an array with at least *nmatch* elements, and *regexec*( ) shall fill in the elements of that array with offsets of the substrings of *string* that correspond to the parenthesized subexpressions of *pattern*: *pmatch*[*i*].*rm_so* shall be the byte offset of the beginning and *pmatch*[*i*].*rm_eo* shall be one greater than the byte offset of the end of substring *i*. (Subexpression *i* begins at the *i*th matched open parenthesis, counting from 1.) Offsets in *pmatch*[0] identify the substring that corresponds to the entire regular expression. Unused elements of *pmatch* up to *pmatch*[*nmatch*−1] shall be filled with −1. If there are more than *nmatch* subexpressions in *pattern* (*pattern* itself counts as a subexpression), then *regexec*( ) shall still do the match, but shall record only the first *nmatch* substrings.

When matching a basic or extended regular expression, any given parenthesized subexpression of *pattern* might participate in the match of several different substrings of *string*, or it might not match any substring even though the pattern as a whole did match. The following rules are used to determine which substrings to report in *pmatch* when matching regular expressions:

1. If subexpression *i* in a regular expression is not contained within another subexpression, and it participated in the match several times, then the byte offsets in *pmatch*[*i*] shall delimit the last such match.

2. If subexpression *i* is not contained within another subexpression, and it did not participate in an otherwise successful match, the byte offsets in *pmatch*[*i*] shall be −1. A subexpression does not participate in the match when:

   `'*'` or `"\{\}"` appears immediately after the subexpression in a basic regular expression, or `'*'`, `'?'`, or `"{}"` appears immediately after the subexpression in an extended regular expression, and the subexpression did not match (matched 0 times)

   or:

   `'|'` is used in an extended regular expression to select this subexpression or another, and the other subexpression matched.

3. If subexpression *i* is contained within another subexpression *j*, and *i* is not contained within any other subexpression that is contained within *j*, and a match of subexpression *j* is reported in *pmatch*[*j*], then the match or non-match of subexpression *i* reported in *pmatch*[*i*] shall be as described in 1. and 2. above, but within the substring reported in *pmatch*[*j*] rather than the whole string. The offsets in *pmatch*[*i*] are still relative to the start of string.

Notes to Reviewers

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

D1, XSH, ERN 283 proposes changing ‘‘but within’’ above to ‘‘but describing the substring found within the substring reported in *pmatch*[*j*] rather than the whole string. The byte offsets are relative to the whole string.’’

4. If subexpression *i* is contained in subexpression *j*, and the byte offsets in *pmatch*[*j*] are –1, then the pointers in *pmatch*[*i*] shall also be –1.

5. If subexpression *i* matched a zero-length string, then both byte offsets in *pmatch*[*i*] shall be the byte offset of the character or null terminator immediately following the zero-length string.

If, when *regexec*( ) is called, the locale is different from when the regular expression was compiled, the result is undefined.

If REG_NEWLINE is not set in *cflags*, then a <newline> character in *pattern* or *string* shall be treated as an ordinary character. If REG_NEWLINE is set, then <newline> shall be treated as an ordinary character except as follows:

1. A <newline> character in *string* shall not be matched by a period outside a bracket expression or by any form of a non-matching list (see the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 9, Regular Expressions).

2. A circumflex (`'^'`) in *pattern*, when used to specify expression anchoring (see the Base Definitions volume of IEEE Std. 1003.1-200x, Section 9.3.8, BRE Expression Anchoring), shall match the zero-length string immediately after a <newline> in *string*, regardless of the setting of REG_NOTBOL.

3. A dollar sign (`'$'`) in *pattern*, when used to specify expression anchoring, shall match the zero-length string immediately before a <newline> in *string*, regardless of the setting of REG_NOTEOL.

The *regfree*( ) function frees any memory allocated by *regcomp*( ) associated with *preg*.

The following constants are defined as error return values:

REG_NOMATCH	*regexec*( ) failed to match.
REG_BADPAT	Invalid regular expression.
REG_ECOLLATE	Invalid collating element referenced.
REG_ECTYPE	Invalid character class type referenced.
REG_EESCAPE	Trailing `'\'` in pattern.
REG_ESUBREG	Number in `"\digit"` invalid or in error.
REG_EBRACK	`"[]"` imbalance.
REG_EPAREN	`"\(\)"` or `"()"` imbalance.
REG_EBRACE	`"\{\}"` imbalance.
REG_BADBR	Content of `"\{\}"` invalid: not a number, number too large, more than two numbers, first larger than second.
REG_ERANGE	Invalid endpoint in range expression.
REG_ESPACE	Out of memory.

37425 REG_BADRPT '?', '*', or '+' not preceded by valid regular expression.

37426 The *regerror*( ) function provides a mapping from error codes returned by *regcomp*( ) and
37427 *regexec*( ) to unspecified printable strings. It generates a string corresponding to the value of the
37428 *errcode* argument, which the application shall ensure is the last non-zero value returned by
37429 *regcomp*( ) or *regexec*( ) with the given value of *preg*. If *errcode* is not such a value, the content of
37430 the generated string is unspecified.

37431 If *preg* is a null pointer, but *errcode* is a value returned by a previous call to *regexec*( ) or *regcomp*( ),
37432 the *regerror*( ) still generates an error string corresponding to the value of *errcode*, but it might not
37433 be as detailed under some implementations.

37434 If the *errbuf_size* argument is not 0, *regerror*( ) shall place the generated string into the buffer of
37435 size *errbuf_size* bytes pointed to by *errbuf*. If the string (including the terminating null) cannot fit
37436 in the buffer, *regerror*( ) shall truncate the string and null-terminates the result.

37437 If *errbuf_size* is 0, *regerror*( ) shall ignore the *errbuf* argument, and return the size of the buffer
37438 needed to hold the generated string.

37439 If the *preg* argument to *regexec*( ) or *regfree*( ) is not a compiled regular expression returned by
37440 *regcomp*( ), the result is undefined. A *preg* is no longer treated as a compiled regular expression
37441 after it is given to *regfree*( ).

37442 **RETURN VALUE**

37443 Upon successful completion, the *regcomp*( ) function shall return 0. Otherwise, it shall return an
37444 integer value indicating an error as described in **<regex.h>**, and the content of *preg* is undefined.
37445 If a code is returned, the interpretation shall be as given in **<regex.h>**.

37446 If *regcomp*( ) detects an invalid RE, it may return REG_BADPAT, or it may return one of the error
37447 codes that more precisely describes the error.

37448 Upon successful completion, the *regexec*( ) function shall return 0. Otherwise, it shall return
37449 REG_NOMATCH to indicate no match. |

37450 Upon successful completion, the *regerror*( ) function shall return the number of bytes needed to
37451 hold the entire generated string, including the null termination. If the return value is greater than
37452 *errbuf_size*, the string returned in the buffer pointed to by *errbuf* has been truncated. |

37453 The *regfree*( ) function shall return no value.

37454 **ERRORS**

37455 No errors are defined.

37456 **EXAMPLES**

37457
```
#include <regex.h>
```

37458
```
/*
```
37459
```
 * Match string against the extended regular expression in
```
37460
```
 * pattern, treating errors as no match.
```
37461
```
 *
```
37462
```
 * Return 1 for match, 0 for no match.
```
37463
```
 */
```

37464
```
int
```
37465
```
match(const char *string, char *pattern)
```
37466
```
{
```
37467
```
    int    status;
```
37468
```
    regex_t    re;
```

```
        if (regcomp(&re, pattern, REG_EXTENDED|REG_NOSUB) != 0) {
            return(0);      /* Report error. */
        }
        status = regexec(&re, string, (size_t) 0, NULL, 0);
        regfree(&re);
        if (status != 0) {
            return(0);      /* Report error. */
        }
        return(1);
    }
```

The following demonstrates how the REG_NOTBOL flag could be used with *regexec*( ) to find all substrings in a line that match a pattern supplied by a user. (For simplicity of the example, very little error checking is done.)

```
    (void) regcomp (&re, pattern, 0);
    /* This call to regexec() finds the first match on the line. */
    error = regexec (&re, &buffer[0], 1, &pm, 0);
    while (error == 0) {  /* While matches found. */
        /* Substring found between pm.rm_so and pm.rm_eo. */
        /* This call to regexec() finds the next match. */
        error = regexec (&re, buffer + pm.rm_eo, 1, &pm, REG_NOTBOL);
    }
```

APPLICATION USAGE

An application could use:

```
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 2.14, Pattern Matching Notation, use the *fnmatch*( ) function.

RATIONALE

The *regmatch*( ) 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. Since this is a new interface, there should be no impact on historical implementations or applications, and offsets should be just as easy to use as pointers. The change to offsets was made to facilitate future extensions in which the string to be searched is presented to *regexec*( ) in blocks, allowing a string to be searched that is not all in memory at once.

A new type **regoff_t** is used for the elements of *pmatch*[ ] to ensure that the application can represent either the largest possible array in memory (important for an application conforming to the Shell and Utilities volume of IEEE Std. 1003.1-200x) or the largest possible file (important for an application using the extension where a file is searched in chunks).

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 multibyte characters. That is a much larger issue, and one that needs a more general solution.

The *errno* variable has not been used for error returns to avoid filling the *errno* name space for this feature.

The interface is defined so that the matched substrings *rm_sp* and *rm_ep* are in a separate **regmatch_t** structure instead of in **regex_t**. This allows a single compiled RE to be used simultaneously in several contexts; in *main*( ) and a signal handler, perhaps, or in multiple threads of lightweight processes. (The *preg* argument to *regexec*( ) is declared with type **const**, so the implementation is not permitted to use the structure to store intermediate results.) It also allows an application to request an arbitrary number of substrings from an RE. The number of subexpressions in the RE is reported in *re_nsub* in *preg*. With this change to *regexec*( ), consideration was given to dropping the REG_NOSUB flag since the user can now specify this with a zero *nmatch* argument to *regexec*( ). However, keeping REG_NOSUB allows an implementation to use a different (perhaps more efficient) algorithm if it knows in *regcomp*( ) that no subexpressions need be reported. The implementation is only required to fill in *pmatch* if *nmatch* is not zero and if REG_NOSUB is not specified. Note that the **size_t** type, as defined in the ISO C standard, is unsigned, so the description of *regexec*( ) does not need to address negative values of *nmatch*.

REG_NOTBOL was added to allow an application to do repeated searches for the same pattern in a line. If the pattern contains a circumflex character that should match the beginning of a line, then the pattern should only match when matched against the beginning of the line. Without the REG_NOTBOL flag, the application could rewrite the expression for subsequent matches, but in the general case this would require parsing the expression. The need for REG_NOTEOL is not as clear; it was added for symmetry.

The addition of the *regerror*( ) function addresses the historical need for portable application programs to have access to error information more than ‘‘Function failed to compile/match your RE for unknown reasons’’.

This interface provides for two different methods of dealing with error conditions. The specific error codes (REG_EBRACE, for example), defined in **<regex.h>**, allow an application to recover

from an error if it is so able. Many applications, especially those that use patterns supplied by a user, will not try to deal with specific error cases, but will just use *regerror*( ) to obtain a human-readable error message to present to the user.

The *regerror*( ) function uses a scheme similar to *confstr*( ) to deal with the problem of allocating memory to hold the generated string. The scheme used by *strerror*( ) in the ISO C standard was considered unacceptable since it creates difficulties for multi-threaded applications.

The *preg* argument is provided to *regerror*( ) to allow an implementation to generate a more descriptive message than would be possible with *errcode* alone. An implementation might, for example, save the character offset of the offending character of the pattern in a field of *preg*, and then include that in the generated message string. The implementation may also ignore *preg*.

A REG_FILENAME flag was considered, but omitted. This flag caused *regexec*( ) to match patterns as described in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.14, Pattern Matching Notation instead of REs. This service is now provided by the *fnmatch*( ) function.

Notice that there is a difference in philosophy between the ISO POSIX-2: 1993 standard and IEEE Std. 1003.1-200x in how to handle a bad regular expression. The ISO POSIX-2: 1993 standard says that many bad constructs produce undefined results, or that the interpretation is undefined. IEEE Std. 1003.1-200x, however, says that the interpretation of such REs is unspecified. The term ''undefined'' means that the action by the application is an error, of similar severity to passing a bad pointer to a function.

The *regcomp*( ) and *regexec*( ) functions are required to accept any null-terminated string as the *pattern* argument. If the meaning of the string is undefined, the behavior of the function is unspecified. IEEE Std. 1003.1-200x does not specify how the functions will interpret the pattern; they might return error codes, or they might do pattern matching in some completely unexpected way, but they should not do something like abort the process.

FUTURE DIRECTIONS

None.

SEE ALSO

fnmatch( ), *glob*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<regex.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 5

Moved from POSIX2 C-language Binding to BASE.

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.

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

The REG_ENOSYS constant is removed.

The **restrict** keyword is added to the *regcomp*( ), *regerror*( ), and *regexec*( ) prototypes for alignment with the ISO / IEC 9899: 1999 standard.

NAME

remainder, remainderf, remainderl — remainder function

SYNOPSIS

XSI
```
#include <math.h>

double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);
```

DESCRIPTION

These functions shall return the floating point remainder $r=x-ny$ when y is non-zero. The value n is the integral value nearest the exact value x/y. When $|n-x/y|=\frac{1}{2}$, the value n is chosen to be even.

The behavior of *remainder*( ) is independent of the rounding mode.

RETURN VALUE

These functions shall return the floating point remainder $r=x-ny$ when y is non-zero.

When y is 0, *remainder*( ) shall return NaN (or equivalent if available) and set *errno* to [EDOM].

If the value of x is ±Inf, *remainder*( ) shall return NaN and set *errno* to [EDOM].

If x or y is NaN, then the function shall return NaN and *errno* may be set to [EDOM].

ERRORS

These functions shall fail if:

[EDOM] The y argument is 0 or the x argument is positive or negative infinity.

These functions may fail if:

[EDOM] The x or y argument is NaN.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

abs( ), *div*( ), *ldiv*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The *remainderf*( ) and *remainderl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

37649 **NAME**

37650 remove — remove a file

37651 **SYNOPSIS**

37652 `#include <stdio.h>`

37653 `int remove(const char *path);`

37654 **DESCRIPTION**

37655 CX The functionality described on this reference page is aligned with the ISO C standard. Any
37656 conflict between the requirements described here and the ISO C standard is unintentional. This
37657 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

37658 The *remove*( ) function shall cause the file named by the path name pointed to by path to be no
37659 longer accessible by that name. A subsequent attempt to open that file using that name shall fail,
37660 unless it is created anew.

37661 CX If *path* does not name a directory, *remove*(*path*) shall be equivalent to *unlink*(*path*).

37662 If *path* names a directory, *remove*(*path*) shall be equivalent to *rmdir*(*path*).

37663 **RETURN VALUE**

37664 CX Refer to *rmdir*( ) or *unlink*( ).

37665 **ERRORS**

37666 CX Refer to *rmdir*( ) or *unlink*( ).

37667 **EXAMPLES**

37668 **Removing Access to a File**

37669 The following example shows how to remove access to a file named **/home/cnd/old_mods**.

37670 `#include <stdio.h>`

37671 `int status;`
37672 `...`
37673 `status = remove("/home/cnd/old_mods");`

37674 **APPLICATION USAGE**

37675 None.

37676 **RATIONALE**

37677 None.

37678 **FUTURE DIRECTIONS**

37679 None.

37680 **SEE ALSO**

37681 *rmdir*( ), *unlink*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**

37682 **CHANGE HISTORY**

37683 First released in Issue 3.

37684 Entry included for alignment with the POSIX.1-1988 standard and the ISO C standard.

37685 **Issue 4**

37686 All statements containing references to *unlink*( ) and *rmdir*( ) in the DESCRIPTION, RETURN
37687 VALUE, and ERRORS sections are marked as extensions.

37688 The following changes are incorporated for alignment with the ISO C standard:

- The type of argument *path* is changed from **char*** to **const char***.
- The DESCRIPTION is expanded to describe the operation of *remove*( ) more completely.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- 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*( ).

37698 **NAME**

37699 remque — remove an element from a queue

37700 **SYNOPSIS**

37701 XSI
```
#include <search.h>

void remque(void *element);
```
37703

37704 **DESCRIPTION**

37705 Refer to *insque*( ).

NAME

remquo, remquof, remquol — remainder functions

SYNOPSIS

```
#include <math.h>

double remquo(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
long double remquol(long double x, long double y, int *quo);
```

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 same remainder as the *remainder*( ), *remainderf*( ), and *remainderl*( ) functions, respectively. In the object pointed to by *quo* they store a value whose sign is the sign of *x* /*y* and whose magnitude is congruent modulo 2^n to the magnitude of the integral quotient of *x* /*y*, where *n* is an implementation-defined integer greater than or equal to 3.

An application wishing to check for error situations should set *errno* to 0 before calling these functions. If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

RETURN VALUE

These functions shall return *x* REM *y*.

When *y* is 0, these functions shall return NaN (or equivalent if available) and set *errno* to [EDOM].

If the value of *x* is ±Inf, these functions shall return NaN and set *errno* to [EDOM].

If *x* or *y* is NaN, then these functions shall return NaN and *errno* may be set to [EDOM].

ERRORS

These functions shall fail if:

[EDOM] The *y* argument is 0 or the *x* argument is positive or negative infinity.

These functions may fail if:

[EDOM] The *x* or *y* argument is NaN.

EXAMPLES

None.

APPLICATION USAGE

None.

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

remainder( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

37746 **CHANGE HISTORY**
37747 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

rename — rename a file

SYNOPSIS

```
#include <stdio.h>

int rename(const char *old, const char *new);
```

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 *rename*( ) function shall change the name of a file. The *old* argument points to the path name of the file to be renamed. The *new* argument points to the new path name of the file.

CX If the *old* argument and the *new* argument both refer to, and both link to, the same existing file, *rename*( ) shall return successfully and perform no other action.

If the *old* argument points to the path name of a file that is not a directory, the application shall ensure that the *new* argument does not point to the path name of a directory. If the link named by the *new* argument exists, it shall be removed and *old* renamed to *new*. In this case, a link named *new* shall remain visible to other processes throughout the renaming operation and refer either to the file referred to by *new* or *old* before the operation began. Write access permission is required for both the directory containing *old* and the directory containing *new*.

If the *old* argument points to the path name of a directory, the application shall ensure that the *new* argument does not point to the path name of a file that is not a directory. If the directory named by the *new* argument exists, it shall be removed and *old* renamed to *new*. In this case, a link named *new* shall exist throughout the renaming operation and shall refer either to the directory referred to by *new* or *old* before the operation began. If *new* names an existing directory, the application shall ensure that it is an empty directory.

If either the *old* or the *new* arguments name a symbolic link, *rename*( ) shall operate on the symbolic link itself, and shall not resolve the last component of the argument. If *old* points to a path name that names a symbolic link, the symbolic link shall be renamed. If *new* points to a path name that names a symbolic link, the symbolic link shall be removed.

The application shall ensure that the *new* path name does not contain a path prefix that names *old*. Write access permission is required for the directory containing *old* and the directory containing *new*. If the *old* argument points to the path name of a directory, write access permission may be required for the directory named by *old*, and, if it exists, the directory named by *new*.

If the link named by the *new* argument exists and the file's link count becomes 0 when it is removed and no process has the file open, the space occupied by the file shall be freed and the file shall no longer be accessible. If one or more processes have the file open when the last link is removed, the link shall be removed before *rename*( ) returns, but the removal of the file contents shall be postponed until all references to the file are closed.

Upon successful completion, *rename*( ) shall mark for update the *st_ctime* and *st_mtime* fields of the parent directory of each file.

If the *rename*( ) function fails for any reason other than [EIO], any file named by *new* shall be unaffected.

RETURN VALUE

CX Upon successful completion, *rename*( ) shall return 0; otherwise, –1 shall be returned, *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.

ERRORS

The *rename*( ) function shall fail if:

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.

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.

CX [EEXIST] or [ENOTEMPTY]
The link named by *new* is a directory that is not an empty directory.

CX [EINVAL] The *new* directory path name contains a path prefix that names the *old* directory.

CX [EIO] A physical I/O error has occurred.

CX [EISDIR] The *new* argument points to a directory and the *old* argument points to a file that is not a directory.

CX [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

CX [EMLINK] The file named by *old* is a directory, and the link count of the parent directory of *new* would exceed {LINK_MAX}.

CX [ENAMETOOLONG]
The length of the *old* or *new* argument exceeds {PATH_MAX} or a path name component is longer than {NAME_MAX}.

CX [ENOENT] The link named by *old* does not name an existing file, or either *old* or *new* points to an empty string.

CX [ENOSPC] The directory that would contain *new* cannot be extended.

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.

XSI [EPERM] or [EACCES]
The S_ISVTX flag is set on the directory containing the file referred to by *old* and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges; or *new* refers to an existing file, the S_ISVTX flag is set on the directory containing this file, and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges.

CX [EROFS] The requested operation requires writing in a directory on a read-only file system.

CX [EXDEV] The links named by *new* and *old* are on different file systems and the implementation does not support links between file systems.

The *rename*( ) function may fail if:

37834 XSI [EBUSY] The file named by the *old* or *new* arguments is a named STREAM.

37835 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
37836 resolution of the *path* argument.

37837 CX [ENAMETOOLONG]
37838 As a result of encountering a symbolic link in resolution of the *path* argument,
37839 the length of the substituted path name string exceeded {PATH_MAX}.

37840 CX [ETXTBSY] The file to be renamed is a pure procedure (shared text) file that is being
37841 executed.

37842 **EXAMPLES**

37843 **Renaming a File**

37844 The following example shows how to rename a file named **/home/cnd/mod1** to
37845 **/home/cnd/mod2**.

```
37846 #include <stdio.h>
37847 int status;
37848 ...
37849 status = rename("/home/cnd/mod1", "/home/cnd/mod2");
```

37850 **APPLICATION USAGE**

37851 None.

37852 **RATIONALE**

37853 This *rename*( ) function is equivalent for regular files to that defined by the ISO C standard. Its
37854 inclusion here expands that definition to include actions on directories and specifies behavior
37855 when the *new* parameter names a file that already exists. That specification requires that the
37856 action of the function be atomic.

37857 One of the reasons for introducing this function was to have a means of renaming directories
37858 while permitting implementations to prohibit the use of *link*( ) and *unlink*( ) with directories,
37859 thus constraining links to directories to those made by *mkdir*( ).

37860 The specification that if *old* and *new* refer to the same file is intended to guarantee that:

```
37861 rename("x", "x");
```

37862 does not remove the file.

37863 Renaming dot or dot-dot is prohibited in order to prevent cyclical file system paths.

37864 See also the descriptions of [ENOTEMPTY] and [ENAMETOOLONG] in *rmdir*( ) and [EBUSY] in
37865 *unlink*( ). For a discussion of [EXDEV], see *link*( ).

37866 **FUTURE DIRECTIONS**

37867 None.

37868 **SEE ALSO**

37869 *link*( ), *rmdir*( ), *symlink*( ), *unlink*( ), the Base Definitions volume of IEEE Std. 1003.1-200x,
37870 **<stdio.h>**

37871 **CHANGE HISTORY**

37872 First released in Issue 3.

37873 Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The [EMLINK] error is added to the ERRORS section.

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The type of arguments *old* and *new* are changed from **char*** to **const char***.
- The RETURN VALUE section now states that if an error occurs, neither file is changed or created.

The following change is incorporated for alignment with the FIPS requirements:

- In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path name component is larger that {NAME_MAX}, is now defined as mandatory and marked as an extension.

Issue 4, Version 2

The following changes are made for X/OPEN UNIX conformance:

- The DESCRIPTION is updated to indicate the results of naming a symbolic link in either *old* or *new*.
- In the ERRORS section, [EIO] is added to indicate that a physical I/O error has occurred, [ELOOP] to indicate that too many symbolic links were encountered during path name resolution, and [EPERM] or [EACCES] to indicate a permission check failure when operating on directories with S_ISVTX set.
- In the ERRORS section, a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of path name resolution of a symbolic link.

Issue 5

The [EBUSY] error is added to the ''may fail'' part of the ERRORS section.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

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 DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

NAME

rewind — reset file position indicator in a stream

SYNOPSIS

```
#include <stdio.h>

void rewind(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 call:

```
rewind(stream)
```

shall be equivalent to:

```
(void) fseek(stream, 0L, SEEK_SET)
```

except that *rewind*( ) also clears the error indicator.

Because *rewind*( ) does not return a value, an application wishing to detect errors should clear *errno*, then call *rewind*( ), and if *errno* is non-zero, assume an error has occurred.

RETURN VALUE

The *rewind*( ) function shall return no value.

ERRORS

CX Refer to *fseek*( ) with the exception of [EINVAL] which does not apply.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fseek( ), 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 now marked.

37945 **NAME**

37946 rewinddir — reset position of directory stream to the beginning of a directory

37947 **SYNOPSIS**

37948 `#include <dirent.h>`

37949 `void rewinddir(DIR *dirp);`

37950 **DESCRIPTION**

37951 The *rewinddir*( ) function resets the position of the directory stream to which *dirp* refers to the
37952 beginning of the directory. It shall also cause the directory stream to refer to the current state of
37953 the corresponding directory, as a call to *opendir*( ) would have done. If *dirp* does not refer to a
37954 directory stream, the effect is undefined.

37955 After a call to the *fork*( ) function, either the parent or child (but not both) may continue
37956 XSI processing the directory stream using *readdir*( ), *rewinddir*( ), or *seekdir*( ). If both the parent and
37957 child processes use these functions, the result is undefined.

37958 **RETURN VALUE**

37959 The *rewinddir*( ) function shall not return a value.

37960 **ERRORS**

37961 No errors are defined.

37962 **EXAMPLES**

37963 None.

37964 **APPLICATION USAGE**

37965 The *rewinddir*( ) function should be used in conjunction with *opendir*( ), *readdir*( ), and *closedir*( ) to
37966 examine the contents of the directory. This method is recommended for portability.

37967 **RATIONALE**

37968 None.

37969 **FUTURE DIRECTIONS**

37970 None.

37971 **SEE ALSO**

37972 *closedir*( ), *opendir*( ), *readdir*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<dirent.h>**
37973 **<sys/types.h>**

37974 **CHANGE HISTORY**

37975 First released in Issue 2.

37976 **Issue 4**

37977 The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on
37978 XSI-conformant systems.

37979 The following change is incorporated for alignment with the ISO POSIX-1 standard:

37980 • The last paragraph of the DESCRIPTION, describing a restriction after a *fork*( ) function, is
37981 added.

37982 **Issue 6**

37983 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

37984 The following new requirements on POSIX implementations derive from alignment with the
37985 Single UNIX Specification:

37986 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
37987 required for conforming implementations of previous POSIX specifications, it was not

37988 required for UNIX applications.

37989 **NAME**

37990 rindex — character string operations (**LEGACY**)

37991 **SYNOPSIS**

37992 XSI
```
#include <strings.h>
```

37993
```
char *rindex(const char *s, int c);
```

37994

37995 **DESCRIPTION**

37996 The *rindex*( ) function is identical to *strrchr*( ).

37997 **RETURN VALUE**

37998 Refer to *strrchr*( ).

37999 **ERRORS**

38000 Refer to *strrchr*( ).

38001 **EXAMPLES**

38002 None.

38003 **APPLICATION USAGE**

38004 *strrchr*( ) is preferred over this function.

38005 For maximum portability, it is recommended to replace the function call to *rindex*( ) as follows:

38006
```
#define rindex(a,b) strrchr((a),(b))
```

38007 **RATIONALE**

38008 None.

38009 **FUTURE DIRECTIONS**

38010 This function may be withdrawn in a future version.

38011 **SEE ALSO**

38012 *strrchr*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<strings.h>** |

38013 **CHANGE HISTORY**

38014 First released in Issue 4, Version 2.

38015 **Issue 5**

38016 Moved from X/OPEN UNIX extension to BASE.

38017 **Issue 6**

38018 This function is marked LEGACY.

NAME

rint, rintf, rintl — round-to-nearest integral value

SYNOPSIS

```
#include <math.h>

double rint(double x);
float rintf(float x);
long double rintl(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 integral value (represented as a **double**) nearest *x* in the direction of the current rounding mode. The current rounding mode is implementation-defined.

If the current rounding mode rounds toward negative infinity, then *rint*( ) is identical to *floor*( ). If the current rounding mode rounds toward positive infinity, then *rint*( ) is identical to *ceil*( ).

These functions differ from the *nearbyint*( ), *nearbyintf*( ), and *nearbyintl*( ) functions only in that they may raise the inexact floating-point exception if the result differs in value from the argument.

RETURN VALUE

Upon successful completion, these functions shall return the integer (represented as a double precision number) nearest *x* in the direction of the current rounding mode.

When *x* is ±Inf, *rint*( ) shall return *x*.

If the value of *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

ERRORS

These functions may fail if:

[EDOM] The *x* argument is NaN.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

abs( ), *isnan*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<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 XSI as it is part of the ISO / IEC 9899: 1999 standard.

38063 **NAME**

38064 rmdir — remove a directory

38065 **SYNOPSIS**

38066 `#include <unistd.h>`

38067 `int rmdir(const char *path);`

38068 **DESCRIPTION**

38069 The *rmdir*( ) function shall remove a directory whose name is given by *path*. The directory is
38070 removed only if it is an empty directory.

38071 If the directory is the root directory or the current working directory of any process, it is
38072 unspecified whether the function succeeds, or whether it shall fail and set *errno* to [EBUSY].

38073 If *path* names a symbolic link, then *rmdir*( ) shall fail and set *errno* to [ENOTDIR].

38074 If the *path* argument refers to a path whose final component is either dot or dot-dot, *rmdir*( ) shall
38075 fail.

38076 If the directory's link count becomes 0 and no process has the directory open, the space occupied
38077 by the directory shall be freed and the directory shall no longer be accessible. If one or more
38078 processes have the directory open when the last link is removed, the dot and dot-dot entries, if
38079 present, are removed before *rmdir*( ) returns and no new entries may be created in the directory,
38080 but the directory is not removed until all references to the directory are closed.

38081 If the directory is not an empty directory, *rmdir*( ) shall fail and set *errno* to [EEXIST] or
38082 [ENOTEMPTY].

38083 Upon successful completion, the *rmdir*( ) function shall mark for update the *st_ctime* and
38084 *st_mtime* fields of the parent directory.

38085 **RETURN VALUE**

38086 Upon successful completion, the function *rmdir*( ) shall return 0. Otherwise, –1 shall be returned,
38087 and *errno* set to indicate the error. If –1 is returned, the named directory shall not be changed.

38088 **ERRORS**

38089 The *rmdir*( ) function shall fail if:

38090 [EACCES] Search permission is denied on a component of the path prefix, or write |
38091 permission is denied on the parent directory of the directory to be removed.

38092 [EBUSY] The directory to be removed is currently in use by the system or some process |
38093 and the implementation considers this to be an error.

38094 [EEXIST] or [ENOTEMPTY] |
38095 The *path* argument names a directory that is not an empty directory, or there
38096 are hard links to the directory other than dot or a single entry in dot-dot.

38097 [EINVAL] The *path* argument contains a last component that is dot. |

38098 [EIO] A physical I/O error has occurred. |

38099 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* |
38100 argument. |

38101 [ENAMETOOLONG] |
38102 The length of the *path* argument exceeds {PATH_MAX} or a path name
38103 component is longer than |

38104 NAME_MAX |

[ENOENT] A component of *path* does not name an existing file, or the *path* argument names a nonexistent directory or points to an empty string.

[ENOTDIR] A component of *path* is not a directory.

XSI [EPERM] or [EACCES]
The S_ISVTX flag is set on the parent directory of the directory to be removed and the caller is not the owner of the directory to be removed, nor is the caller the owner of the parent directory, nor does the caller have the appropriate privileges.

[EROFS] The directory entry to be removed resides on a read-only file system.

The *rmdir*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted path name 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.

IEEE Std. 1003.1-200x 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 process's home directory 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 (on page 521) 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.

FUTURE DIRECTIONS

None.

SEE ALSO

mkdir( ), *remove*( ), *unlink*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The **<unistd.h>** header is added to the SYNOPSIS section.

The [ENAMETOOLONG] description is amended.

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument *path* is changed from **char*** to **const char***.
- The DESCRIPTION is expanded to indicate that, if the directory is a root directory or a current working directory, it is unspecified whether the function succeeds, or whether it fails and sets *errno* to [EBUSY]. In Issue 3, the behavior under these circumstances was defined as implementation-defined.
- The RETURN VALUE section is expanded to direct that if –1 is returned, the directory is not changed.

The following change is incorporated for alignment with the FIPS requirements:

- In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path name component is larger that {NAME_MAX} is now defined as mandatory and marked as an extension.

Issue 4, Version 2

The following changes are made for X/OPEN UNIX conformance:

- The DESCRIPTION is updated to indicate the results of naming a symbolic link in *path*.
- In the ERRORS section, [EIO] is added to indicate that a physical I/O error has occurred, [ELOOP] to indicate that too many symbolic links were encountered during path name resolution, and [EPERM] or [EACCES] to indicate a permission check failure when operating on directories with S_ISVTX set.
- In the ERRORS section, a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of path name resolution of a symbolic link.

Issue 6

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

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.

38192 **NAME**

38193 round, roundf, roundl — round to nearest integer value in floating-point format

38194 **SYNOPSIS**

38195 `#include <math.h>`

38196 `double round(double x);`
38197 `float roundf(float x);`
38198 `long double roundl(long double x);`

38199 **DESCRIPTION**

38200 CX The functionality described on this reference page is aligned with the ISO C standard. Any
38201 conflict between the requirements described here and the ISO C standard is unintentional. This
38202 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

38203 These functions shall round their argument to the nearest integer value in floating-point format,
38204 rounding halfway cases away from zero, regardless of the current rounding direction.

38205 An application wishing to check for error situations should set *errno* to 0 before calling these
38206 functions. If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

38207 **RETURN VALUE**

38208 Upon successful completion, these functions shall return the rounded integer value.

38209 If *x* is ±Inf, these functions shall return *x*.

38210 If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

38211 **ERRORS**

38212 These functions may fail if:

38213 [EDOM] The value of *x* is NaN.

38214 **EXAMPLES**

38215 None.

38216 **APPLICATION USAGE**

38217 None.

38218 **RATIONALE**

38219 None.

38220 **FUTURE DIRECTIONS**

38221 None.

38222 **SEE ALSO**

38223 The Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

38224 **CHANGE HISTORY**

38225 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

38226 **NAME**

38227 scalb — load exponent of a radix-independent floating-point number

38228 **SYNOPSIS**

38229 XSI
```
#include <math.h>

double scalb(double x, double n);
```
38230

38231

38232 **DESCRIPTION**

38233 The *scalb*( ) function shall compute $x*r^n$, where *r* is the radix of the machine's floating point
38234 arithmetic. When *r* is 2, *scalb*( ) is equivalent to *ldexp*( ). The value of *r* is FLT_RADIX which is
38235 defined in **<float.h>**.

38236 An application wishing to check for error situations should set *errno* to 0 before calling *scalb*( ). If
38237 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

38238 **RETURN VALUE**

38239 Upon successful completion, the *scalb*( ) function shall return $x*r^n$.

38240 If the correct value would overflow, *scalb*( ) shall return ±HUGE_VAL (according to the sign of *x*)
38241 and set *errno* to [ERANGE].

38242 If the correct value would underflow, *scalb*( ) shall return 0 and set *errno* to [ERANGE].

38243 The *scalb*( ) function shall return *x* when *x* is ±Inf.

38244 If *x* or *n* is NaN, then *scalb*( ) shall return NaN and may set *errno* to [EDOM].

38245 **ERRORS**

38246 The *scalb*( ) function shall fail if:

38247 [ERANGE] The correct value would overflow or underflow.

38248 The *scalb*( ) function may fail if:

38249 [EDOM] The *x* or *n* argument is NaN.

38250 **EXAMPLES**

38251 None.

38252 **APPLICATION USAGE**

38253 None.

38254 **RATIONALE**

38255 None.

38256 **FUTURE DIRECTIONS**

38257 None.

38258 **SEE ALSO**

38259 *ilogb*( ), *ldexp*( ), *logb*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<float.h>**,
38260 **<math.h>**

38261 **CHANGE HISTORY**

38262 First released in Issue 4, Version 2.

38263 **Issue 5**

38264 Moved from X/OPEN UNIX extension to BASE.

38265 The DESCRIPTION is updated to indicate how an application should check for an error. This
38266 text was previously published in the APPLICATION USAGE section.

NAME

scalbln, scalblnf, scalblnl scalbn, scalbnf, scalbnl, — compute exponent using FLT_RADIX

SYNOPSIS

```
#include <math.h>

double scalbln(double x, long n);
float scalblnf(float x, long n);
long double scalblnl(long double x, long n);
double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int 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.

These functions shall compute $x * \text{FLT\_RADIX}^n$ efficiently, not normally by computing FLT_RADIX^n explicitly.

An application wishing to check for error situations should set *errno* to 0 before calling these functions. If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return $x * \text{FLT\_RADIX}^n$.

If the correct value would overflow, these functions shall return ±HUGE_VAL (according to the sign of *x*) and set *errno* to [ERANGE].

If the correct value would underflow, these functions shall return 0 and set *errno* to [ERANGE].

These functions shall return *x* when *x* is ±Inf.

If *x* or *n* is NaN, then these functions shall return NaN and may set *errno* to [EDOM].

ERRORS

These functions shall fail if:

[ERANGE] The correct value would overflow or underflow.

These functions may fail if:

[EDOM] The *x* or *n* argument is NaN.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

These functions are named so as to avoid conflicting with the Single UNIX Specification, which has a *scalb*() function whose second argument is **double** instead of **int**. The *scalb*() function is not part of ISO C standard. These functions, whose second parameter has type **long**, is provided because the factor required to scale from the smallest positive floating-point value to the largest finite one, on many implementations, is too large to represent in the minimum-width **int** format.

38307 **FUTURE DIRECTIONS**
38308 None.

38309 **SEE ALSO**
38310 *scalb*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

38311 **CHANGE HISTORY**
38312 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

38313 **NAME**

38314 scanf — convert formatted input

38315 **SYNOPSIS**

38316 `#include <stdio.h>`

38317 `int scanf(const char *restrict format, ... );` |

38318 **DESCRIPTION** |

38319 Refer to *fscanf*( ).

38320 **NAME**

38321 sched_get_priority_max, sched_get_priority_min — get priority limits (**REALTIME**)

38322 **SYNOPSIS**

38323 PS
```
#include <sched.h>

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```
38326

38327 **DESCRIPTION**

38328 The *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions return the appropriate
38329 maximum or minimum, respectively, for the scheduling policy specified by *policy*.

38330 The value of *policy* is one of the scheduling policy values defined in **<sched.h>**.

38331 **RETURN VALUE**

38332 If successful, the *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions shall return the
38333 appropriate maximum or minimum values, respectively. If unsuccessful, they shall return a
38334 value of –1 and set *errno* to indicate the error.

38335 **ERRORS**

38336 The *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions shall fail if:

38337 [EINVAL] The value of the *policy* parameter does not represent a defined scheduling |
38338 policy.

38339 **EXAMPLES**

38340 None.

38341 **APPLICATION USAGE**

38342 None.

38343 **RATIONALE**

38344 None.

38345 **FUTURE DIRECTIONS**

38346 None.

38347 **SEE ALSO**

38348 *sched_getparam*( ), *sched_setparam*( ), *sched_getscheduler*( ), *sched_rr_get_interval*( ),
38349 *sched_setscheduler*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sched.h>** |

38350 **CHANGE HISTORY**

38351 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38352 **Issue 6**

38353 These functions are marked as part of the Process Scheduling option. |

38354 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38355 implementation does not support the Process Scheduling option. |

38356 The [ESRCH] error condition has been removed since these functions do not take a *pid*
38357 argument.

NAME

sched_getparam — get scheduling parameters (**REALTIME**)

SYNOPSIS

PS
```
#include <sched.h>

int sched_getparam(pid_t pid, struct sched_param *param);
```

DESCRIPTION

The *sched_getparam*( ) function shall return the scheduling parameters of a process specified by *pid* in the **sched_param** structure pointed to by *param*.

If a process specified by *pid* exists, and if the calling process has permission, the scheduling parameters for the process whose process ID is equal to *pid* shall be returned.

If *pid* is zero, the scheduling parameters for the calling process shall be returned. The behavior of the *sched_getparam*( ) function is unspecified if the value of *pid* is negative.

RETURN VALUE

Upon successful completion, the *sched_getparam*( ) function shall return zero. If the call to *sched_getparam*( ) is unsuccessful, the function shall return a value of –1 and set *errno* to indicate the error.

ERRORS

The *sched_getparam*( ) function shall fail if:

[EPERM] The requesting process does not have permission to obtain the scheduling parameters of the specified process.

[ESRCH] No process can be found corresponding to that specified by *pid*.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sched_getscheduler( ), *sched_setparam*( ), *sched_setscheduler*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sched.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sched_getparam*( ) function is marked as part of the Process Scheduling option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Scheduling option.

38397 **NAME**

38398 sched_getscheduler — get scheduling policy (**REALTIME**)

38399 **SYNOPSIS**

38400 PS
```
#include <sched.h>
```

38401
```
int sched_getscheduler(pid_t pid);
```

38402

38403 **DESCRIPTION**

38404 The *sched_getscheduler*( ) function shall return the scheduling policy of the process specified by
38405 *pid*. If the value of *pid* is negative, the behavior of the *sched_getscheduler*( ) function is
38406 unspecified.

38407 The values that can be returned by *sched_getscheduler*( ) are defined in the header file **<sched.h>**.

38408 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
38409 policy shall be returned for the process whose process ID is equal to *pid*.

38410 If *pid* is zero, the scheduling policy shall be returned for the calling process.

38411 **RETURN VALUE**

38412 Upon successful completion, the *sched_getscheduler*( ) function shall return the scheduling policy
38413 of the specified process. If unsuccessful, the function shall return –1 and set *errno* to indicate the
38414 error.

38415 **ERRORS**

38416 The *sched_getscheduler*( ) function shall fail if:

38417 [EPERM] The requesting process does not have permission to determine the scheduling |
38418 policy of the specified process.

38419 [ESRCH] No process can be found corresponding to that specified by *pid*. |

38420 **EXAMPLES**

38421 None.

38422 **APPLICATION USAGE**

38423 None.

38424 **RATIONALE**

38425 None.

38426 **FUTURE DIRECTIONS**

38427 None.

38428 **SEE ALSO**

38429 *sched_getparam*( ), *sched_setparam*( ), *sched_setscheduler*( ), the Base Definitions volume of |
38430 IEEE Std. 1003.1-200x, **<sched.h>** |

38431 **CHANGE HISTORY**

38432 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38433 **Issue 6**

38434 The *sched_getscheduler*( ) function is marked as part of the Process Scheduling option. |

38435 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38436 implementation does not support the Process Scheduling option. |

NAME

sched_rr_get_interval — get execution time limits (**REALTIME**)

SYNOPSIS

PS
```
#include <sched.h>

int sched_rr_get_interval(pid_t pid, struct timespec *interval);
```

DESCRIPTION

The *sched_rr_get_interval*( ) function updates the **timespec** structure referenced by the *interval* argument to contain the current execution time limit (that is, time quantum) for the process specified by *pid*. If *pid* is zero, the current execution time limit for the calling process shall be returned.

RETURN VALUE

If successful, the *sched_rr_get_interval*( ) function shall return zero. Otherwise, it shall return a value of –1 and set *errno* to indicate the error.

ERRORS

The *sched_rr_get_interval*( ) function shall fail if:

[ESRCH] No process can be found corresponding to that specified by *pid*.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sched_getparam( ), *sched_get_priority_max*( ), *sched_getscheduler*( ), *sched_setparam*( ), *sched_setscheduler*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sched.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sched_rr_get_interval*( ) function is marked as part of the Process Scheduling option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Scheduling option.

38471 **NAME**

38472 sched_setparam — set scheduling parameters (**REALTIME**)

38473 **SYNOPSIS**

38474 PS
```
#include <sched.h>

int sched_setparam(pid_t pid, const struct sched_param *param);
```
38475

38476

38477 **DESCRIPTION**

38478 The *sched_setparam*( ) function sets the scheduling parameters of the process specified by *pid* to
38479 the values specified by the **sched_param** structure pointed to by *param*. The value of the
38480 *sched_priority* member in the **sched_param** structure is any integer within the inclusive priority
38481 range for the current scheduling policy of the process specified by *pid*. Higher numerical values
38482 for the priority represent higher priorities. If the value of *pid* is negative, the behavior of the
38483 *sched_setparam*( ) function is unspecified.

38484 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
38485 parameters shall be set for the process whose process ID is equal to *pid*.

38486 If *pid* is zero, the scheduling parameters shall be set for the calling process.

38487 The conditions under which one process has permission to change the scheduling parameters of
38488 another process are implementation-defined.

38489 Implementations may require the requesting process to have the appropriate privilege to set its
38490 own scheduling parameters or those of another process.

38491 The target process, whether it is running or not running, resumes execution after all other
38492 runnable processes of equal or greater priority have been scheduled to run.

38493 If the priority of the process specified by the *pid* argument is set higher than that of the lowest
38494 priority running process and if the specified process is ready to run, the process specified by the
38495 *pid* argument preempts a lowest priority running process. Similarly, if the process calling
38496 *sched_setparam*( ) sets its own priority lower than that of one or more other non-empty process
38497 lists, then the process that is the head of the highest priority list also preempts the calling
38498 process. Thus, in either case, the originating process might not receive notification of the
38499 completion of the requested priority change until the higher priority process has executed.

38500 SS If the scheduling policy of the target process is SCHED_SPORADIC, the value specified by the
38501 *sched_ss_low_priority* member of the *param* argument shall be any integer within the inclusive
38502 priority range for the sporadic server policy. The *sched_ss_repl_period* and *sched_ss_init_budget*
38503 members of the *param* argument shall represent the time parameters to be used by the sporadic
38504 server scheduling policy for the target process. The *sched_ss_max_repl* member of the *param*
38505 argument shall represent the maximum number of replenishments that are allowed to be
38506 pending simultaneously for the process scheduled under this scheduling policy.

38507 The specified *sched_ss_repl_period* shall be greater than or equal to the specified
38508 *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.

38509 The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the
38510 function to succeed; if not, the function shall fail.

38511 If the scheduling policy of the target process is either SCHED_FIFO or SCHED_RR, the
38512 *sched_ss_low_priority*, *sched_ss_repl_period*, and *sched_ss_init_budget* members of the *param*
38513 argument shall have no effect on the scheduling behavior. If the scheduling policy of this process
38514 is not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC, including SCHED_OTHER, the
38515 effects of these members shall be implementation-defined.

Notes to Reviewers

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

D3, XSH, ERN 502 questions "including SCHED_OTHER" above. (Problem left over from D2.)

If the current scheduling policy for the process specified by *pid* is not SCHED_FIFO, SS SCHED_RR, or SCHED_SPORADIC , the result is implementation-defined; this case includes the SCHED_OTHER policy.

The effect of this function on individual threads is dependent on the scheduling contention scope of the threads:

- For threads with system scheduling contention scope, these functions have no effect on their scheduling.
- For threads with process scheduling contention scope, the threads' scheduling parameters shall not be affected. However, the scheduling of these threads with respect to threads in other processes may be dependent on the scheduling parameters of their process, which are governed using these functions.

If an implementation supports a two-level scheduling model in which library threads are multiplexed on top of several kernel-scheduled entities, then the underlying kernel-scheduled entities for the system contention scope threads shall not be affected by these functions.

The underlying kernel-scheduled entities for the process contention scope threads shall have their scheduling parameters changed to the value specified in *param*. Kernel scheduled entities for use by process contention scope threads that are created after this call completes inherit their scheduling policy and associated scheduling parameters from the process.

This function is not atomic with respect to other threads in the process. Threads are allowed to continue to execute while this function call is in the process of changing the scheduling policy for the underlying kernel-scheduled entities used by the process contention scope threads.

RETURN VALUE

If successful, the *sched_setparam*( ) function shall return zero.

If the call to *sched_setparam*( ) is unsuccessful, the priority shall remain unchanged, and the function shall return a value of –1 and set *errno* to indicate the error.

ERRORS

The *sched_setparam*( ) function shall fail if:

[EINVAL] One or more of the requested scheduling parameters is outside the range defined for the scheduling policy of the specified *pid*.

[EPERM] The requesting process does not have permission to set the scheduling parameters for the specified process, or does not have the appropriate privilege to invoke *sched_setparam*( ).

[ESRCH] No process can be found corresponding to that specified by *pid*.

38552 **EXAMPLES**

38553 None.

38554 **APPLICATION USAGE**

38555 None.

38556 **RATIONALE**

38557 None.

38558 **FUTURE DIRECTIONS**

38559 None.

38560 **SEE ALSO**

38561 *sched_getparam*( ), *sched_getscheduler*( ), *sched_setscheduler*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sched.h>**

38563 **CHANGE HISTORY**

38564 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38565 **Issue 6**

38566 The *sched_setparam*( ) function is marked as part of the Process Scheduling option.

38567 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38568 implementation does not support the Process Scheduling option.

38569 The following new requirements on POSIX implementations derive from alignment with the
38570 Single UNIX Specification:

38571 • In the DESCRIPTION, the effect of this function on a thread's scheduling parameters is
38572 added.

38573 • Sections describing two-level scheduling and atomicity of the function are added.

38574 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std. 1003.1d-1999.

NAME

sched_setscheduler — set scheduling policy and parameters (**REALTIME**)

SYNOPSIS

PS
```
#include <sched.h>

int sched_setscheduler(pid_t pid, int policy,
    const struct sched_param *param);
```

DESCRIPTION

The *sched_setscheduler*( ) function sets the scheduling policy and scheduling parameters of the process specified by *pid* to *policy* and the parameters specified in the **sched_param** structure pointed to by *param*, respectively. The value of the *sched_priority* member in the **sched_param** structure is any integer within the inclusive priority range for the scheduling policy specified by *policy*. If the value of *pid* is negative, the behavior of the *sched_setscheduler*( ) function is unspecified.

The possible values for the *policy* parameter are defined in the header file **<sched.h>**.

If a process specified by *pid* exists, and if the calling process has permission, the scheduling policy and scheduling parameters shall be set for the process whose process ID is equal to *pid*.

If *pid* is zero, the scheduling policy and scheduling parameters shall be set for the calling process.

The conditions under which one process has the appropriate privilege to change the scheduling parameters of another process are implementation-defined.

Implementations may require that the requesting process have permission to set its own scheduling parameters or those of another process. Additionally, implementation-defined restrictions may apply as to the appropriate privileges required to set a process' own scheduling policy, or another process' scheduling policy, to a particular value.

The *sched_setscheduler*( ) function is considered successful if it succeeds in setting the scheduling policy and scheduling parameters of the process specified by *pid* to the values specified by *policy* and the structure pointed to by *param*, respectively.

SS If the scheduling policy specified by *policy* is SCHED_SPORADIC, the value specified by the *sched_ss_low_priority* member of the *param* argument shall be any integer within the inclusive priority range for the sporadic server policy. The *sched_ss_repl_period* and *sched_ss_init_budget* members of the *param* argument shall represent the time parameters used by the sporadic server scheduling policy for the target process. The *sched_ss_max_repl* member of the *param* argument shall represent the maximum number of replenishments that are allowed to be pending simultaneously for the process scheduled under this scheduling policy.

The specified *sched_ss_repl_period* shall be greater than or equal to the specified *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.

The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the function to succeed; if not, the function shall fail.

If the scheduling policy specified by *policy* is either SCHED_FIFO or SCHED_RR, the *sched_ss_low_priority*, *sched_ss_repl_period*, and *sched_ss_init_budget* members of the *param* argument shall have no effect on the scheduling behavior.

The effect of this function on individual threads is dependent on the scheduling contention scope of the threads:

- For threads with system scheduling contention scope, these functions have no effect on their scheduling.
- For threads with process scheduling contention scope, the threads' scheduling policy and associated parameters shall not be affected. However, the scheduling of these threads with respect to threads in other processes may be dependent on the scheduling parameters of their process, which are governed using these functions.

If an implementation supports a two-level scheduling model in which library threads are multiplexed on top of several kernel-scheduled entities, then the underlying kernel-scheduled entities for the system contention scope threads shall not be affected by these functions.

The underlying kernel-scheduled entities for the process contention scope threads shall have their scheduling policy and associated scheduling parameters changed to the values specified in *policy* and *param*, respectively. Kernel scheduled entities for use by process contention scope threads that are created after this call completes inherit their scheduling policy and associated scheduling parameters from the process.

This function is not atomic with respect to other threads in the process. Threads are allowed to continue to execute while this function call is in the process of changing the scheduling policy and associated scheduling parameters for the underlying kernel-scheduled entities used by the process contention scope threads.

RETURN VALUE

Upon successful completion, the function shall return the former scheduling policy of the specified process. If the *sched_setscheduler*( ) function fails to complete successfully, the policy and scheduling paramenters shall remain unchanged, and the function shall return a value of –1 and set *errno* to indicate the error.

ERRORS

The *sched_setscheduler*( ) function shall fail if:

[EINVAL] The value of the *policy* parameter is invalid, or one or more of the parameters contained in *param* is outside the valid range for the specified scheduling policy.

[EPERM] The requesting process does not have permission to set either or both of the scheduling parameters or the scheduling policy of the specified process.

[ESRCH] No process can be found corresponding to that specified by *pid*.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sched_getparam( ), *sched_getscheduler*( ), *sched_setparam*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sched.h>**

38661 **CHANGE HISTORY**

38662 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38663 **Issue 6**

38664 The *sched_setscheduler*( ) function is marked as part of the Process Scheduling option.

38665 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38666 implementation does not support the Process Scheduling option.

38667 The following new requirements on POSIX implementations derive from alignment with the
38668 Single UNIX Specification:

38669 • In the DESCRIPTION, the effect of this function on a thread's scheduling parameters is
38670 added.

38671 • Sections describing two-level scheduling and atomicity of the function are added.

38672 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std. 1003.1d-1999.

38673 **NAME**

38674 sched_yield — yield processor

38675 **SYNOPSIS**

38676 PS|THR
```
#include <sched.h>
```
38677
```
int sched_yield(void);
```
38678

38679 **DESCRIPTION**

38680 The *sched_yield*( ) function forces the running thread to relinquish the processor until it again
38681 becomes the head of its thread list. It takes no arguments.

38682 **RETURN VALUE**

38683 The *sched_yield*( ) function shall return 0 if it completes successfully; otherwise, it shall return a
38684 value of –1 and set *errno* to indicate the error.

38685 **ERRORS**

38686 No errors are defined.

38687 **EXAMPLES**

38688 None.

38689 **APPLICATION USAGE**

38690 None.

38691 **RATIONALE**

38692 None.

38693 **FUTURE DIRECTIONS**

38694 None.

38695 **SEE ALSO**

38696 The Base Definitions volume of IEEE Std. 1003.1-200x, **<sched.h>** |

38697 **CHANGE HISTORY**

38698 First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the
38699 POSIX Threads Extension.

38700 **Issue 6**

38701 The *sched_yield*( ) function is now marked as part of the Process Scheduling and Threads options. |

38702 **NAME**

38703 seed48 — seed uniformly distributed pseudo-random non-negative long integer generator

38704 **SYNOPSIS**

38705 XSI
```
#include <stdlib.h>

unsigned short *seed48(unsigned short seed16v[3]);
```
38706 |

38707 |

38708 **DESCRIPTION**

38709 Refer to *drand48*( ).

38710 **NAME**

38711 seekdir — set position of directory stream

38712 **SYNOPSIS**

38713 XSI
```
#include <dirent.h>

void seekdir(DIR *dirp, long loc);
```

38714 |

38715 |

38716 **DESCRIPTION**

38717 The *seekdir*( ) function sets the position of the next *readdir*( ) operation on the directory stream
38718 specified by *dirp* to the position specified by *loc*. The value of *loc* should have been returned
38719 from an earlier call to *telldir*( ). The new position reverts to the one associated with the directory
38720 stream when *telldir*( ) was performed.

38721 If the value of *loc* was not obtained from an earlier call to *telldir*( ), or if a call to *rewinddir*( )
38722 occurred between the call to *telldir*( ) and the call to *seekdir*( ), the results of subsequent calls to
38723 *readdir*( ) are unspecified.

38724 **RETURN VALUE**

38725 The *seekdir*( ) function shall return no value.

38726 **ERRORS**

38727 No errors are defined.

38728 **EXAMPLES**

38729 None.

38730 **APPLICATION USAGE**

38731 None.

38732 **RATIONALE**

38733 None.

38734 **FUTURE DIRECTIONS**

38735 None.

38736 **SEE ALSO**

38737 *opendir*( ), *readdir*( ), *telldir*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<dirent.h>**, |
38738 **<stdio.h>**, **<sys/types.h>**

38739 **CHANGE HISTORY**

38740 First released in Issue 2.

38741 **Issue 4**

38742 The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on
38743 XSI-conformant systems.

38744 The type of argument *loc* is expanded to **long**. |

38745 **Issue 4, Version 2**

38746 The DESCRIPTION is updated for X/OPEN UNIX conformance to indicate that a call to
38747 *readdir*( ) may produce unspecified results if either *loc* was not obtained by a previous call to
38748 *telldir*( ), or if there is an intervening call to *rewinddir*( ).

38749 **Issue 6**

38750 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

38751 **NAME**

38752 select — synchronous I/O multiplexing

38753 **SYNOPSIS**

38754 `#include <sys/time.h>`

38755 `int select(int` *`nfds`*`, fd_set *restrict` *`readfds`*`, fd_set *restrict` *`writefds`*`,`|
38756 `    fd_set *restrict` *`errorfds`*`, struct timeval *restrict` *`timeout`*`);` |
38757 |

38758 **DESCRIPTION** |

38759 Refer to *pselect*( ). |

38760 **NAME**

38761 sem_close — close a named semaphore (**REALTIME**)

38762 **SYNOPSIS**

38763 SEM
```
#include <semaphore.h>

int sem_close(sem_t *sem);
```
38764

38765

38766 **DESCRIPTION**

38767 The *sem_close*( ) function is used to indicate that the calling process is finished using the named
38768 semaphore indicated by *sem*. The effects of calling *sem_close*( ) for an unnamed semaphore (one
38769 created by *sem_init*( )) are undefined. The *sem_close*( ) function deallocates (that is, makes
38770 available for reuse by a subsequent *sem_open*( ) by this process) any system resources allocated
38771 by the system for use by this process for this semaphore. The effect of subsequent use of the
38772 semaphore indicated by *sem* by this process is undefined. If the semaphore has not been
38773 removed with a successful call to *sem_unlink*( ), then *sem_close*( ) has no effect on the state of the
38774 semaphore. If the *sem_unlink*( ) function has been successfully invoked for *name* after the most
38775 recent call to *sem_open*( ) with O_CREAT for this semaphore, then when all processes that have
38776 opened the semaphore close it, the semaphore is no longer accessible.

38777 **RETURN VALUE**

38778 Upon successful completion, a value of zero shall be returned. Otherwise, a value of –1 shall be
38779 returned and *errno* set to indicate the error.

38780 **ERRORS**

38781 The *sem_close*( ) function shall fail if:

38782 [EINVAL] The *sem* argument is not a valid semaphore descriptor. |

38783 **EXAMPLES**

38784 None.

38785 **APPLICATION USAGE**

38786 The *sem_close*( ) function is part of the Semaphores option and need not be available on all |
38787 implementations.

38788 **RATIONALE**

38789 None.

38790 **FUTURE DIRECTIONS**

38791 None.

38792 **SEE ALSO**

38793 *semctl*( ), *semget*( ), *semop*( ), *sem_init*( ), *sem_open*( ), *sem_unlink*( ), the Base Definitions volume of |
38794 IEEE Std. 1003.1-200x, **<semaphore.h>** |

38795 **CHANGE HISTORY**

38796 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38797 **Issue 6**

38798 The *sem_close*( ) function is marked as part of the Semaphores option. |

38799 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38800 implementation does not support the Semaphores option. |

38801 **NAME**

38802 sem_destroy — destroy an unnamed semaphore (**REALTIME**)

38803 **SYNOPSIS**

38804 SEM
```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```
38805

38806

38807 **DESCRIPTION**

38808 The *sem_destroy*( ) function is used to destroy the unnamed semaphore indicated by *sem*. Only a
38809 semaphore that was created using *sem_init*( ) may be destroyed using *sem_destroy*( ); the effect of
38810 calling *sem_destroy*( ) with a named semaphore is undefined. The effect of subsequent use of the
38811 semaphore *sem* is undefined until *sem* is re-initialized by another call to *sem_init*( ).

38812 It is safe to destroy an initialized semaphore upon which no threads are currently blocked. The
38813 effect of destroying a semaphore upon which other threads are currently blocked is undefined.

38814 **RETURN VALUE**

38815 Upon successful completion, a value of zero shall be returned. Otherwise, a value of –1 shall be
38816 returned and *errno* set to indicate the error.

38817 **ERRORS**

38818 The *sem_destroy*( ) function shall fail if:

38819 [EINVAL] The *sem* argument is not a valid semaphore. |

38820 The *sem_destroy*( ) function may fail if:

38821 [EBUSY] There are currently processes blocked on the semaphore. |

38822 **EXAMPLES**

38823 None.

38824 **APPLICATION USAGE**

38825 The *sem_destroy*( ) function is part of the Semaphores option and need not be available on all |
38826 implementations.

38827 **RATIONALE**

38828 None.

38829 **FUTURE DIRECTIONS**

38830 None.

38831 **SEE ALSO**

38832 *semctl*( ), *semget*( ), *semop*( ), *sem_init*( ), *sem_open*( ), the Base Definitions volume of |
38833 IEEE Std. 1003.1-200x, **<semaphore.h>** |

38834 **CHANGE HISTORY**

38835 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38836 **Issue 6**

38837 The *sem_destroy*( ) function is marked as part of the Semaphores option. |

38838 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38839 implementation does not support the Semaphores option. |

38840 **NAME**

38841 sem_getvalue — get the value of a semaphore (**REALTIME**)

38842 **SYNOPSIS**

38843 SEM
```
#include <semaphore.h>

int sem_getvalue(sem_t *restrict sem, int *restrict sval);
```

38844

38845

38846 **DESCRIPTION**

38847 The *sem_getvalue*( ) function updates the location referenced by the *sval* argument to have the
38848 value of the semaphore referenced by *sem* without affecting the state of the semaphore. The
38849 updated value represents an actual semaphore value that occurred at some unspecified time
38850 during the call, but it need not be the actual value of the semaphore when it is returned to the
38851 calling process.

38852 If *sem* is locked, then the value returned by *sem_getvalue*( ) is either zero or a negative number
38853 whose absolute value represents the number of processes waiting for the semaphore at some
38854 unspecified time during the call.

38855 **RETURN VALUE**

38856 Upon successful completion, the *sem_getvalue*( ) function shall return a value of zero. Otherwise,
38857 it shall return a value of –1 and set *errno* to indicate the error.

38858 **ERRORS**

38859 The *sem_getvalue*( ) function shall fail if:

38860 [EINVAL] The *sem* argument does not refer to a valid semaphore.

38861 **EXAMPLES**

38862 None.

38863 **APPLICATION USAGE**

38864 The *sem_getvalue*( ) function is part of the Semaphores option and need not be available on all
38865 implementations.

38866 **RATIONALE**

38867 None.

38868 **FUTURE DIRECTIONS**

38869 None.

38870 **SEE ALSO**

38871 *semctl*( ), *semget*( ), *semop*( ), *sem_post*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_wait*( ), the Base
38872 Definitions volume of IEEE Std. 1003.1-200x, **<semaphore.h>**

38873 **CHANGE HISTORY**

38874 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38875 **Issue 6**

38876 The *sem_getvalue*( ) function is marked as part of the Semaphores option.

38877 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38878 implementation does not support the Semaphores option.

38879 The *sem_timedwait*( ) function is added to the SEE ALSO section for alignment with
38880 IEEE Std. 1003.1d-1999.

38881 The **restrict** keyword is added to the *sem_getvalue*( ) prototype for alignment with the
38882 ISO / IEC 9899: 1999 standard.

38883 **NAME**

38884 sem_init — initialize an unnamed semaphore (**REALTIME**)

38885 **SYNOPSIS**

38886 SEM
```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned value);
```

38887 |

38888 |

38889 **DESCRIPTION**

38890 The *sem_init*( ) function is used to initialize the unnamed semaphore referred to by *sem*. The
38891 value of the initialized semaphore is *value*. Following a successful call to *sem_init*( ), the
38892 semaphore may be used in subsequent calls to *sem_wait*( ), *sem_trywait*( ), *sem_post*( ), and
38893 *sem_destroy*( ). This semaphore remains usable until the semaphore is destroyed.

38894 If the *pshared* argument has a non-zero value, then the semaphore is shared between processes;
38895 in this case, any process that can access the semaphore *sem* can use *sem* for performing
38896 *sem_wait*( ), *sem_trywait*( ), *sem_post*( ), and *sem_destroy*( ) operations.

38897 Only *sem* itself may be used for performing synchronization. The result of referring to copies of
38898 *sem* in calls to *sem_wait*( ), *sem_trywait*( ), *sem_post*( ), and *sem_destroy*( ), is undefined.

38899 If the *pshared* argument is zero, then the semaphore is shared between threads of the process; any
38900 thread in this process can use *sem* for performing *sem_wait*( ), *sem_trywait*( ), *sem_post*( ), and
38901 *sem_destroy*( ) operations. The use of the semaphore by threads other than those created in the
38902 same process is undefined.

38903 Attempting to initialize an already initialized semaphore results in undefined behavior.

38904 **RETURN VALUE**

38905 Upon successful completion, the *sem_init*( ) function shall initialize the semaphore in *sem*.
38906 Otherwise, it shall return –1 and set *errno* to indicate the error.

38907 **ERRORS**

38908 The *sem_init*( ) function shall fail if:

38909 [EINVAL] The *value* argument exceeds {SEM_VALUE_MAX}. |

38910 [ENOSPC] A resource required to initialize the semaphore has been exhausted, or the |
38911 limit on semaphores ({SEM_NSEMS_MAX}) has been reached.

38912 [EPERM] The process lacks the appropriate privileges to initialize the semaphore. |

38913 **EXAMPLES**

38914 None.

38915 **APPLICATION USAGE**

38916 The *sem_init*( ) function is part of the Semaphores option and need not be available on all |
38917 implementations.

38918 **RATIONALE**

38919 Although this volume of IEEE Std. 1003.1-200x fails to specify a successful return value, it is
38920 likely that a later version may require the implementation to return a value of zero if the call to
38921 *sem_init*( ) is successful.

38922 **FUTURE DIRECTIONS**

38923 None.

38924 **SEE ALSO**

38925 *sem_destroy*( ), *sem_post*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_wait*( ), the Base Definitions |
38926 volume of IEEE Std. 1003.1-200x, **<semaphore.h>** |

38927 **CHANGE HISTORY**

38928 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38929 **Issue 6**

38930 The *sem_init*( ) function is marked as part of the Semaphores option. |

38931 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38932 implementation does not support the Semaphores option. |

38933 The *sem_timedwait*( ) function is added to the SEE ALSO section for alignment with
38934 IEEE Std. 1003.1d-1999.

NAME

sem_open — initialize and open a named semaphore (**REALTIME**)

SYNOPSIS

SEM

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ...);
```

DESCRIPTION

The *sem_open*( ) function establishes a connection between a named semaphore and a process. Following a call to *sem_open*( ) with semaphore name *name*, the process may reference the semaphore associated with *name* using the address returned from the call. This semaphore may be used in subsequent calls to *sem_wait*( ), *sem_trywait*( ), *sem_post*( ), and *sem_close*( ). The semaphore remains usable by this process until the semaphore is closed by a successful call to *sem_close*( ), *_exit*( ), or one of the *exec* functions.

The *oflag* argument controls whether the semaphore is created or merely accessed by the call to *sem_open*( ). The following flag bits may be set in *oflag*:

O_CREAT This flag is used to create a semaphore if it does not already exist. If O_CREAT is set and the semaphore already exists, then O_CREAT has no effect, except as noted under O_EXCL. Otherwise, *sem_open*( ) creates a named semaphore. The O_CREAT flag requires a third and a fourth argument: *mode*, which is of type **mode_t**, and *value*, which is of type **unsigned**. The semaphore is created with an initial value of *value*. Valid initial values for semaphores are less than or equal to {SEM_VALUE_MAX}.

The user ID of the semaphore is set to the effective user ID of the process; the group ID of the semaphore is set to a system default group ID or to the effective group ID of the process. The permission bits of the semaphore are set to the value of the *mode* argument except those set in the file mode creation mask of the process. When bits in *mode* other than the file permission bits are specified, the effect is unspecified.

After the semaphore named *name* has been created by *sem_open*( ) with the O_CREAT flag, other processes can connect to the semaphore by calling *sem_open*( ) with the same value of *name*.

O_EXCL If O_EXCL and O_CREAT are set, *sem_open*( ) fails if the semaphore *name* exists. The check for the existence of the semaphore and the creation of the semaphore if it does not exist are atomic with respect to other processes executing *sem_open*( ) with O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the effect is undefined.

If flags other than O_CREAT and O_EXCL are specified in the *oflag* parameter, the effect is unspecified.

The *name* argument points to a string naming a semaphore object. It is unspecified whether the name appears in the file system and is visible to functions that take path names as arguments. The *name* argument conforms to the construction rules for a path name. If *name* begins with the slash character, then processes calling *sem_open*( ) with the same value of *name* shall refer to the same semaphore object, as long as that name has not been removed. If *name* does not begin with the slash character, the effect is implementation-defined. The interpretation of slash characters other than the leading slash character in *name* is implementation-defined.

If a process makes multiple successful calls to *sem_open*( ) with the same value for *name*, the same semaphore address is returned for each such successful call, provided that there have been

no calls to *sem_unlink*( ) for this semaphore.

References to copies of the semaphore produce undefined results.

RETURN VALUE

Upon successful completion, the *sem_open*( ) function shall return the address of the semaphore. Otherwise, it shall return a value of SEM_FAILED and set *errno* to indicate the error. The symbol SEM_FAILED is defined in the header **<semaphore.h>**. No successful return from *sem_open*( ) shall return the value SEM_FAILED.

ERRORS

If any of the following conditions occur, the *sem_open*( ) function shall return SEM_FAILED and set *errno* to the corresponding value:

[EACCES] The named semaphore exists and the permissions specified by *oflag* are denied, or the named semaphore does not exist and permission to create the named semaphore is denied.

[EEXIST] O_CREAT and O_EXCL are set and the named semaphore already exists.

[EINTR] The *sem_open*( ) operation was interrupted by a signal.

[EINVAL] The *sem_open*( ) operation is not supported for the given name, or O_CREAT was specified in *oflag* and *value* was greater than {SEM_VALUE_MAX}.

[EMFILE] Too many semaphore descriptors or file descriptors are currently in use by this process.

[ENAMETOOLONG]
The length of the *name* argument exceeds {PATH_MAX} or a path name component is longer than {NAME_MAX}.

[ENFILE] Too many semaphores are currently open in the system.

[ENOENT] O_CREAT is not set and the named semaphore does not exist.

[ENOSPC] There is insufficient space for the creation of the new named semaphore.

EXAMPLES

None.

APPLICATION USAGE

The *sem_open*( ) function is part of the Semaphores option and need not be available on all implementations.

RATIONALE

An earlier version of this volume of IEEE Std. 1003.1-200x required an error return value of –1 with the type **sem_t*** for the *sem_open*( ) function, which is not guaranteed to be portable across implementations. The revised text provides the symbolic error code SEM_FAILED to eliminate the type conflict.

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>**

39022 **CHANGE HISTORY**

39023 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39024 **Issue 6**

39025 The *sem_open*( ) function is marked as part of the Semaphores option. |

39026 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39027 implementation does not support the Semaphores option. |

39028 The *sem_timedwait*( ) function is added to the SEE ALSO section for alignment with
39029 IEEE Std. 1003.1d-1999.

39030 **NAME**

39031 sem_post — unlock a semaphore (**REALTIME**)

39032 **SYNOPSIS**

39033 SEM
```
#include <semaphore.h>

int sem_post(sem_t *sem);
```
39034

39035

39036 **DESCRIPTION**

39037 The *sem_post*( ) function unlocks the semaphore referenced by *sem* by performing a semaphore
39038 unlock operation on that semaphore.

39039 If the semaphore value resulting from this operation is positive, then no threads were blocked
39040 waiting for the semaphore to become unlocked; the semaphore value is simply incremented.

39041 If the value of the semaphore resulting from this operation is zero, then one of the threads
39042 blocked waiting for the semaphore shall be allowed to return successfully from its call to
39043 PS *sem_wait*( ). If the Process Scheduling option is supported, the thread to be unblocked shall be |
39044 chosen in a manner appropriate to the scheduling policies and parameters in effect for the
39045 blocked threads. In the case of the schedulers SCHED_FIFO and SCHED_RR, the highest
39046 priority waiting thread shall be unblocked, and if there is more than one highest priority thread
39047 blocked waiting for the semaphore, then the highest priority thread that has been waiting the
39048 longest shall be unblocked. If the Process Scheduling option is not defined, the choice of a thread
39049 to unblock is unspecified.

39050 SS If the Process Sporadic Server option is supported, and the scheduling policy is |
39051 SCHED_SPORADIC, the semantics are as per SCHED_FIFO above. |

39052 The *sem_post*( ) function shall be reentrant with respect to signals and may be invoked from a
39053 signal-catching function.

39054 **RETURN VALUE**

39055 If successful, the *sem_post*( ) function shall return zero; otherwise, the function shall return –1
39056 and set *errno* to indicate the error.

39057 **ERRORS**

39058 The *sem_post*( ) function shall fail if:

39059 [EINVAL] The *sem* argument does not refer to a valid semaphore. |

39060 **EXAMPLES**

39061 None.

39062 **APPLICATION USAGE**

39063 The *sem_post*( ) function is part of the Semaphores option and need not be available on all |
39064 implementations.

39065 **RATIONALE**

39066 None.

39067 **FUTURE DIRECTIONS**

39068 None.

39069 **SEE ALSO**

39070 *semctl*( ), *semget*( ), *semop*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_wait*( ), the Base Definitions |
39071 volume of IEEE Std. 1003.1-200x, **<semaphore.h>** |

39072 **CHANGE HISTORY**

39073 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39074 **Issue 6**

39075 The *sem_post*( ) function is marked as part of the Semaphores option. |

39076 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39077 implementation does not support the Semaphores option. |

39078 The *sem_timedwait*( ) function is added to the SEE ALSO section for alignment with
39079 IEEE Std. 1003.1d-1999.

39080 SCHED_SPORADIC is added to the list of scheduling policies for which the thread that is to be
39081 unblocked is specified for alignment with IEEE Std. 1003.1d-1999.

39082 **NAME**

39083 sem_timedwait — lock a semaphore (**REALTIME**)

39084 **SYNOPSIS**

39085 SEM TMO
```
#include <semaphore.h>
#include <time.h>

int sem_timedwait(sem_t *restrict sem,
    const struct timespec *restrict abs_timeout);
```

39089

39090 **DESCRIPTION**

39091 The *sem_timedwait*( ) function locks the semaphore referenced by *sem* as in the *sem_wait*( ) function. However, if the semaphore cannot be locked without waiting for another process or thread to unlock the semaphore by performing a *sem_post*( ) function, this wait shall be terminated when the specified timeout expires.

39095 The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call. If the Timers option is supported, the timeout is based on the CLOCK_REALTIME clock; if the Timers option is not supported, the timeout is based on the system clock as returned by the *time*( ) function.

39101 **RETURN VALUE**

39102 The *sem_timedwait*( ) function shall return zero if the calling process successfully performed the semaphore lock operation on the semaphore designated by *sem*. If the call was unsuccessful, the state of the semaphore shall be unchanged, and the function shall return a value of –1 and set *errno* to indicate the error.

39106 **ERRORS**

39107 The *sem_timedwait*( ) function shall fail if:

39108 [EINVAL] The *sem* argument does not refer to a valid semaphore.

39109 [EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1 000 million.

39112 [ETIMEDOUT] The semaphore could not be locked before the specified timeout expired.

39113 The *sem_timedwait*( ) function may fail if:

39114 [EDEADLK] A deadlock condition was detected.

39115 [EINTR] A signal interrupted this function.

39116 **EXAMPLES**

39117 None.

39118 **APPLICATION USAGE**

39119 Applications using these functions may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std. 1003.1-200x, Section 3.287, Priority Inversion.

39121 The *sem_timedwait*( ) function is part of the Semaphores and Timeouts options and need not be provided on all implementations.

39123 **RATIONALE**

39124 None.

39125 **FUTURE DIRECTIONS**

39126 None.

39127 **SEE ALSO**

39128 *sem_post*( ), *sem_trywait*( ), *sem_wait*( ), *semctl*( ), *semget*( ), *semop*( ), *time*( ), the Base Definitions |
39129 volume of IEEE Std. 1003.1-200x, **<semaphore.h>**, **<time.h>** |

39130 **CHANGE HISTORY**

39131 First released in Issue 6. Derived from IEEE Std. 1003.1d-1999.

39132 **NAME**

39133 sem_trywait, sem_wait — lock a semaphore (**REALTIME**)

39134 **SYNOPSIS**

39135 SEM
```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
```
39138

39139 **DESCRIPTION**

39140 The *sem_trywait*( ) function locks the semaphore referenced by *sem* only if the semaphore is
39141 currently not locked; that is, if the semaphore value is currently positive. Otherwise, it does not
39142 lock the semaphore.

39143 The *sem_wait*( ) function locks the semaphore referenced by *sem* by performing a semaphore lock
39144 operation on that semaphore. If the semaphore value is currently zero, then the calling thread
39145 shall not return from the call to *sem_wait*( ) until it either locks the semaphore or the call is
39146 interrupted by a signal.

39147 Upon successful return, the state of the semaphore shall be locked and shall remain locked until
39148 the *sem_post*( ) function is executed and returns successfully.

39149 The *sem_wait*( ) function is interruptible by the delivery of a signal.

39150 **RETURN VALUE**

39151 The *sem_trywait*( ) and *sem_wait*( ) functions shall return zero if the calling process successfully
39152 performed the semaphore lock operation on the semaphore designated by *sem*. If the call was
39153 unsuccessful, the state of the semaphore shall be unchanged, and the function shall return a
39154 value of –1 and set *errno* to indicate the error.

39155 **ERRORS**

39156 The *sem_trywait*( ) and *sem_wait*( ) functions shall fail if:

39157 [EAGAIN] The semaphore was already locked, so it cannot be immediately locked by the |
39158 *sem_trywait*( ) operation (*sem_trywait*( ) only).

39159 [EINVAL] The *sem* argument does not refer to a valid semaphore. |

39160 The *sem_trywait*( ) and *sem_wait*( ) functions may fail if:

39161 [EDEADLK] A deadlock condition was detected. |

39162 [EINTR] A signal interrupted this function. |

39163 **EXAMPLES**

39164 None.

39165 **APPLICATION USAGE**

39166 Applications using these functions may be subject to priority inversion, as discussed in the Base |
39167 Definitions volume of IEEE Std. 1003.1-200x, Section 3.287, Priority Inversion. |

39168 The *sem_trywait*( ) and *sem_wait*( ) functions are part of the Semaphores option and need not be |
39169 provided on all implementations.

39170 **RATIONALE**

39171 None.

FUTURE DIRECTIONS

None.

SEE ALSO

semctl( ), *semget*( ), *semop*( ), *sem_post*( ), *sem_timedwait*( ), 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_trywait*( ) and *sem_wait*( ) functions are 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.

39185 **NAME**

39186 sem_unlink — remove a named semaphore (**REALTIME**)

39187 **SYNOPSIS**

39188 SEM
```
#include <semaphore.h>
```

39189
```
int sem_unlink(const char *name);
```

39190

39191 **DESCRIPTION**

39192 The *sem_unlink*( ) function removes the semaphore named by the string *name*. If the semaphore
39193 named by *name* is currently referenced by other processes, then *sem_unlink*( ) has no effect on the
39194 state of the semaphore. If one or more processes have the semaphore open when *sem_unlink*( ) is
39195 called, destruction of the semaphore is postponed until all references to the semaphore have
39196 been destroyed by calls to *sem_close*( ), *_exit*( ), or *exec*. Calls to *sem_open*( ) to recreate or
39197 reconnect to the semaphore refer to a new semaphore after *sem_unlink*( ) is called. The
39198 *sem_unlink*( ) call does not block until all references have been destroyed; it shall return
39199 immediately.

39200 **RETURN VALUE**

39201 Upon successful completion, the *sem_unlink*( ) function shall return a value of 0. Otherwise, the
39202 semaphore shall not be changed and the function shall return a value of −1 and set *errno* to
39203 indicate the error.

39204 **ERRORS**

39205 The *sem_unlink*( ) function shall fail if:

39206 [EACCES] Permission is denied to unlink the named semaphore. |

39207 [ENAMETOOLONG] |
39208 The length of the *name* argument exceeds {PATH_MAX} or a path name |
39209 component is longer than {NAME_MAX}. |

39210 [ENOENT] The named semaphore does not exist. |

39211 **EXAMPLES**

39212 None.

39213 **APPLICATION USAGE**

39214 The *sem_unlink*( ) function is part of the Semaphores option and need not be available on all |
39215 implementations.

39216 **RATIONALE**

39217 None.

39218 **FUTURE DIRECTIONS**

39219 None.

39220 **SEE ALSO**

39221 *semctl*( ), *semget*( ), *semop*( ), *sem_close*( ), *sem_open*( ), the Base Definitions volume of |
39222 IEEE Std. 1003.1-200x, **<semaphore.h>** |

39223 **CHANGE HISTORY**

39224 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39225 **Issue 6**

39226 The *sem_unlink*( ) function is marked as part of the Semaphores option. |

39227 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39228 implementation does not support the Semaphores option. |

39229 **NAME**

39230 sem_wait — lock a semaphore (**REALTIME**)

39231 **SYNOPSIS**

39232 SEM
```
#include <semaphore.h>
```

39233
```
int sem_wait(sem_t *sem);
```

39234

39235 **DESCRIPTION**

39236 Refer to *sem_trywait*( ).

NAME

semctl — XSI semaphore control operations

SYNOPSIS

XSI
```
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
```

DESCRIPTION

The *semctl*( ) function operates on XSI semaphores (see the Base Definitions volume of IEEE Std. 1003.1-200x, Section 4.13, Semaphore). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 543).

The *semctl*( ) function provides a variety of semaphore control operations as specified by *cmd*. The fourth argument is optional and depends upon the operation requested. If required, it is of type **union semun**, which the application shall explicitly declare:

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short  *array;
} arg;
```

The following semaphore control operations as specified by *cmd* are executed with respect to the semaphore specified by *semid* and *semnum*. The level of permission required for each operation is shown with each command; see Section 2.7 (on page 541). The symbolic names for the values of *cmd* are defined by the **<sys/sem.h>** header:

GETVAL — Return the value of *semval*; see **<sys/sem.h>**. Requires read permission.

SETVAL — Set the value of *semval* to *arg.val*, where *arg* is the value of the fourth argument to *semctl*( ). When this command is successfully executed, the *semadj* value corresponding to the specified semaphore in all processes is cleared. Requires alter permission; see Section 2.7 (on page 541).

GETPID — Return the value of *sempid*. Requires read permission.

GETNCNT — Return the value of *semncnt*. Requires read permission.

GETZCNT — Return the value of *semzcnt*. Requires read permission.

The following values of *cmd* operate on each *semval* in the set of semaphores:

GETALL — Return the value of *semval* for each semaphore in the semaphore set and place into the array pointed to by *arg.array*, where *arg* is the fourth argument to *semctl*( ). Requires read permission.

SETALL — Set the value of *semval* for each semaphore in the semaphore set according to the array pointed to by *arg.array*, where *arg* is the fourth argument to *semctl*( ). When this command is successfully executed, the *semadj* values corresponding to each specified semaphore in all processes are cleared. Requires alter permission.

The following values of *cmd* are also available:

IPC_STAT — Place the current value of each member of the **semid_ds** data structure associated with *semid* into the structure pointed to by *arg.buf*, where *arg* is the fourth argument to *semctl*( ). The contents of this structure are defined in

39281 **<sys/sem.h>**. Requires read permission.

39282 IPC_SET Set the value of the following members of the **semid_ds** data structure
39283 associated with *semid* to the corresponding value found in the structure
39284 pointed to by *arg.buf*, where *arg* is the fourth argument to *semctl*( ):

39285 `sem_perm.uid`
39286 `sem_perm.gid`
39287 `sem_perm.mode`

39288 The mode bits specified in Section 2.7.1 (on page 541) are copied into the
39289 corresponding bits of the *sem_perm.mode* associated with *semid*. The stored
39290 values of any other bits are unspecified.

39291 This command can only be executed by a process that has an effective user ID
39292 equal to either that of a process with appropriate privileges or to the value of
39293 *sem_perm.cuid* or *sem_perm.uid* in the **semid_ds** data structure associated with
39294 *semid*.

39295 IPC_RMID Remove the semaphore identifier specified by *semid* from the system and
39296 destroy the set of semaphores and **semid_ds** data structure associated with it.
39297 This command can only be executed by a process that has an effective user ID
39298 equal to either that of a process with appropriate privileges or to the value of
39299 *sem_perm.cuid* or *sem_perm.uid* in the **semid_ds** data structure associated with
39300 *semid*.

39301 **RETURN VALUE**

39302 If successful, the value returned by *semctl*( ) depends on *cmd* as follows:

39303 GETVAL The value of *semval*.

39304 GETPID The value of *sempid*.

39305 GETNCNT The value of *semncnt*.

39306 GETZCNT The value of *semzcnt*.

39307 All others 0.

39308 Otherwise, *semctl*( ) shall return –1 and set *errno* to indicate the error.

39309 **ERRORS**

39310 The *semctl*( ) function shall fail if:

39311 [EACCES] Operation permission is denied to the calling process; see Section 2.7 (on page |
39312 541).

39313 [EINVAL] The value of *semid* is not a valid semaphore identifier, or the value of *semnum* |
39314 is less than 0 or greater than or equal to *sem_nsems*, or the value of *cmd* is not a
39315 valid command.

39316 [EPERM] The argument *cmd* is equal to IPC_RMID or IPC_SET and the effective user ID |
39317 of the calling process is not equal to that of a process with appropriate
39318 privileges and it is not equal to the value of *sem_perm.cuid* or *sem_perm.uid* in
39319 the data structure associated with *semid*.

39320 [ERANGE] The argument *cmd* is equal to SETVAL or SETALL and the value to which |
39321 *semval* is to be set is greater than the system-imposed maximum.

EXAMPLES

None.

APPLICATION USAGE

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 backward compatibility.

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 (on page 541) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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>**, Section 2.7 (on page 541)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 4

The function is no longer marked as OPTIONAL FUNCTIONALITY.

The **<sys/types.h>** and **<sys/ipc.h>** headers are removed from the SYNOPSIS section.

The last argument is now defined by an ellipsis symbol. In previous issues it was defined as a union of the various types required by settings of *cmd*. These are now defined individually in each description of permitted *cmd* settings. The text of the description of SETALL in the DESCRIPTION now refers to the fourth argument instead of *arg.buf*.

In the DESCRIPTION the type of the array is specified in the descriptions of GETALL and SETALL.

The [ENOSYS] error is removed from the ERRORS section.

A FUTURE DIRECTIONS section is added warning application developers about migration to IEEE 1003.4 interfaces for interprocess communication.

Issue 4, Version 2

The fourth argument to *semctl*( ), formerly specified in the APPLICATION USAGE section, is moved to the DESCRIPTION, and references to its elements are made more precise.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to the APPLICATION USAGE section.

NAME

semget — get set of XSI semaphores

SYNOPSIS

XSI
```
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

DESCRIPTION

The *semget*( ) function operates on XSI semaphores (see the Base Definitions volume of IEEE Std. 1003.1-200x, Section 4.13, Semaphore). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 543).

The *semget*( ) function shall return the semaphore identifier associated with *key*.

A semaphore identifier with its associated **semid_ds** data structure and its associated set of *nsems* semaphores (see **<sys/sem.h>**) is 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 semaphore identifier associated with it and (*semflg* &IPC_CREAT) is non-zero.

Upon creation, the **semid_ds** data structure associated with the new semaphore identifier is initialized as follows:

- In the operation permissions structure *sem_perm.cuid*, *sem_perm.uid*, *sem_perm.cgid*, and *sem_perm.gid* are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of *sem_perm.mode* are set equal to the low-order 9 bits of *semflg*.
- The variable *sem_nsems* is set equal to the value of *nsems*.
- The variable *sem_otime* is set equal to 0 and *sem_ctime* is set equal to the current time.
- The data structure associated with each semaphore in the set is not initialized. The *semctl*( ) function with the command SETVAL or SETALL can be used to initialize each semaphore.

RETURN VALUE

Upon successful completion, *semget*( ) shall return a non-negative integer, namely a semaphore identifier; otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *semget*( ) function shall fail if:

[EACCES] A semaphore identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *semflg* would not be granted; see Section 2.7 (on page 541).

[EEXIST] A semaphore identifier exists for the argument *key* but ((*semflg* &IPC_CREAT) &&(*semflg* &IPC_EXCL)) is non-zero.

[EINVAL] The value of *nsems* is either less than or equal to 0 or greater than the system-imposed limit, or a semaphore identifier exists for the argument *key*, but the number of semaphores in the set associated with it is less than *nsems* and *nsems* is not equal to 0.

[ENOENT] A semaphore identifier does not exist for the argument *key* and (*semflg* &IPC_CREAT) is equal to 0.

[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. |

EXAMPLES

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 queueing 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.

```
#include <sys/types.h>
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <pwd.h>
#include <fcntl.h>
#include <limits.h>
...
key_t semkey;
int semid, pfd, fv;
struct sembuf sbuf;
char *lgn;
char filename[PATH_MAX+1];
struct stat outstat;
struct passwd *pw;
...
/* Get unique key for semaphore. */
if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
    perror("IPC error: ftok"); exit(1);
}

/* Get semaphore ID associated with this key. */
if ((semid = semget(semkey, 0, 0)) == -1) {

    /* Semaphore does not exist - Create. */
    if ((semid = semget(semkey, 1, IPC_CREAT | IPC_EXCL | S_IRUSR |
        S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)) != -1)
    {
        /* Initialize the semaphore. */
        sbuf.sem_num = 0;
        sbuf.sem_op = 2;  /* This is the number of runs without queuing. */
        sbuf.sem_flg = 0;
```

```
            if (semop(semid, &sbuf, 1) == -1) {
                perror("IPC error: semop"); exit(1);
            }
        }
        else if (errno == EEXIST) {
            if ((semid = semget(semkey, 0, 0)) == -1) {
                perror("IPC error 1: semget"); exit(1);
            }
        }
        else {
            perror("IPC error 2: semget"); exit(1);
        }
    }
    ...
```

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 (on page 541) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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>**, Section 2.7 (on page 541).

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 4

The function is no longer marked as OPTIONAL FUNCTIONALITY.

The **<sys/types.h>** and **<sys/ipc.h>** headers are removed from the SYNOPSIS section.

The [ENOSYS] error is removed from the ERRORS section.

A FUTURE DIRECTIONS section is added warning application developers about migration to IEEE 1003.4 interfaces for interprocess communication.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

NAME

semop — XSI semaphore operations

SYNOPSIS

XSI
```
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);
```

DESCRIPTION

The *semop*( ) function operates on XSI semaphores (see the Base Definitions volume of IEEE Std. 1003.1-200x, Section 4.13, Semaphore). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 543).

The *semop*( ) function is used to perform atomically a user-defined array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by the argument *semid*.

The argument *sops* is a pointer to a user-defined array of semaphore operation structures. The implementation shall not modify elements of this array unless the application uses implementation-defined extensions.

The argument *nsops* is the number of such structures in the array.

Each structure, **sembuf**, includes the following members:

Member Type	Member Name	Description
short	*sem_num*	Semaphore number.
short	*sem_op*	Semaphore operation.
short	*sem_flg*	Operation flags.

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by *semid* and *sem_num*.

The variable *sem_op* specifies one of three semaphore operations:

1. If *sem_op* is a negative integer and the calling process has alter permission, one of the following shall occur:

   - If *semval*(see **<sys/sem.h>**) is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from *semval*. Also, if (*sem_flg* &SEM_UNDO) is non-zero, the absolute value of *sem_op* is added to the calling process' *semadj* value for the specified semaphore.

   - If *semval* is less than the absolute value of *sem_op* and (*sem_flg* &IPC_NOWAIT) is non-zero, *semop*( ) shall return immediately.

   - If *semval* is less than the absolute value of *sem_op* and (*sem_flg* &IPC_NOWAIT) is 0, *semop*( ) shall increment the *semncnt* associated with the specified semaphore and suspend execution of the calling thread until one of the following conditions occurs:

     — The value of *semval* becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of *semncnt* associated with the specified semaphore is decremented, the absolute value of *sem_op* is subtracted from *semval* and, if (*sem_flg* &SEM_UNDO) is non-zero, the absolute value of *sem_op* is added to the calling process' *semadj* value for the specified semaphore.

— The *semid* for which the calling thread is awaiting action is removed from the system. When this occurs, *errno* shall be set equal to [EIDRM] and –1 shall be returned.

— The calling thread receives a signal that is to be caught. When this occurs, the value of *semncnt* associated with the specified semaphore is decremented, and the calling thread resumes execution in the manner prescribed in *sigaction*( ).

2. If *sem_op* is a positive integer and the calling process has alter permission, the value of *sem_op* is added to *semval* and, if (*sem_flg* &SEM_UNDO) is non-zero, the value of *sem_op* is subtracted from the calling process' *semadj* value for the specified semaphore.

3. If *sem_op* is 0 and the calling process has read permission, one of the following shall occur:

   - If *semval* is 0, *semop*( ) shall return immediately.
   - If *semval* is non-zero and (*sem_flg* &IPC_NOWAIT) is non-zero, *semop*( ) shall return immediately.
   - If *semval* is non-zero and (*sem_flg* &IPC_NOWAIT) is 0, *semop*( ) will increment the *semzcnt* associated with the specified semaphore and suspends execution of the calling thread until one of the following occurs:

     — The value of *semval* becomes 0, at which time the value of *semzcnt* associated with the specified semaphore is decremented.

     — The *semid* for which the calling thread is awaiting action is removed from the system. When this occurs, *errno* shall be set equal to [EIDRM] and –1 shall be returned.

     — The calling thread receives a signal that is to be caught. When this occurs, the value of *semzcnt* associated with the specified semaphore is decremented, and the calling thread resumes execution in the manner prescribed in *sigaction*( ).

Upon successful completion, the value of *sempid* for each semaphore specified in the array pointed to by *sops* shall be set equal to the process ID of the calling process.

RETURN VALUE

Upon successful completion, *semop*( ) shall return 0; otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *semop*( ) function shall fail if:

[E2BIG]	The value of *nsops* is greater than the system-imposed maximum.
[EACCES]	Operation permission is denied to the calling process; see Section 2.7 (on page 541).
[EAGAIN]	The operation would result in suspension of the calling process but (*sem_flg* &IPC_NOWAIT) is non-zero.
[EFBIG]	The value of *sem_num* is less than 0 or greater than or equal to the number of semaphores in the set associated with *semid*.
[EIDRM]	The semaphore identifier *semid* is removed from the system.
[EINTR]	The *semop*( ) function was interrupted by a signal.
[EINVAL]	The value of *semid* 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.

39575 [ENOSPC] The limit on the number of individual processes requesting a SEM_UNDO |
39576 would be exceeded.

39577 [ERANGE] An operation would cause a *semval* to overflow the system-imposed limit, or |
39578 an operation would cause a *semadj* value to overflow the system-imposed
39579 limit.

39580 **EXAMPLES**

39581 **Setting Values in Semaphores**

39582 The following example sets the values of the two semaphores associated with the *semid*
39583 identifier to the values contained in the *sb* array.

```
39584 #include <sys/sem.h>
39585 ...
39586 int semid;
39587 struct sembuf sb[2];
39588 int nsops = 2;
39589 int result;

39590 /* Adjust value of semaphore in the semaphore array semid. */
39591 sb[0].sem_num = 0;
39592 sb[0].sem_op = -1;
39593 sb[0].sem_flg = SEM_UNDO | IPC_NOWAIT;
39594 sb[1].sem_num = 1;
39595 sb[1].sem_op =  1;
39596 sb[1].sem_flg = 0;

39597 result = semop(semid, sb, nsops);
```

39598 **Creating a Semaphore Identifier**

39599 The following example gets a unique semaphore key using the *ftok*( ) function, then gets a
39600 semaphore ID associated with that key using the *semget*( ) function (the first call also tests to
39601 make sure the semaphore exists). If the semaphore does not exist, the program creates it, as
39602 shown by the second call to *semget*( ). In creating the semaphore for the queueing process, the
39603 program attempts to create one semaphore with read/write permission for all. It also uses the
39604 IPC_EXCL flag, which forces *semget*( ) to fail if the semaphore already exists.

39605 After creating the semaphore, the program uses a call to *semop*( ) to initialize it to the values in
39606 the *sbuf* array. The number of processes that can execute concurrently without queuing is
39607 initially set to 2. The final call to *semget*( ) creates a semaphore identifier that can be used later in
39608 the program.

39609 The final call to *semop*( ) acquires the semaphore and waits until it is free; the SEM_UNDO
39610 option releases the semaphore when the process exits, waiting until there are less than two
39611 processes running concurrently.

```
39612 #include <sys/types.h>
39613 #include <stdio.h>
39614 #include <sys/ipc.h>
39615 #include <sys/sem.h>
39616 #include <sys/stat.h>
39617 #include <errno.h>
39618 #include <unistd.h>
39619 #include <stdlib.h>
```

```
#include <pwd.h>
#include <fcntl.h>
#include <limits.h>
...
key_t semkey;
int semid, pfd, fv;
struct sembuf sbuf;
char *lgn;
char filename[PATH_MAX+1];
struct stat outstat;
struct passwd *pw;
...
/* Get unique key for semaphore. */
if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
    perror("IPC error: ftok"); exit(1);
}
/* Get semaphore ID associated with this key. */
if ((semid = semget(semkey, 0, 0)) == -1) {
    /* Semaphore does not exist - Create. */
    if ((semid = semget(semkey, 1, IPC_CREAT | IPC_EXCL | S_IRUSR |
        S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)) != -1)
    {
        /* Initialize the semaphore. */
        sbuf.sem_num = 0;
        sbuf.sem_op = 2;  /* This is the number of runs without queuing. */
        sbuf.sem_flg = 0;
        if (semop(semid, &sbuf, 1) == -1) {
            perror("IPC error: semop"); exit(1);
        }
    }
    else if (errno == EEXIST) {
        if ((semid = semget(semkey, 0, 0)) == -1) {
            perror("IPC error 1: semget"); exit(1);
        }
    }
    else {
        perror("IPC error 2: semget"); exit(1);
    }
}
...
sbuf.sem_num = 0;
sbuf.sem_op = -1;
sbuf.sem_flg = SEM_UNDO;
if (semop(semid, &sbuf, 1) == -1) {
    perror("IPC Error: semop"); exit(1);
}
```

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 (on page 541) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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>**, Section 2.7 (on page 541)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 4

The function is no longer marked as OPTIONAL FUNCTIONALITY.

The **<sys/types.h>** and **<sys/ipc.h>** headers are removed from the SYNOPSIS section.

The type of *nsops* is changed to **size_t**.

The DESCRIPTION is updated to indicate that an implementation does not modify the elements of *sops* unless the application uses implementation-defined extensions.

The [ENOSYS] error is removed from the ERRORS section.

A FUTURE DIRECTIONS section is added warning application developers about migration to IEEE 1003.4 interfaces for interprocess communication.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

39693 **NAME**

39694 send — send a message on a socket

39695 **SYNOPSIS**

39696 `#include <sys/socket.h>`

39697 `ssize_t send(int socket, const void *buffer, size_t length, int flags);`

39698 **DESCRIPTION**

39699 The *send*( ) functions takes the following arguments:

39700 *socket* Specifies the socket file descriptor.

39701 *buffer* Points to the buffer containing the message to send.

39702 *length* Specifies the length of the message in bytes.

39703 *flags* Specifies the type of message transmission. Values of this argument are
39704 formed by logically OR'ing zero or more of the following flags:

39705 MSG_EOR Terminates a record (if supported by the protocol).

39706 MSG_OOB Sends out-of-band data on sockets that support out-of-band
39707 communications. The significance and semantics of out-of-
39708 band data are protocol-specific.

39709 The *send*( ) function initiates transmission of a message from the specified socket to its peer. The
39710 *send*( ) function sends a message only when the socket is connected (including when the peer of a
39711 connectionless socket has been set via *connect*( )).

39712 The length of the message to be sent is specified by the *length* argument. If the message is too
39713 long to pass through the underlying protocol, *send*( ) shall fail and no data shall be transmitted.

39714 Successful completion of a call to *send*( ) does not guarantee delivery of the message. A return
39715 value of –1 indicates only locally-detected errors.

39716 If space is not available at the sending socket to hold the message to be transmitted, and the
39717 socket file descriptor does not have O_NONBLOCK set, *send*( ) shall block until space is
39718 available. If space is not available at the sending socket to hold the message to be transmitted,
39719 and the socket file descriptor does have O_NONBLOCK set, *send*( ) shall fail. The *select*( ) and
39720 *poll*( ) functions can be used to determine when it is possible to send more data.

39721 The socket in use may require the process to have appropriate privileges to use the *send*( )
39722 function.

39723 **RETURN VALUE**

39724 Upon successful completion, *send*( ) shall return the number of bytes sent. Otherwise, –1 shall be
39725 returned and *errno* set to indicate the error.

39726 **ERRORS**

39727 The *send*( ) function shall fail if:

39728 [EAGAIN] or [EWOULDBLOCK]
39729 The socket's file descriptor is marked O_NONBLOCK and the requested
39730 operation would block.

39731 [EBADF] The *socket* argument is not a valid file descriptor.

39732 [ECONNRESET] A connection was forcibly closed by a peer.

39733 [EDESTADDRREQ]
39734 The socket is not connection-mode and no peer address is set. |

[EINTR] A signal interrupted *send*( ) before any data was transmitted.

[EMSGSIZE] The message is too large be sent all at once, as the socket requires.

[ENOTCONN] The socket is not connected or otherwise has not had the peer pre-specified.

[ENOTSOCK] The *socket* argument does not refer to a socket.

[EOPNOTSUPP] The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.

[EPIPE] The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling thread. |

The *send*( ) function may fail if:

[EACCES] The calling process does not have the appropriate privileges.

[EIO] An I/O error occurred while reading from or writing to the file system.

[ENETDOWN] The local network interface used to reach the destination is down. |

[ENETUNREACH]
No route to the network is present.

[ENOBUFS] Insufficient resources were available in the system to perform the operation. |

EXAMPLES

None.

APPLICATION USAGE

The *send*( ) function is identical to *sendto*( ) with a null pointer *dest_len* argument, and to *write*( ) if no flags are used.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

connect( ), *getsockopt*( ), *poll*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *select*( ), *sendmsg*( ), *sendto*( ), *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. |

39766 **NAME**

39767 sendmsg — send a message on a socket using a message structure

39768 **SYNOPSIS**

39769 `#include <sys/socket.h>`

39770 `ssize_t sendmsg(int socket, const struct msghdr *message, int flags);`

39771 **DESCRIPTION**

39772 The *sendmsg*( ) function sends a message through a connection-mode or connectionless-mode
39773 socket. If the socket is connectionless-mode, the message shall be sent to the address specified by
39774 **msghdr**. If the socket is connection-mode, the destination address in **msghdr** is ignored.

39775 The *sendmsg*( ) function takes the following arguments:

39776 *socket* Specifies the socket file descriptor.

39777 *message* Points to a **msghdr** structure, containing both the destination address and the
39778 buffers for the outgoing message. The length and format of the address
39779 depend on the address family of the socket. The *msg_flags* member is ignored.

39780 *flags* Specifies the type of message transmission. The application may specify 0 or
39781 the following flag:

39782 MSG_EOR Terminates a record (if supported by the protocol).

39783 MSG_OOB Sends out-of-band data on sockets that support out-of-
39784 bound data. The significance and semantics of out-of-band
39785 data are protocol-specific.

39786 The *msg_iov* and *msg_iovlen* fields of *message* specify zero or more buffers containing the data to
39787 be sent. *msg_iov* points to an array of **iovec** structures; *msg_iovlen* shall be set to the dimension of
39788 this array. In each **iovec** structure, the *iov_base* field specifies a storage area and the *iov_len* field
39789 gives its size in bytes. Some of these sizes can be zero. The data from each storage area indicated
39790 by *msg_iov* is sent in turn.

39791 Successful completion of a call to *sendmsg*( ) does not guarantee delivery of the message. A
39792 return value of –1 indicates only locally-detected errors.

39793 If space is not available at the sending socket to hold the message to be transmitted and the
39794 socket file descriptor does not have O_NONBLOCK set, *sendmsg*( ) function blocks until space is
39795 available. If space is not available at the sending socket to hold the message to be transmitted
39796 and the socket file descriptor does have O_NONBLOCK set, *sendmsg*( ) function shall fail.

39797 If the socket protocol supports broadcast and the specified address is a broadcast address for the
39798 socket protocol, *sendmsg*( ) shall fail if the SO_BROADCAST option is not set for the socket.

39799 The socket in use may require the process to have appropriate privileges to use the *sendmsg*( )
39800 function.

39801 **RETURN VALUE**

39802 Upon successful completion, *sendmsg*( ) shall return the number of bytes sent. Otherwise, –1
39803 shall be returned and *errno* set to indicate the error.

39804 **ERRORS**

39805 The *sendmsg*( ) function shall fail if:

39806 [EAGAIN] or [EWOULDBLOCK]
39807 The socket's file descriptor is marked O_NONBLOCK and the requested
39808 operation would block.

39809 [EAFNOSUPPORT]
39810 Addresses in the specified address family cannot be used with this socket.

39811 [EBADF] The *socket* argument is not a valid file descriptor.

39812 [ECONNRESET] A connection was forcibly closed by a peer.

39813 [EINTR] A signal interrupted *sendmsg*( ) before any data was transmitted.

39814 [EINVAL] The sum of the *iov_len* values overflows an **ssize_t**.

39815 [EMSGSIZE] The message is too large to be sent all at once (as the socket requires), or the
39816 *msg_iovlen* member of the **msghdr** structure pointed to by *message* is less than
39817 or equal to 0 or is greater than {IOV_MAX}.

39818 [ENOTCONN] The socket is connection-mode but is not connected.

39819 [ENOTSOCK] The *socket* argument does not refer a socket.

39820 [EOPNOTSUPP] The *socket* argument is associated with a socket that does not support one or
39821 more of the values set in *flags*.

39822 [EPIPE] The socket is shut down for writing, or the socket is connection-mode and is
39823 no longer connected. In the latter case, and if the socket is of type
39824 SOCK_STREAM, the SIGPIPE signal is generated to the calling thread.

39825 If the address family of the socket is AF_UNIX, then *sendmsg*( ) shall fail if:

39826 [EIO] An I/O error occurred while reading from or writing to the file system.

39827 [ELOOP] A loop exists in symbolic links encountered during resolution of the path
39828 name in the socket address.

39829 [ENAMETOOLONG]
39830 A component of a path name exceeded {NAME_MAX} characters, or an entire
39831 path name exceeded {PATH_MAX} characters.

39832 [ENOENT] A component of the path name does not name an existing file or the path
39833 name is an empty string.

39834 [ENOTDIR] A component of the path prefix of the path name in the socket address is not a
39835 directory.

39836 The *sendmsg*( ) function may fail if:

39837 [EACCES] Search permission is denied for a component of the path prefix; or write
39838 access to the named socket is denied.

39839 [EDESTADDRREQ]
39840 The socket is not connection-mode and does not have its peer address set, and
39841 no destination address was specified.

39842 [EHOSTUNREACH]
39843 The destination host cannot be reached (probably because the host is down or
39844 a remote router cannot reach it).

39845 [EIO] An I/O error occurred while reading from or writing to the file system.

39846 [EISCONN] A destination address was specified and the socket is already connected.

39847 [ENETDOWN] The local network interface used to reach the destination is down.

39848 [ENETUNREACH]
39849 No route to the network is present.

39850 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

39851 [ENOMEM] Insufficient memory was available to fulfill the request.

39852 If the address family of the socket is AF_UNIX, then *sendmsg*( ) may fail if:

39853 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
39854 resolution of the path name in the socket address.

39855 [ENAMETOOLONG]
39856 Path name resolution of a symbolic link produced an intermediate result
39857 whose length exceeds {PATH_MAX}.

39858 **EXAMPLES**

39859 Done.

39860 **APPLICATION USAGE**

39861 The *select*( ) and *poll*( ) functions can be used to determine when it is possible to send more data.

39862 **RATIONALE**

39863 None.

39864 **FUTURE DIRECTIONS**

39865 None.

39866 **SEE ALSO**

39867 *getsockopt*( ), *poll*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *select*( ), *send*( ), *sendto*( ), *setsockopt*( ),
39868 *shutdown*( ), *socket*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/socket.h>**

39869 **CHANGE HISTORY**

39870 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

39871 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
39872 [ELOOP] error condition is added.

39873 **NAME**

39874 sendto — send a message on a socket

39875 **SYNOPSIS**

39876 `#include <sys/socket.h>`

39877 `ssize_t sendto(int` *`socket`*`, const void *`*`message`*`, size_t` *`length`*`,`
39878 `int` *`flags`*`, const struct sockaddr *`*`dest_addr`*`,`
39879 `socklen_t` *`dest_len`*`);`

39880 **DESCRIPTION**

39881 The *sendto*( ) function sends a message through a connection-mode or connectionless-mode
39882 socket. If the socket is connectionless-mode, the message shall be sent to the address specified by
39883 *dest_addr*. If the socket is connection-mode, *dest_addr* is ignored.

39884 The *sendto*( ) function takes the following arguments:

39885 *socket* Specifies the socket file descriptor.

39886 *message* Points to a buffer containing the message to be sent.

39887 *length* Specifies the size of the message in bytes.

39888 *flags* Specifies the type of message transmission. Values of this argument are
39889 formed by logically OR'ing zero or more of the following flags:

39890 MSG_EOR Terminates a record (if supported by the protocol).

39891 MSG_OOB Sends out-of-band data on sockets that support out-of-band
39892 data. The significance and semantics of out-of-band data are
39893 protocol-specific.

39894 *dest_addr* Points to a **sockaddr** structure containing the destination address. The length
39895 and format of the address depend on the address family of the socket.

39896 *dest_len* Specifies the length of the **sockaddr** structure pointed to by the *dest_addr*
39897 argument.

39898 If the socket protocol supports broadcast and the specified address is a broadcast address for the
39899 socket protocol, *sendto*( ) shall fail if the SO_BROADCAST option is not set for the socket.

39900 The *dest_addr* argument specifies the address of the target. The *length* argument specifies the
39901 length of the message.

39902 Successful completion of a call to *sendto*( ) does not guarantee delivery of the message. A return
39903 value of –1 indicates only locally-detected errors.

39904 If space is not available at the sending socket to hold the message to be transmitted and the
39905 socket file descriptor does not have O_NONBLOCK set, *sendto*( ) blocks until space is available.
39906 If space is not available at the sending socket to hold the message to be transmitted and the
39907 socket file descriptor does have O_NONBLOCK set, *sendto*( ) shall fail.

39908 The socket in use may require the process to have appropriate privileges to use the *sendto*( )
39909 function.

39910 **RETURN VALUE**

39911 Upon successful completion, *sendto*( ) shall return the number of bytes sent. Otherwise, –1 shall
39912 be returned and *errno* set to indicate the error.

39913 **ERRORS**

39914 The *sendto*( ) function shall fail if:

39915 [EAFNOSUPPORT]
39916 Addresses in the specified address family cannot be used with this socket.

39917 [EAGAIN] or [EWOULDBLOCK]
39918 The socket's file descriptor is marked O_NONBLOCK and the requested
39919 operation would block.

39920 [EBADF] The *socket* argument is not a valid file descriptor.

39921 [ECONNRESET] A connection was forcibly closed by a peer.

39922 [EINTR] A signal interrupted *sendto*( ) before any data was transmitted.

39923 [EMSGSIZE] The message is too large to be sent all at once, as the socket requires.

39924 [ENOTCONN] The socket is connection-mode but is not connected.

39925 [ENOTSOCK] The *socket* argument does not refer to a socket.

39926 [EOPNOTSUPP] The *socket* argument is associated with a socket that does not support one or
39927 more of the values set in *flags*.

39928 [EPIPE] The socket is shut down for writing, or the socket is connection-mode and is
39929 no longer connected. In the latter case, and if the socket is of type
39930 SOCK_STREAM, the SIGPIPE signal is generated to the calling thread.

39931 If the address family of the socket is AF_UNIX, then *sendto*( ) shall fail if:

39932 [EIO] An I/O error occurred while reading from or writing to the file system.

39933 [ELOOP] A loop exists in symbolic links encountered during resolution of the path
39934 name in the socket address.

39935 [ENAMETOOLONG]
39936 A component of a path name exceeded {NAME_MAX} characters, or an entire
39937 path name exceeded {PATH_MAX} characters.

39938 [ENOENT] A component of the path name does not name an existing file or the path
39939 name is an empty string.

39940 [ENOTDIR] A component of the path prefix of the path name in the socket address is not a
39941 directory.

39942 The *sendto*( ) function may fail if:

39943 [EACCES] Search permission is denied for a component of the path prefix; or write
39944 access to the named socket is denied.

39945 [EDESTADDRREQ]
39946 The socket is not connection-mode and does not have its peer address set, and
39947 no destination address was specified.

39948 [EHOSTUNREACH]
39949 The destination host cannot be reached (probably because the host is down or
39950 a remote router cannot reach it).

39951 [EINVAL] The *dest_len* argument is not a valid length for the address family.

39952 [EIO] An I/O error occurred while reading from or writing to the file system.

[EISCONN] A destination address was specified and the socket is already connected. This error may or may not be returned for connection mode sockets.

[ENETDOWN] The local network interface used to reach the destination is down.

[ENETUNREACH] No route to the network is present.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENOMEM] Insufficient memory was available to fulfill the request.

If the address family of the socket is AF_UNIX, then *sendto*( ) may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the path name in the socket address.

[ENAMETOOLONG] Path name resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

EXAMPLES

None.

APPLICATION USAGE

The *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.

39981 **NAME**

39982 setbuf — assign buffering to a stream

39983 **SYNOPSIS**

39984 `#include <stdio.h>`

39985 `void setbuf(FILE *restrict stream, char *restrict buf);`

39986 **DESCRIPTION**

39987 CX The functionality described on this reference page is aligned with the ISO C standard. Any
39988 conflict between the requirements described here and the ISO C standard is unintentional. This
39989 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

39990 Except that it returns no value, the function call:

39991 `setbuf(stream, buf)`

39992 shall be equivalent to:

39993 `setvbuf(stream, buf, _IOFBF, BUFSIZ)`

39994 if *buf* is not a null pointer, or to:

39995 `setvbuf(stream, buf, _IONBF, BUFSIZ)`

39996 if *buf* is a null pointer.

39997 **RETURN VALUE**

39998 The *setbuf*( ) function shall return no value.

39999 **ERRORS**

40000 No errors are defined.

40001 **EXAMPLES**

40002 None.

40003 **APPLICATION USAGE**

40004 A common source of error is allocating buffer space as an ‘‘automatic’’ variable in a code block,
40005 and then failing to close the stream in the same block.

40006 With *setbuf*( ), allocating a buffer of {BUFSIZ} bytes does not necessarily imply that all of
40007 {BUFSIZ} bytes are used for the buffer area.

40008 **RATIONALE**

40009 None.

40010 **FUTURE DIRECTIONS**

40011 None.

40012 **SEE ALSO**

40013 *fopen*( ), *setvbuf*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**

40014 **CHANGE HISTORY**

40015 First released in Issue 1. Derived from Issue 1 of the SVID.

40016 **Issue 6**

40017 The prototype for *setbuf*( ) is updated for alignment with the ISO / IEC 9899: 1999 standard.

40018 **NAME**

40019 setcontext — set current user context

40020 **SYNOPSIS**

40021 XSI
```
#include <ucontext.h>
```

40022
```
int setcontext(const ucontext_t *ucp);
```

40023

40024 **DESCRIPTION**

40025 Refer to *getcontext*( ).

40026 **NAME**

40027 setegid — set effective group ID

40028 **SYNOPSIS**

40029 `#include <unistd.h>`

40030 `int setegid(gid_t gid);`

40031 **DESCRIPTION**

40032 If *gid* is equal to the real group ID or the saved set-group-ID, or if the process has appropriate
40033 privileges, *setegid*( ) shall set the effective group ID of the calling process to *gid*; the real group |
40034 ID, saved set-group-ID, and any supplementary group IDs shall remain unchanged.

40035 The *setegid*( ) function shall not affect the supplementary group list in any way.

40036 **RETURN VALUE**

40037 Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to
40038 indicate the error.

40039 **ERRORS**

40040 The *setegid*( ) function shall fail if:

40041 [EINVAL] The value of the *gid* argument is invalid and is not supported by the
40042 implementation.

40043 [EPERM] The process does not have appropriate privileges and *gid* does not match the
40044 real group ID or the saved set-group-ID.

40045 **EXAMPLES**

40046 None.

40047 **APPLICATION USAGE**

40048 None.

40049 **RATIONALE**

40050 Refer to the RATIONALE section in *setuid*( ).

40051 **FUTURE DIRECTIONS**

40052 None.

40053 **SEE ALSO**

40054 *exec*, *getegid*( ), *geteuid*( ), *getgid*( ), *getuid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the |
40055 Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<unistd.h>** |

40056 **CHANGE HISTORY**

40057 First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

NAME

setenv — add or change environment variable

SYNOPSIS

```
#include <stdlib.h>

int setenv(const char *envname, const char *envval, int overwrite);
```

DESCRIPTION

The *setenv*( ) function updates or adds a variable in the environment of the calling process. The *envname* argument points to a string containing the name of an environment variable to be added or altered. The environment variable shall be set to the value to which *envval* points. The function shall fail if *envname* points to a string which contains an '=' character. If the environment variable named by *envname* already exists and the value of *overwrite* is non-zero, the function shall return success and the environment shall be updated. If the environment variable named by *envname* already exists and the value of *overwrite* is zero, the function shall return success and the environment shall remain unchanged.

If the application modifies *environ* or the pointers to which it points, the behavior of *setenv*( ) is undefined. The *setenv*( ) function shall update the list of pointers to which *environ* points.

The strings described by *envname* and *envval* are copied by this function.

The *setenv*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, zero shall be returned. Otherwise, –1 shall be returned, *errno* set to indicate the error, and the environment shall be unchanged.

ERRORS

The *setenv*( ) function shall fail if:

[EINVAL] The *name* argument is a null pointer, points to an empty string, or points to a string containing an '=' character.

[ENOMEM] Insufficient memory was available to add a variable or its value to the environment.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Unanticipated results may occur if *setenv*( ) changes the external variable *environ*. In particular, if the optional *envp* argument to *main*( ) is present, it is not changed, and thus may point to an obsolete copy of the environment (as may any other copy of *environ*). However, other than the aforementioned restriction, the developers of IEEE Std. 1003.1-200x intended that the traditional method of walking through the environment by way of the *environ* pointer must be supported.

It was decided that *setenv*( ) should be required by this revision because it addresses a piece of missing functionality, and does not impose a significant burden on the implementor.

There was considerable debate as to whether the System V *putenv*( ) function or the BSD *setenv*( ) function should be required as a mandatory function. The *setenv*( ) function was chosen because it permitted the implementation of *unsetenv*( ) function to delete environmental variables, without specifying an additional interface. The *putenv*( ) function is available as an XSI extension.

The standard developers considered requiring that *setenv*( ) indicate an error when a call to it would result in exceeding {ARG_MAX}. The requirement was rejected since the condition might be temporary, with the application eventually reducing the environment size. The ultimate success or failure depends on the size at the time of a call to *exec*, which returns an indication of this error condition.

FUTURE DIRECTIONS

None.

SEE ALSO

getenv( ), *unsetenv*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>**, | **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

NAME

seteuid — set effective user ID

SYNOPSIS

```
#include <unistd.h>

int seteuid(uid_t uid);
```

DESCRIPTION

If *uid* is equal to the real user ID or the saved set-user-ID, or if the process has appropriate privileges, *seteuid*( ) shall set the effective user ID of the calling process to *uid*; the real user ID and saved set-user-ID shall remain unchanged.

The *seteuid*( ) function shall not affect the supplementary group list in any way.

RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *seteuid*( ) function shall fail if:

[EINVAL] The value of the *uid* argument is invalid and is not supported by the implementation.

[EPERM] The process does not have appropriate privileges and *uid* does not match the real group ID or the saved set-group-ID.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Refer to the RATIONALE section in *setuid*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *getegid*( ), *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

40147 **NAME**

40148 setgid — set-group-ID

40149 **SYNOPSIS**

40150 `#include <unistd.h>`

40151 `int setgid(gid_t gid);`

40152 **DESCRIPTION**

40153 If the process has appropriate privileges, *setgid*( ) shall set the real group ID, effective group ID,
40154 and the saved set-group-ID of the calling process to *gid*.

40155 If the process does not have appropriate privileges, but *gid* is equal to the real group ID or the
40156 saved set-group-ID, *setgid*( ) shall set the effective group ID to *gid*; the real group ID and saved
40157 set-group-ID shall remain unchanged.

40158 The *setgid*( ) function shall not affect the supplementary group list in any way.

40159 Any supplementary group IDs of the calling process shall remain unchanged.

40160 **RETURN VALUE**

40161 Upon successful completion, 0 is returned. Otherwise, –1 shall be returned and *errno* set to
40162 indicate the error.

40163 **ERRORS**

40164 The *setgid*( ) function shall fail if:

40165 [EINVAL] The value of the *gid* argument is invalid and is not supported by the
40166 implementation.

40167 [EPERM] The process does not have appropriate privileges and *gid* does not match the
40168 real group ID or the saved set-group-ID.

40169 **EXAMPLES**

40170 None.

40171 **APPLICATION USAGE**

40172 None.

40173 **RATIONALE**

40174 Refer to the RATIONALE section in *setuid*( ).

40175 **FUTURE DIRECTIONS**

40176 None.

40177 **SEE ALSO**

40178 *exec*, *getegid*( ), *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the
40179 Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

40180 **CHANGE HISTORY**

40181 First released in Issue 1. Derived from Issue 1 of the SVID.

40182 **Issue 4**

40183 The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on
40184 XSI-conformant systems.

40185 The following change is incorporated for alignment with the FIPS requirements:

40186 • All references to the saved set-user-ID are marked as extensions. This is because Issue 4
40187 defines this mechanism as mandatory, whereas the ISO POSIX-1 standard defines that it is
40188 only supported if _POSIX_SAVED_IDS is set.

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.
- 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.

40201 **NAME**

40202 setgrent — reset group database to first entry

40203 **SYNOPSIS**

40204 XSI
```
#include <grp.h>
```

40205
```
void setgrent(void);
```

40206

40207 **DESCRIPTION**

40208 Refer to *endgrent*( ).

40209 **NAME**

40210 sethostent — network host database functions

40211 **SYNOPSIS**

40212 `#include <netdb.h>`

40213 `void sethostent(int` *`stayopen`*`);`

40214 **DESCRIPTION**

40215 Refer to *endhostent*( ).

40216 **NAME**

40217 setitimer — set value of interval timer

40218 **SYNOPSIS**

40219 XSI
```
#include <sys/time.h>
```

40220
```
int setitimer(int which, const struct itimerval *restrict value,
```
40221
```
    struct itimerval *restrict ovalue);
```
40222

40223 **DESCRIPTION**

40224 Refer to *getitimer*( ).

40225 **NAME**

40226 setjmp — set jump point for a non-local goto

40227 **SYNOPSIS**

40228 `#include <setjmp.h>`

40229 `int setjmp(jmp_buf` *`env`*`);`

40230 **DESCRIPTION**

40231 CX The functionality described on this reference page is aligned with the ISO C standard. Any
40232 conflict between the requirements described here and the ISO C standard is unintentional. This
40233 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

40234 A call to *setjmp*( ), shall save the calling environment in its *env* argument for later use by
40235 *longjmp*( ).

40236 It is unspecified whether *setjmp*( ) is a macro or a function. If a macro definition is suppressed in
40237 order to access an actual function, or a program defines an external identifier with the name
40238 *setjmp*, the behavior is undefined.

40239 All accessible objects have values as of the time *longjmp*( ) was called, except that the values of
40240 objects of automatic storage duration which are local to the function containing the invocation of
40241 the corresponding *setjmp*( ) which do not have volatile-qualified type and which are changed
40242 between the *setjmp*( ) invocation and *longjmp*( ) call are indeterminate.

40243 An application shall ensure that an invocation of *setjmp*( ) appears in one of the following
40244 contexts only:

40245 - The entire controlling expression of a selection or iteration statement
40246 - One operand of a relational or equality operator with the other operand an integral constant
40247 expression, with the resulting expression being the entire controlling expression of a
40248 selection or iteration statement
40249 - The operand of a unary `'!'` operator with the resulting expression being the entire
40250 controlling expression of a selection or iteration
40251 - The entire expression of an expression statement (possibly cast to **void**)

40252 If the invocation appears in any other context, the behavior is undefined.

40253 **RETURN VALUE**

40254 If the return is from a direct invocation, *setjmp*( ) shall return 0. If the return is from a call to
40255 *longjmp*( ), *setjmp*( ) shall return a non-zero value.

40256 **ERRORS**

40257 No errors are defined.

40258 **EXAMPLES**

40259 None.

40260 **APPLICATION USAGE**

40261 In general, *sigsetjmp*( ) is more useful in dealing with errors and interrupts encountered in a low-
40262 level subroutine of a program.

40263 **RATIONALE**

40264 None.

40265 **FUTURE DIRECTIONS**

40266 None.

40267 **SEE ALSO**

40268 *longjmp*( ), *sigsetjmp*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<setjmp.h>**

40269 **CHANGE HISTORY**

40270 First released in Issue 1. Derived from Issue 1 of the SVID.

40271 **Issue 4**

40272 This issue states that *setjmp*( ) is a macro or a function; previous issues stated that it was a macro.
40273 Warnings have also been added about the suppression of a *setjmp*( ) macro definition.

40274 Text describing the accessibility of objects after a *longjmp*( ) call is added to the DESCRIPTION.
40275 This text is imported from the entry for *longjmp*( ).

40276 Text describing the contexts in which calls to *setjmp*( ) are valid is moved to the DESCRIPTION
40277 from the APPLICATION USAGE section.

40278 The APPLICATION USAGE section is changed to refer to *sigsetjmp*( ).

40279 **Issue 6**

40280 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

setkey — set encoding key (**CRYPT**)

SYNOPSIS

XSI
```
#include <stdlib.h>

void setkey(const char *key);
```

DESCRIPTION

The *setkey*( ) function provides (rather primitive) access to an implementation-defined encoding algorithm. The argument of *setkey*( ) is an array of length 64 bytes containing only the bytes with numerical value of 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is used by the algorithm. This is the key that shall be used with the algorithm to encode a string *block* passed to *encrypt*( ).

The *setkey*( ) 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 *setkey*( ). If *errno* is non-zero on return, an error has occurred.

The *setkey*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

No values are returned.

ERRORS

The *setkey*( ) function shall fail if:

[ENOSYS] The functionality is not supported on this implementation.

EXAMPLES

None.

APPLICATION USAGE

Decoding need not be implemented in all environments. This is related to U.S. Government restrictions on encryption and decryption routines: the DES decryption algorithm cannot be exported outside the U.S. 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( ), *encrypt*( ), 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 4

The type of argument *key* is changed from **char*** to **const char***.

The description of the array is put in terms of bytes instead of characters.

The APPLICATION USAGE section is added.

40323 **Issue 5**

40324 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

40325 **NAME**

40326 setlocale — set program locale

40327 **SYNOPSIS**

40328 `#include <locale.h>`

40329 `char *setlocale(int` *`category`*`, const char *`*`locale`*`);`

40330 **DESCRIPTION**

40331 CX The functionality described on this reference page is aligned with the ISO C standard. Any
40332 conflict between the requirements described here and the ISO C standard is unintentional. This
40333 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

40334 The *setlocale*( ) function selects the appropriate piece of the program's locale, as specified by the
40335 *category* and *locale* arguments, and may be used to change or query the program's entire locale or
40336 portions thereof. The value *LC_ALL* for *category* names the program's entire locale; other values
40337 for *category* name only a part of the program's locale:

40338 *LC_COLLATE* Affects the behavior of regular expressions and the collation functions.

40339 *LC_CTYPE* Affects the behavior of regular expressions, character classification, character
40340 conversion functions, and wide-character functions.

40341 CX *LC_MESSAGES* Affects what strings are expected by commands and utilities as affirmative or
40342 negative responses.

40343 XSI It also affects what strings are given by commands and utilities as affirmative
40344 or negative responses, and the content of messages.

40345 *LC_MONETARY* Affects the behavior of functions that handle monetary values.

40346 *LC_NUMERIC* Affects the behavior of functions that handle numeric values. |

40347 *LC_TIME* Affects the behavior of the time conversion functions.

40348 The *locale* argument is a pointer to a character string containing the required setting of *category*. |
40349 The contents of this string are implementation-defined. In addition, the following preset values |
40350 of *locale* are defined for all settings of *category*:

40351 CX "POSIX" Specifies the minimal environment for C-language translation called POSIX
40352 locale. If *setlocale*( ) is not invoked, the POSIX locale is the default at entry to
40353 *main*( ).

40354 "C" Same as "POSIX".

40355 "" Specifies an implementation-defined native environment. For XSI-conformant |
40356 systems, this corresponds to the value of the associated environment
40357 variables, *LC_** and *LANG*; see the Base Definitions volume of |
40358 IEEE Std. 1003.1-200x, Chapter 7, Locale and the Base Definitions volume of |
40359 IEEE Std. 1003.1-200x, Chapter 8, Environment Variables. |

40360 A null pointer Used to direct *setlocale*( ) to query the current internationalized environment
40361 and return the name of the *locale*( ).

40362 THR The locale state is common to all threads within a process.

40363 **RETURN VALUE**

40364 Upon successful completion, *setlocale*( ) shall return the string associated with the specified
40365 category for the new locale. Otherwise, *setlocale*( ) shall return a null pointer and the program's
40366 locale is not changed.

A null pointer for *locale* causes *setlocale*( ) to return a pointer to the string associated with the *category* for the program's current locale. The program's locale shall not be changed.

The string returned by *setlocale*( ) is such that a subsequent call with that string and its associated *category* shall restore that part of the program's locale. The application shall not modify the string returned which may be overwritten by a subsequent call to *setlocale*( ).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The following code illustrates how a program can initialize the international environment for one language, while selectively modifying the program's locale such that regular expressions and string operations can be applied to text recorded in a different language:

```
setlocale(LC_ALL, "De");
setlocale(LC_COLLATE, "Fr@dict");
```

Internationalized programs must call *setlocale*( ) to initiate a specific language operation. This can be done by calling *setlocale*( ) as follows:

```
setlocale(LC_ALL, "");
```

Changing the setting of *LC_MESSAGES* has no effect on catalogs that have already been opened by calls to *catopen*( ).

RATIONALE

The ISO C standard defines a collection of functions to support internationalization. One of the most significant aspects of these functions is a facility to set and query the *international environment*. The international environment is a repository of information that affects the behavior of certain functionality, namely:

1. Character handling
2. String handling (that is, collating)
3. Date / time formatting
4. Numeric editing

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 five 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*( ) is 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.8859");
```

In this example, all categories of the international environment are set to the locale corresponding to the string `"Fr_FR.8859"`, 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.

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.

setlocale(*category*,"")
: This sets a specific category to an implementation-defined default. For POSIX-conforming systems, this corresponds to the value of the environment variables.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *localeconv*( ), *mblen*( ), *mbstowcs*( ), *mbtowc*( ), *nl_langinfo*( ), *printf*( ), *scanf*( ), *setlocale*( ), *strcoll*( ), *strerror*( ), *strfmon*( ), *strtod*( ), *strxfrm*( ), *tolower*( ), *toupper*( ), *towlower*( ), *towupper*( ), *wcscoll*( ), *wcstod*( ), *wcstombs*( ), *wcsxfrm*( ), *wctomb*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<langinfo.h>**, **<locale.h>**

40453 **CHANGE HISTORY**

40454 First released in Issue 3.

40455 **Issue 4**

40456 The description of *LC_MESSAGES* is extended to indicate that this category also determines
40457 what strings are produced by commands and utilities for affirmative and negative responses,
40458 and that it affects the content of other program messages. This is marked as an extension.

40459 References to *nl_langinfo*( ) are removed.

40460 The description of the implementation-defined native locale ("" is clarified by stating the related |
40461 environment variables explicitly.

40462 The APPLICATION USAGE section is expanded.

40463 The following changes are incorporated for alignment with the ISO C standard and the
40464 ISO POSIX-1 standard:

40465 • The type of the argument *locale* is changed from **char*** to **const char***.

40466 • The name "POSIX" is added to the list of standard locale names.

40467 The following change is incorporated for alignment with the ISO POSIX-2 standard:

40468 • The *LC_MESSAGES* value for *category* is added to the DESCRIPTION.

40469 **Issue 5**

40470 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

40471 **Issue 6**

40472 Extensions beyond the ISO C standard are now marked.

40473 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

40474 **NAME**

40475 setlogmask — set log priority mask

40476 **SYNOPSIS**

40477 XSI
```
#include <syslog.h>
```

40478
```
int setlogmask(int maskpri);
```

40479

40480 **DESCRIPTION**

40481 Refer to *closelog*( ).

40482 **NAME**

40483 setnetent — network database function

40484 **SYNOPSIS**

40485 `#include <netdb.h>`

40486 `void setnetent(int stayopen);`

40487 **DESCRIPTION**

40488 Refer to *endnetent*( ).

NAME

setpgid — set process group ID for job control

SYNOPSIS

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

DESCRIPTION

The *setpgid*( ) function is used either to join an existing process group or create a new process group within the session of the calling process. The process group ID of a session leader shall not change. Upon successful completion, the process group ID of the process with a process ID that matches *pid* shall be set to *pgid*. As a special case, if *pid* is 0, the process ID of the calling process shall be used. Also, if *pgid* is 0, the process group ID of the indicated process shall be used.

RETURN VALUE

Upon successful completion, *setpgid*( ) shall return 0; otherwise, –1 shall be returned and *errno* shall be set to indicate the error.

ERRORS

The *setpgid*( ) function shall fail if:

[EACCES] The value of the *pid* argument matches the process ID of a child process of the calling process and the child process has successfully executed one of the *exec* functions.

[EINVAL] The value of the *pgid* argument is less than 0, or is not a value supported by the implementation.

[EPERM] The process indicated by the *pid* argument is a session leader.

[EPERM] The value of the *pid* argument matches the process ID of a child process of the calling process and the child process is not in the same session as the calling process.

[EPERM] The value of the *pgid* argument is valid but does not match the process ID of the process indicated by the *pid* argument and there is no process with a process group ID that matches the value of the *pgid* argument in the same session as the calling process.

[ESRCH] The value of the *pid* argument does not match the process ID of the calling process or of a child process of the calling process.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *setpgid*( ) function is used to group processes together for the purpose of signaling, placement in foreground or background, and other job control actions.

The *setpgid*( ) function is similar to the *setpgrp*( ) function of 4.2 BSD, except that 4.2 BSD allowed the specified new process group to assume any value. This presents certain security problems and is more flexible than necessary to support job control.

To provide tighter security, *setpgid*( ) only allows the calling process to join a process group already in use inside its session or create a new process group whose process group ID was

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 program is launched via one of the *exec* functions.
2. The new process must be placed in the new process group before the shell can correctly send signals to the new process group.

To address these constraints, the following actions are performed. The new processes call *setpgid*( ) to alter their own process groups after *fork*( ) but before *exec*. This satisfies the first constraint. Under 4.3 BSD, the second constraint is satisfied by the synchronization property of *vfork*( ); that is, the shell is suspended until the child has completed the *exec*, thus ensuring that the child has completed the *setpgid*( ). A new version of *fork*( ) with this same synchronization property was considered, but it was decided instead to merely allow the parent shell process to adjust the process group of its child processes via *setpgid*( ). Both timing constraints are now satisfied by having both the parent shell and the child attempt to adjust the process group of the child process; it does not matter which succeeds first.

Because it would be confusing to an application to have its process group change after it began executing (that is, after *exec*), and because the child process would already have adjusted its process group before this, the [EACCES] error was added to disallow this.

One non-obvious use of *setpgid*( ) is to allow a job control shell to return itself to its original process group (the one in effect when the job control shell was executed). A job control shell does this before returning control back to its parent when it is terminating or suspending itself as a way of restoring its job control ''state'' back to what its parent would expect. (Note that the original process group of the job control shell typically matches the process group of its parent, but this is not necessarily always the case.)

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *getpgrp*( ), *setsid*( ), *tcsetpgrp*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The function is no longer marked as OPTIONAL FUNCTIONALITY.

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

The **<unistd.h>** header is added to the SYNOPSIS section.

The DESCRIPTION in Issue 3 defined the behavior of this function for implementations that either supported or did not support job control. As job control is defined as mandatory in Issue 4, only the former of these is now described.

The [ENOSYS] error is removed from the ERRORS section.

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.
- The *setpgid*( ) function is mandatory since _POSIX_JOB_CONTROL is required to be defined in this issue. This is a FIPS requirement.

40584 **NAME**

40585 setpgrp — set process group ID

40586 **SYNOPSIS**

40587 XSI
```
#include <unistd.h>
```

40588
```
pid_t setpgrp(void);
```

40589

40590 **DESCRIPTION**

40591 If the calling process is not already a session leader, *setpgrp*( ) sets the process group ID of the
40592 calling process to the process ID of the calling process. If *setpgrp*( ) creates a new session, then
40593 the new session has no controlling terminal.

40594 The *setpgrp*( ) function has no effect when the calling process is a session leader.

40595 **RETURN VALUE**

40596 Upon completion, *setpgrp*( ) shall return the process group ID.

40597 **ERRORS**

40598 No errors are defined.

40599 **EXAMPLES**

40600 None.

40601 **APPLICATION USAGE**

40602 None.

40603 **RATIONALE**

40604 None.

40605 **FUTURE DIRECTIONS**

40606 None.

40607 **SEE ALSO**

40608 *exec*, *fork*( ), *getpid*( ), *getsid*( ), *kill*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of |
40609 IEEE Std. 1003.1-200x, **<unistd.h>** |

40610 **CHANGE HISTORY**

40611 First released in Issue 4, Version 2.

40612 **Issue 5**

40613 Moved from X/OPEN UNIX extension to BASE.

40614 **NAME**

40615 setpriority — set the nice value

40616 **SYNOPSIS**

40617 XSI
```
#include <sys/resource.h>
```

40618
```
int setpriority(int which, id_t who, int nice);
```

40619

40620 **DESCRIPTION**

40621 Refer to *getpriority*( ).

40622 **NAME**

40623 setprotoent — network protocol database functions

40624 **SYNOPSIS**

40625 `#include <netdb.h>`

40626 `void setprotoent(int stayopen);`

40627 **DESCRIPTION**

40628 Refer to *endprotoent*( ).

40629 **NAME**

40630 setpwent — user database function

40631 **SYNOPSIS**

40632 XSI
```
#include <pwd.h>
```

40633
```
void setpwent(void);
```

40634

40635 **DESCRIPTION**

40636 Refer to *endpwent*( ).

40637 **NAME**

40638 setregid — set real and effective group IDs

40639 **SYNOPSIS**

40640 XSI
```
#include <unistd.h>
```

40641
```
int setregid(gid_t rgid, gid_t egid);
```

40642

40643 **DESCRIPTION**

40644 The *setregid*( ) function is used to set the real and effective group IDs of the calling process.

40645 If *rgid* is –1, the real group ID shall not be changed; if *egid* is –1, the effective group ID shall not
40646 be changed.

40647 The real and effective group IDs may be set to different values in the same call.

40648 Only a process with appropriate privileges can set the real group ID and the effective group ID
40649 to any valid value.

40650 A non-privileged process can set either the real group ID to the saved set-group-ID from one of |
40651 the *exec* family of functions, or the effective group ID to the saved set-group-ID or the real group |
40652 ID. |

40653 Any supplementary group IDs of the calling process remain unchanged.

40654 **RETURN VALUE**

40655 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
40656 indicate the error, and neither of the group IDs are changed.

40657 **ERRORS**

40658 The *setregid*( ) function shall fail if:

40659 [EINVAL] The value of the *rgid* or *egid* argument is invalid or out-of-range. |

40660 [EPERM] The process does not have appropriate privileges and a change other than |
40661 changing the real group ID to the saved set-group-ID, or changing the
40662 effective group ID to the real group ID or the saved set-group-ID, was
40663 requested.

40664 **EXAMPLES**

40665 None.

40666 **APPLICATION USAGE**

40667 If a set-group-ID process sets its effective group ID to its real group ID, it can still set its effective
40668 group ID back to the saved set-group-ID.

40669 **RATIONALE**

40670 None.

40671 **FUTURE DIRECTIONS**

40672 None.

40673 **SEE ALSO**

40674 *exec*, *getegid*( ), *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setreuid*( ), *setuid*( ), the |
40675 Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>** |

40676 **CHANGE HISTORY**

40677 First released in Issue 4, Version 2.

40678 **Issue 5**

40679 Moved from X/OPEN UNIX extension to BASE.

40680 The DESCRIPTION is updated to indicate that the saved set-group-ID can be set by any of the
40681 *exec* family of functions, not just *execev*( ).

40682 **NAME**

40683 setreuid — set real and effective user IDs

40684 **SYNOPSIS**

40685 XSI
```
#include <unistd.h>
```

40686
```
int setreuid(uid_t ruid, uid_t euid);
```

40687

40688 **DESCRIPTION**

40689 The *setreuid*( ) function sets the real and effective user IDs of the current process to the values
40690 specified by the *ruid* and *euid* arguments. If *ruid* or *euid* is –1, the corresponding effective or real
40691 user ID of the current process is left unchanged.

40692 A process with appropriate privileges can set either ID to any value. An unprivileged process
40693 can only set the effective user ID if the *euid* argument is equal to either the real, effective, or
40694 saved user ID of the process.

40695 It is unspecified whether a process without appropriate privileges is permitted to change the real
40696 user ID to match the current real, effective, or saved set-user-ID of the process.

40697 **RETURN VALUE**

40698 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
40699 indicate the error.

40700 **ERRORS**

40701 The *setreuid*( ) function shall fail if:

40702 [EINVAL] The value of the *ruid* or *euid* argument is invalid or out-of-range. |

40703 [EPERM] The current process does not have appropriate privileges, and either an |
40704 attempt was made to change the effective user ID to a value other than the
40705 real user ID or the saved set-user-ID or an attempt was made to change the
40706 real user ID to a value not permitted by the implementation.

40707 **EXAMPLES**

40708 **Setting the Effective User ID to the Real User ID**

40709 The following example sets the effective user ID of the calling process to the real user ID, so that
40710 files created later will be owned by the current user.

40711
```
#include <unistd.h>
```
40712
```
#include <sys/types.h>
```
40713
```
...
```
40714
```
setreuid(getuid(), getuid());
```
40715
```
...
```

40716 **APPLICATION USAGE**

40717 None.

40718 **RATIONALE**

40719 None.

40720 **FUTURE DIRECTIONS**

40721 None.

40722 **SEE ALSO**

40723 *getegid*( ), *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setuid*( ), the Base |
40724 Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>** |

40725 **CHANGE HISTORY**

40726 First released in Issue 4, Version 2.

40727 **Issue 5**

40728 Moved from X/OPEN UNIX extension to BASE.

40729 **NAME**

40730 setrlimit — control maximum resource consumption

40731 **SYNOPSIS**

40732 XSI
```
#include <sys/resource.h>
```

40733
```
int setrlimit(int resource, const struct rlimit *rlp);
```

40734

40735 **DESCRIPTION**

40736 Refer to *getrlimit*( ).

40737 **NAME**

40738 setservent — network services database functions

40739 **SYNOPSIS**

40740 `#include <netdb.h>`

40741 `void setservent(int` *`stayopen`*`);`

40742 **DESCRIPTION**

40743 Refer to *endservent*( ).

40744 **NAME**

40745 setsid — create session and set process group ID

40746 **SYNOPSIS**

40747 `#include <unistd.h>`

40748 `pid_t setsid(void);`

40749 **DESCRIPTION**

40750 The *setsid*( ) function creates a new session, if the calling process is not a process group leader.
40751 Upon return the calling process shall be the session leader of this new session, shall be the
40752 process group leader of a new process group, and shall have no controlling terminal. The
40753 process group ID of the calling process shall be set equal to the process ID of the calling process.
40754 The calling process shall be the only process in the new process group and the only process in
40755 the new session.

40756 **RETURN VALUE**

40757 Upon successful completion, *setsid*( ) shall return the value of the new process group ID of the
40758 calling process. Otherwise, it shall return (**pid_t**)–1 and set *errno* to indicate the error.

40759 **ERRORS**

40760 The *setsid*( ) function shall fail if:

40761 [EPERM] The calling process is already a process group leader, or the process group ID |
40762 of a process other than the calling process matches the process ID of the
40763 calling process.

40764 **EXAMPLES**

40765 None.

40766 **APPLICATION USAGE**

40767 None.

40768 **RATIONALE**

40769 The *setsid*( ) function is similar to the *setpgrp*( ) function of System V. System V, without job
40770 control, groups processes into process groups and creates new process groups via *setpgrp*( ); only
40771 one process group may be part of a login session.

40772 Job control allows multiple process groups within a login session. In order to limit job control
40773 actions so that they can only affect processes in the same login session, this volume of
40774 IEEE Std. 1003.1-200x adds the concept of a session that is created via *setsid*( ). The *setsid*( )
40775 function also creates the initial process group contained in the session. Additional process
40776 groups can be created via the *setpgid*( ) function. A System V process group would correspond to
40777 a POSIX System Interfaces session containing a single POSIX process group. Note that this
40778 function requires that the calling process not be a process group leader. The usual way to ensure
40779 this is true is to create a new process with *fork*( ) and have it call *setsid*( ). The *fork*( ) function
40780 guarantees that the process ID of the new process does not match any existing process group ID.

40781 **FUTURE DIRECTIONS**

40782 None.

40783 **SEE ALSO**

40784 *getsid*( ), *setpgid*( ), *setpgrp*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, |
40785 **<unistd.h>**

40786 **CHANGE HISTORY**

40787 First released in Issue 3.

40788 Entry included for alignment with the POSIX.1-1988 standard.

40789 **Issue 4**

40790 The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on
40791 XSI-conformant systems.

40792 The **<unistd.h>** header is added to the SYNOPSIS section.

40793 The argument list is explicitly defined as **void**.

40794 **Issue 6**

40795 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

40796 The following new requirements on POSIX implementations derive from alignment with the
40797 Single UNIX Specification:

40798 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
40799 required for conforming implementations of previous POSIX specifications, it was not
40800 required for UNIX applications.

40801 **NAME**

40802 setsockopt — set the socket options

40803 **SYNOPSIS**

40804 `#include <sys/socket.h>`

40805 `int setsockopt(int socket, int level, int option_name,`
40806 `    const void *option_value, socklen_t option_len);`

40807 **DESCRIPTION**

40808 The *setsockopt*( ) function sets the option specified by the *option_name* argument, at the protocol
40809 level specified by the *level* argument, to the value pointed to by the *option_value* argument for the
40810 socket associated with the file descriptor specified by the *socket* argument.

40811 The *level* argument specifies the protocol level at which the option resides. To set options at the
40812 socket level, specify the *level* argument as SOL_SOCKET. To set options at other levels, supply
40813 the appropriate *level* identifier for the protocol controlling the option. For example, to indicate |
40814 that an option is interpreted by the TCP (Transport Control Protocol), set *level* to IPPROTO_TCP |
40815 as defined in the **<netinet/in.h>** header.

40816 The *option_name* argument specifies a single option to set. The *option_name* argument and any
40817 specified options are passed uninterpreted to the appropriate protocol module for
40818 interpretations. The **<sys/socket.h>** header defines the socket-level options. The options are as
40819 follows:

40820 SO_DEBUG — Turns on recording of debugging information. This option enables or
40821 disables debugging in the underlying protocol modules. This option takes
40822 an **int** value. This is a Boolean option.

40823 SO_BROADCAST — Permits sending of broadcast messages, if this is supported by the
40824 protocol. This option takes an **int** value. This is a Boolean option.

40825 SO_REUSEADDR — Specifies that the rules used in validating addresses supplied to *bind*( )
40826 should allow reuse of local addresses, if this is supported by the protocol.
40827 This option takes an **int** value. This is a Boolean option.

40828 SO_KEEPALIVE — Keeps connections active by enabling the periodic transmission of
40829 messages, if this is supported by the protocol. This option takes an **int**
40830 value.

40831 If the connected socket fails to respond to these messages, the connection |
40832 is broken and threads writing to that socket are notified with a SIGPIPE |
40833 signal.

40834 This is a Boolean option.

40835 SO_LINGER — Lingers on a *close*( ) if data is present. This option controls the action
40836 taken when unsent messages queue on a socket and *close*( ) is performed.
40837 If SO_LINGER is set, the system blocks the process during *close*( ) until it
40838 can transmit the data or until the time expires. If SO_LINGER is not
40839 specified, and *close*( ) is issued, the system handles the call in a way that
40840 allows the process to continue as quickly as possible. This option takes a
40841 **linger** structure, as defined in the **<sys/socket.h>** header, to specify the
40842 state of the option and linger interval.

40843 SO_OOBINLINE — Leaves received out-of-band data (data marked urgent) inline. This
40844 option takes an **int** value. This is a Boolean option.

SO_SNDBUF Sets send buffer size. This option takes an **int** value.

SO_RCVBUF Sets receive buffer size. This option takes an **int** value.

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.

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 than 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.

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 *errno* 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.

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.

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 *errno* 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.

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.

40891 [EDOM] The send and receive timeout values are too big to fit into the timeout fields in
40892 the socket structure.

40893 [EINVAL] The specified option is invalid at the specified socket level or the socket has
40894 been shut down.

40895 [EISCONN] The socket is already connected, and a specified option cannot be set while the
40896 socket is connected.

40897 [ENOPROTOOPT]
40898 The option is not supported by the protocol.

40899 [ENOTSOCK] The *socket* argument does not refer to a socket.

40900 The *setsockopt*( ) function may fail if:

40901 [ENOMEM] There was insufficient memory available for the operation to complete.

40902 [ENOBUFS] Insufficient resources are available in the system to complete the call.

40903 **EXAMPLES**

40904 None.

40905 **APPLICATION USAGE**

40906 The *setsockopt*( ) function provides an application program with the means to control socket
40907 behavior. An application program can use *setsockopt*( ) to allocate buffer space, control timeouts,
40908 or permit socket data broadcasts. The **<sys/socket.h>** header defines the socket-level options
40909 available to *setsockopt*( ).

40910 Options may exist at multiple protocol levels. The SO_ options are always present at the
40911 uppermost socket level.

40912 **RATIONALE**

40913 None.

40914 **FUTURE DIRECTIONS**

40915 None.

40916 **SEE ALSO**

40917 Section 2.10 (on page 562), *bind*( ), *endprotoent*( ), *getsockopt*( ), *socket*( ), the Base Definitions
40918 volume of IEEE Std. 1003.1-200x, **<netinet/in.h>**, **<sys/socket.h>**

40919 **CHANGE HISTORY**

40920 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

40921 **NAME**

40922 setstate — switch pseudorandom number generator state arrays

40923 **SYNOPSIS**

40924 XSI
```
#include <stdlib.h>
```

40925
```
char *setstate(const char *state);
```

40926

40927 **DESCRIPTION**

40928 Refer to *initstate*( ).

40929 **NAME**

40930 setuid — set user ID

40931 **SYNOPSIS**

40932 `#include <unistd.h>`

40933 `int setuid(uid_t uid);`

40934 **DESCRIPTION**

40935 If the process has appropriate privileges, *setuid*( ) shall set the real user ID, effective user ID, and
40936 the saved set-user-ID of the calling process to *uid*.

40937 If the process does not have appropriate privileges, but *uid* is equal to the real user ID or the
40938 saved set-user-ID, *setuid*( ) shall set the effective user ID to *uid*; the real user ID and saved set-
40939 user-ID shall remain unchanged.

40940 The *setuid*( ) function shall not affect the supplementary group list in any way.

40941 **RETURN VALUE**

40942 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
40943 indicate the error.

40944 **ERRORS**

40945 The *setuid*( ) function shall fail, return –1, and set *errno* to the corresponding value if one or more
40946 of the following are true:

40947 [EINVAL] The value of the *uid* argument is invalid and not supported by the
40948 implementation.

40949 [EPERM] The process does not have appropriate privileges and *uid* does not match the
40950 real user ID or the saved set-user-ID.

40951 **EXAMPLES**

40952 None.

40953 **APPLICATION USAGE**

40954 None.

40955 **RATIONALE**

40956 The various behaviors of the *setuid*( ) and *setgid*( ) functions when called by non-privileged
40957 processes reflect the behavior of different historical implementations. For portability, it is
40958 recommended that new non-privileged applications use the *seteuid*( ) and *setegid*( ) functions
40959 instead.

40960 The saved set-user-ID capability allows a program to regain the effective user ID established at
40961 the last *exec* call. Similarly, the saved set-group-ID capability allows a program to regain the
40962 effective group ID established at the last *exec* call. These capabilities are derived from System V.
40963 Without them, a program might have to run as superuser in order to perform the same
40964 functions, because superuser can write on the user's files. This is a problem because such a
40965 program can write on any user's files, and so must be carefully written to emulate the
40966 permissions of the calling process properly. In System V, these capabilities have traditionally
40967 been implemented only via the *setuid*( ) and *setgid*( ) functions for non-privileged processes. The
40968 fact that the behavior of those functions was different for privileged processes made them
40969 difficult to use. The POSIX.1-1990 standard defined the *setuid*( ) function to behave differently
40970 for privileged and unprivileged users. When the caller had the appropriate privilege, the
40971 function set the calling process's real user ID, effective user ID, and saved set-user ID on
40972 implementations that supported it. When the caller did not have the appropriate privilege, the
40973 function set only the effective user ID, subject to permission checks. The former use is generally
40974 needed for utilities like *login* and *su*, which are not portable applications and thus outside the

scope of IEEE Std. 1003.1-200x. These utilities wish to change the user ID irrevocably to a new value, generally that of an unprivileged user. The latter use is needed for portable applications that are installed with the set-user-ID bit and need to perform operations using the real user ID.

IEEE Std. 1003.1-200x augments the latter functionality with a mandatory feature named _POSIX_SAVED_IDS. This feature permits a set-user-ID application to switch its effective user ID back and forth between the values of its *exec*-time real user ID and effective user ID. Unfortunately, the POSIX.1-1990 standard did not permit a portable application using this feature to work properly when it happened to be executed with the (implementation-defined) appropriate privilege. Furthermore, the application did not even have a means to tell whether it had this privilege. Because the saved set-user-ID feature is quite desirable for applications, as evidenced by the fact that NIST required it in FIPS 151-2, it has been mandated by IEEE Std. 1003.1-200x. However, there are implementors who have been reluctant to support it given the limitation described above.

The 4.3BSD system handles the problem by supporting separate functions: *setuid*( ) (which always sets both the real and effective user IDs, like *setuid*( ) in IEEE Std. 1003.1-200x for privileged users), and *seteuid*( ) (which always sets just the effective user ID, like *setuid*( ) in IEEE Std. 1003.1-200x for non-privileged users). This separation of functionality into distinct functions seems desirable. 4.3BSD does not support the saved set-user-ID feature. It supports similar functionality of switching the effective user ID back and forth via *setreuid*( ), which permits reversing the real and effective user IDs. This model seems less desirable than the saved set-user-ID because the real user ID changes as a side effect. The current 4.4BSD includes saved effective IDs and uses them for *seteuid*( ) and *setegid*( ) as described above. The *setreuid*( ) and *setregid*( ) functions will be deprecated or removed.

The solution here is:

- Require that all implementations support the functionality of the saved set-user-ID, which is set by the *exec* functions and by privileged calls to *setuid*( ).
- Add the *seteuid*( ) and *setegid*( ) functions as portable alternatives to *setuid*( ) and *setgid*( ) for non-privileged and privileged processes.

Historical systems have provided two mechanisms for a set-user-ID process to change its effective user ID to be the same as its real user ID in such a way that it could return to the original effective user ID: the use of the *setuid*( ) function in the presence of a saved set-user-ID, or the use of the BSD *setreuid*( ) function, which was able to swap the real and effective user IDs. The changes included in IEEE Std. 1003.1-200x provide a new mechanism using *seteuid*( ) in conjunction with a saved set-user-ID. Thus, all implementations with the new *seteuid*( ) mechanism will have a saved set-user-ID for each process, and most of the behavior controlled by _POSIX_SAVED_IDS has been changed to agree with the case where the option was defined. The *kill*( ) function is an exception. Implementors of the new *seteuid*( ) mechanism will generally be required to maintain compatibility with the older mechanisms previously supported by their systems. However, compatibility with this use of *setreuid*( ) and with the _POSIX_SAVED_IDS behavior of *kill*( ) is unfortunately complicated. If an implementation with a saved set-user-ID allows a process to use *setreuid*( ) to swap its real and effective user IDs, but were to leave the saved set-user-ID unmodified, the process would then have an effective user ID equal to the original real user ID, and both real and saved set-user-ID would be equal to the original effective user ID. In that state, the real user would be unable to kill the process, even though the effective user ID of the process matches that of the real user, if the *kill*( ) behavior of _POSIX_SAVED_IDS was used. This is obviously not acceptable. The alternative choice, which is used in at least one implementation, is to change the saved set-user-ID to the effective user ID during most calls to *setreuid*( ). The standard developers considered that alternative to be less correct than the retention of the old behavior of *kill*( ) in such systems. Current conforming applications shall

41024 accommodate either behavior from *kill*( ), and there appears to be no strong reason for *kill*( ) to
41025 check the saved set-user-ID rather than the effective user ID.

41026 **FUTURE DIRECTIONS**

41027 None.

41028 **SEE ALSO**

41029 *exec*, *getegid*( ), *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), the
41030 Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

41031 **CHANGE HISTORY**

41032 First released in Issue 1. Derived from Issue 1 of the SVID.

41033 **Issue 4**

41034 The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on
41035 XSI-conformant systems.

41036 The **<unistd.h>** header is added to the SYNOPSIS section.

41037 The following change is incorporated for alignment with the FIPS requirements:

41038 • All references to the saved set-user-ID are marked as extensions. This is because Issue 4
41039 defines this mechanism as mandatory, whereas the ISO POSIX-1 standard defines that it is
41040 only supported if _POSIX_SAVED_IDS is set.

41041 **Issue 6**

41042 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

41043 The following new requirements on POSIX implementations derive from alignment with the
41044 Single UNIX Specification:

41045 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
41046 required for conforming implementations of previous POSIX specifications, it was not
41047 required for UNIX applications.

41048 • The functionality associated with _POSIX_SAVED_IDS is now mandatory. This is a FIPS
41049 requirement.

41050 The following changes were made to align with the IEEE P1003.1a draft standard:

41051 • The effects of *setuid*( ) in processes without appropriate privileges are changed.

41052 • A requirement that the supplementary group list is not affected is added.

41053 **NAME**

41054 setutxent — reset user accounting database to first entry

41055 **SYNOPSIS**

41056 XSI
```
#include <utmpx.h>
```

41057
```
void setutxent(void);
```

41058

41059 **DESCRIPTION**

41060 Refer to *endutxent*( ).

41061 **NAME**

41062 setvbuf — assign buffering to a stream

41063 **SYNOPSIS**

41064
```
#include <stdio.h>

int setvbuf(FILE *restrict stream, char *restrict buf, int type,
    size_t size);
```

41067 **DESCRIPTION**

41068 CX The functionality described on this reference page is aligned with the ISO C standard. Any
41069 conflict between the requirements described here and the ISO C standard is unintentional. This
41070 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

41071 The *setvbuf*( ) function may be used after the stream pointed to by *stream* is associated with an
41072 open file but before any other operation (other than an unsuccessful call to *setvbuf*( )) is
41073 performed on the stream. The argument *type* determines how *stream* shall be buffered, as
41074 follows:

41075 • {_IOFBF} shall cause input/output to be fully buffered.

41076 • {_IOLBF} shall cause input/output to be line buffered.

41077 • {_IONBF} shall cause input/output to be unbuffered.

41078 If *buf* is not a null pointer, the array it points to may be used instead of a buffer allocated by
41079 *setvbuf*( ) and the argument *size* specifies the size of the array; otherwise, *size* may determine the
41080 size of a buffer allocated by the *setvbuf*( ) function. The contents of the array at any time are
41081 indeterminate.

41082 For information about streams, see Section 2.5 (on page 535).

41083 **RETURN VALUE**

41084 Upon successful completion, *setvbuf*( ) shall return 0. Otherwise, it shall return a non-zero value
41085 CX if an invalid value is given for *type* or if the request cannot be honored, and may set *errno* to
41086 indicate the error.

41087 **ERRORS**

41088 The *setvbuf*( ) function may fail if:

41089 CX [EBADF] The file descriptor underlying *stream* is not valid.

41090 **EXAMPLES**

41091 None.

41092 **APPLICATION USAGE**

41093 A common source of error is allocating buffer space as an ‘‘automatic’’ variable in a code block,
41094 and then failing to close the stream in the same block.

41095 With *setvbuf*( ), allocating a buffer of *size* bytes does not necessarily imply that all of *size* bytes are
41096 used for the buffer area.

41097 Applications should note that many implementations only provide line buffering on input from
41098 terminal devices.

41099 **RATIONALE**

41100 None.

41101 **FUTURE DIRECTIONS**

41102 None.

41103 **SEE ALSO**

41104 *fopen*( ), *setbuf*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>** |

41105 **CHANGE HISTORY**

41106 First released in Issue 1. Derived from Issue 1 of the SVID. |

41107 **Issue 4**

41108 The second paragraph of the DESCRIPTION is now in Section 2.5 (on page 535).

41109 The [EBADF] error is marked as an extension.

41110 The APPLICATION USAGE section is expanded.

41111 The following change is incorporated for alignment with the ISO C standard:

41112 • This function is no longer marked as an extension.

41113 **Issue 6**

41114 Extensions beyond the ISO C standard are now marked. |

41115 The *setvbuf*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard. |

41116 **NAME**

41117 shm_open — open a shared memory object (**REALTIME**)

41118 **SYNOPSIS**

41119 SHM
```
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);
```
41120

41121

41122 **DESCRIPTION**

41123 The *shm_open*( ) function establishes a connection between a shared memory object and a file
41124 descriptor. It creates an open file description that refers to the shared memory object and a file
41125 descriptor that refers to that open file description. The file descriptor is used by other functions
41126 to refer to that shared memory object. The *name* argument points to a string naming a shared
41127 memory object. It is unspecified whether the name appears in the file system and is visible to
41128 other functions that take path names as arguments. The *name* argument conforms to the
41129 construction rules for a path name. If *name* begins with the slash character, then processes calling
41130 *shm_open*( ) with the same value of *name* refer to the same shared memory object, as long as that
41131 name has not been removed. If *name* does not begin with the slash character, the effect is |
41132 implementation-defined. The interpretation of slash characters other than the leading slash |
41133 character in *name* is implementation-defined. |

41134 If successful, *shm_open*( ) shall return a file descriptor for the shared memory object that is the
41135 lowest numbered file descriptor not currently open for that process. The open file description is
41136 new, and therefore the file descriptor does not share it with any other processes. It is unspecified
41137 whether the file offset is set. The FD_CLOEXEC file descriptor flag associated with the new file
41138 descriptor is set.

41139 The file status flags and file access modes of the open file description are according to the value
41140 of *oflag*. The *oflag* argument is the bitwise-inclusive OR of the following flags defined in the
41141 header **<fcntl.h>**. Applications specify exactly one of the first two values (access modes) below
41142 in the value of *oflag*:

41143 O_RDONLY Open for read access only.

41144 O_RDWR Open for read or write access.

41145 Any combination of the remaining flags may be specified in the value of *oflag*:

41146 O_CREAT If the shared memory object exists, this flag has no effect, except as noted
41147 under O_EXCL below. Otherwise, the shared memory object is created; the
41148 user ID of the shared memory object shall be set to the effective user ID of the
41149 process; the group ID of the shared memory object is set to a system default
41150 group ID or to the effective group ID of the process. The permission bits of the
41151 shared memory object shall be set to the value of the *mode* argument except
41152 those set in the file mode creation mask of the process. When bits in *mode*
41153 other than the file permission bits are set, the effect is unspecified. The *mode*
41154 argument does not affect whether the shared memory object is opened for
41155 reading, for writing, or for both. The shared memory object has a size of zero.

41156 O_EXCL If O_EXCL and O_CREAT are set, *shm_open*( ) fails if the shared memory
41157 object exists. The check for the existence of the shared memory object and the
41158 creation of the object if it does not exist is atomic with respect to other
41159 processes executing *shm_open*( ) naming the same shared memory object with
41160 O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the
41161 result is undefined.

41162 O_TRUNC If the shared memory object exists, and it is successfully opened O_RDWR,
41163 the object shall be truncated to zero length and the mode and owner shall be
41164 unchanged by this function call. The result of using O_TRUNC with
41165 O_RDONLY is undefined.

41166 When a shared memory object is created, the state of the shared memory object, including all
41167 data associated with the shared memory object, persists until the shared memory object is
41168 unlinked and all other references are gone. It is unspecified whether the name and shared
41169 memory object state remain valid after a system reboot.

41170 **RETURN VALUE**

41171 Upon successful completion, the *shm_open*( ) function shall return a non-negative integer
41172 representing the lowest numbered unused file descriptor. Otherwise, it shall return –1 and set
41173 *errno* to indicate the error.

41174 **ERRORS**

41175 The *shm_open*( ) function shall fail if:

41176 [EACCES] The shared memory object exists and the permissions specified by *oflag* are |
41177 denied, or the shared memory object does not exist and permission to create
41178 the shared memory object is denied, or O_TRUNC is specified and write
41179 permission is denied.

41180 [EEXIST] O_CREAT and O_EXCL are set and the named shared memory object already |
41181 exists.

41182 [EINTR] The *shm_open*( ) operation was interrupted by a signal. |

41183 [EINVAL] The *shm_open*( ) operation is not supported for the given name. |

41184 [EMFILE] Too many file descriptors are currently in use by this process. |

41185 [ENAMETOOLONG] |
41186 The length of the *name* argument exceeds {PATH_MAX} or a path name |
41187 component is longer than {NAME_MAX}. |

41188 [ENFILE] Too many shared memory objects are currently open in the system. |

41189 [ENOENT] O_CREAT is not set and the named shared memory object does not exist. |

41190 [ENOSPC] There is insufficient space for the creation of the new shared memory object. |

41191 **EXAMPLES**

41192 None.

41193 **APPLICATION USAGE**

41194 None.

41195 **RATIONALE**

41196 When the Memory Mapped Files option is supported, the normal *open*( ) call is used to obtain a
41197 descriptor to a file to be mapped according to existing practice with *mmap*( ). When the Shared
41198 Memory Objects option is supported, the *shm_open*( ) function is used to obtain a descriptor to
41199 the shared memory object to be mapped.

41200 There is ample precedent for having a file descriptor represent several types of objects. In the
41201 POSIX.1-1990 standard, a file descriptor can represent a file, a pipe, a FIFO, a tty, or a directory.
41202 Many implementations simply have an operations vector, which is indexed by the file descriptor
41203 type and does very different operations. Note that in some cases the file descriptor passed to
41204 generic operations on file descriptors are returned by *open*( ) or *creat*( ) and in some cases
41205 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 a *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.

The set of valid open flags to *shm_open*() was restricted to O_RDONLY, O_RDWR, O_CREAT, and O_TRUNC because these could be easily implemented on most memory mapping systems. This volume of IEEE Std. 1003.1-200x is silent on the results if the implementation cannot supply the requested file access because of implementation-defined reasons, including hardware ones.

The error conditions [EACCES] and [ENOTSUP] are provided to inform the application that the implementation cannot complete a request.

[EACCES] indicates for implementation-defined reasons, probably hardware-related, that the implementation cannot comply with a requested mode because it conflicts with another requested mode. An example might be that an application desires to open a memory object two times, mapping different areas with different access modes. If the implementation cannot map a single area into a process space in two places, which would be required if different access modes were required for the two areas, then the implementation may inform the application at the time of the second open.

[ENOTSUP] indicates for implementation-defined reasons, probably hardware-related, that the implementation cannot comply with a requested mode at all. An example would be that the hardware of the implementation cannot support write-only shared memory areas.

On all implementations, it may be desirable to restrict the location of the memory objects to specific file systems for performance (such as a RAM disk) or implementation-defined reasons (shared memory supported directly only on certain file systems). The *shm_open*() function may be used to enforce these restrictions. There are a number of methods available to the application to determine an appropriate name of the file or the location of an appropriate directory. One way is from the environment via *getenv*(). Another would be from a configuration file.

This volume of IEEE Std. 1003.1-200x specifies that memory objects have initial contents of zero when created. This is consistent with current behavior for both files and newly allocated memory. For those implementations that use physical memory, it would be possible that such implementations could simply use available memory and give it to the process uninitialized. This, however, is not consistent with standard behavior for the uninitialized data area, the stack, and of course, files. Finally, it is highly desirable to set the allocated memory to zero for security reasons. Thus, initializing memory objects to zero is required.

41250 **FUTURE DIRECTIONS**

41251 None.

41252 **SEE ALSO**

41253 *close*( ), *dup*( ), *exec*, *fcntl*( ), *mmap*( ), *shmat*( ), *shmctl*( ), *shmdt*( ), *shm_unlink*( ), *umask*( ), the Base |
41254 Definitions volume of IEEE Std. 1003.1-200x, **<fcntl.h>**, **<sys/mman.h>** |

41255 **CHANGE HISTORY**

41256 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

41257 **Issue 6**

41258 The *shm_open*( ) function is marked as part of the Shared Memory Objects option. |

41259 The [ENOSYS] error condition has been removed as stubs need not be provided if an
41260 implementation does not support the Shared Memory Objects option. |

41261 **NAME**

41262 shm_unlink — remove a shared memory object (**REALTIME**)

41263 **SYNOPSIS**

41264 SHM
```
#include <sys/mman.h>
```

41265
```
int shm_unlink(const char * name);
```

41266

41267 **DESCRIPTION**

41268 The *shm_unlink*( ) function removes the name of the shared memory object named by the string
41269 pointed to by *name*.

41270 If one or more references to the shared memory object exist when the object is unlinked, the
41271 name is removed before *shm_unlink*( ) returns, but the removal of the memory object contents is
41272 postponed until all open and map references to the shared memory object have been removed.

41273 Even if the object continues to exist after the last *shm_unlink*( ), reuse of the name shall cause a
41274 new shared memory object to be created.

41275 **RETURN VALUE**

41276 Upon successful completion, a value of zero shall be returned. Otherwise, a value of –1 shall be
41277 returned and *errno* set to indicate the error. If –1 is returned, the named shared memory object
41278 shall not be changed by this function call.

41279 **ERRORS**

41280 The *shm_unlink*( ) function shall fail if:

41281 [EACCES] Permission is denied to unlink the named shared memory object. |

41282 [ENAMETOOLONG] |
41283 The length of the *name* argument exceeds {PATH_MAX} or a path name |
41284 component is longer than {NAME_MAX}. |

41285 [ENOENT] The named shared memory object does not exist. |

41286 **EXAMPLES**

41287 None.

41288 **APPLICATION USAGE**

41289 Names of memory objects that were allocated with *open*( ) are deleted with *unlink*( ) in the usual
41290 fashion. Names of memory objects that were allocated with *shm_open*( ) are deleted with
41291 *shm_unlink*( ). Note that the actual memory object is not destroyed until the last close and
41292 unmap on it have occurred if it was already in use.

41293 **RATIONALE**

41294 None.

41295 **FUTURE DIRECTIONS**

41296 None.

41297 **SEE ALSO**

41298 *close*( ), *mmap*( ), *munmap*( ), *shmat*( ), *shmctl*( ), *shmdt*( ), *shm_open*( ), the Base Definitions volume |
41299 of IEEE Std. 1003.1-200x, **<sys/mman.h>** |

41300 **CHANGE HISTORY**

41301 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.

41308 **NAME**

41309 shmat — XSI shared memory attach operation

41310 **SYNOPSIS**

41311 XSI
```
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```
41312

41313

41314 **DESCRIPTION**

41315 The *shmat*( ) function operates on XSI shared memory (see the Base Definitions volume of | 41316 IEEE Std. 1003.1-200x, Section 3.342, Shared Memory Object). It is unspecified whether this | 41317 function interoperates with the realtime interprocess communication facilities defined in Section 41318 2.8 (on page 543).

41319 The *shmat*( ) function attaches the shared memory segment associated with the shared memory 41320 identifier specified by *shmid* to the address space of the calling process. The segment is attached 41321 at the address specified by one of the following criteria:

41322 • If *shmaddr* is a null pointer, the segment is attached at the first available address as selected 41323 by the system.

41324 • If *shmaddr* is not a null pointer and (*shmflg* &SHM_RND) is non-zero, the segment is attached 41325 at the address given by (*shmaddr* –((*uintptr_t*)*shmaddr* %SHMLBA)). The character '%' is the 41326 C-language remainder operator.

41327 • If *shmaddr* is not a null pointer and (*shmflg* &SHM_RND) is 0, the segment is attached at the 41328 address given by *shmaddr*.

41329 • The segment is attached for reading if (*shmflg* &SHM_RDONLY) is non-zero and the calling 41330 process has read permission; otherwise, if it is 0 and the calling process has read and write 41331 permission, the segment is attached for reading and writing.

41332 **RETURN VALUE**

41333 Upon successful completion, *shmat*( ) shall increment the value of *shm_nattch* in the data 41334 structure associated with the shared memory ID of the attached shared memory segment and 41335 return the segment's start address.

41336 Otherwise, the shared memory segment shall not be attached, *shmat*( ) shall return –1, and *errno* 41337 shall be set to indicate the error.

41338 **ERRORS**

41339 The *shmat*( ) function shall fail if:

41340 [EACCES] Operation permission is denied to the calling process; see Section 2.7 (on page | 41341 541).

41342 [EINVAL] The value of *shmid* is not a valid shared memory identifier, the *shmaddr* is not a | 41343 null pointer, and the value of (*shmaddr* –((*uintptr_t*)*shmaddr* %SHMLBA)) is an 41344 illegal address for attaching shared memory; or the *shmaddr* is not a null 41345 pointer, (*shmflg* &SHM_RND) is 0, and the value of *shmaddr* is an illegal 41346 address for attaching shared memory.

41347 [EMFILE] The number of shared memory segments attached to the calling process | 41348 would exceed the system-imposed limit.

41349 [ENOMEM] The available data space is not large enough to accommodate the shared | 41350 memory segment.

41351 **EXAMPLES**

41352 None.

41353 **APPLICATION USAGE**

41354 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
41355 Application developers who need to use IPC should design their applications so that modules
41356 using the IPC routines described in Section 2.7 (on page 541) can be easily modified to use the
41357 alternative interfaces.

41358 **RATIONALE**

41359 None.

41360 **FUTURE DIRECTIONS**

41361 None.

41362 **SEE ALSO**

41363 *exec*, *exit*( ), *fork*( ), *shmctl*( ), *shmdt*( ), *shmget*( ), *shm_open*( ), *shm_unlink*( ), the Base Definitions |
41364 volume of IEEE Std. 1003.1-200x, **<sys/shm.h>**, Section 2.7 (on page 541) |

41365 **CHANGE HISTORY**

41366 First released in Issue 2. Derived from Issue 2 of the SVID. |

41367 **Issue 4**

41368 The function is no longer marked as OPTIONAL FUNCTIONALITY.

41369 The **<sys/types.h>** and **<sys/ipc.h>** headers are removed from the SYNOPSIS section.

41370 The type of argument *shmaddr* is changed from **char*** to **const void***.

41371 The [ENOSYS] error is removed from the ERRORS section.

41372 The DESCRIPTION is clarified in several places.

41373 A FUTURE DIRECTIONS section is added warning application developers about migration to
41374 IEEE 1003.4 interfaces for interprocess communication.

41375 **Issue 5**

41376 Moved from SHARED MEMORY to BASE.

41377 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
41378 DIRECTIONS to a new APPLICATION USAGE section.

41379 **Issue 6**

41380 The Open Group corrigenda item U021/13 has been applied.

41381 **NAME**

41382 shmctl — XSI shared memory control operations

41383 **SYNOPSIS**

41384 XSI
```
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```
41385

41386

41387 **DESCRIPTION**

41388 The *shmctl*( ) function operates on XSI shared memory (see the Base Definitions volume of |
41389 IEEE Std. 1003.1-200x, Section 3.342, Shared Memory Object). It is unspecified whether this |
41390 function interoperates with the realtime interprocess communication facilities defined in Section
41391 2.8 (on page 543).

41392 The *shmctl*( ) function provides a variety of shared memory control operations as specified by
41393 *cmd*. The following values for *cmd* are available:

41394 IPC_STAT Place the current value of each member of the **shmid_ds** data structure
41395 associated with *shmid* into the structure pointed to by *buf*. The contents of the
41396 structure are defined in **<sys/shm.h>**.

41397 IPC_SET Set the value of the following members of the **shmid_ds** data structure
41398 associated with *shmid* to the corresponding value found in the structure
41399 pointed to by *buf*:

41400 `shm_perm.uid`
41401 `shm_perm.gid`
41402 `shm_perm.mode` Low-order nine bits.

41403 IPC_SET can only be executed by a process that has an effective user ID equal
41404 to either that of a process with appropriate privileges or to the value of
41405 *shm_perm.cuid* or *shm_perm.uid* in the **shmid_ds** data structure associated with
41406 *shmid*.

41407 IPC_RMID Remove the shared memory identifier specified by *shmid* from the system and
41408 destroy the shared memory segment and **shmid_ds** data structure associated
41409 with it. IPC_RMID can only be executed by a process that has an effective user
41410 ID equal to either that of a process with appropriate privileges or to the value
41411 of *shm_perm.cuid* or *shm_perm.uid* in the **shmid_ds** data structure associated
41412 with *shmid*.

41413 **RETURN VALUE**

41414 Upon successful completion, *shmctl*( ) shall return 0; otherwise, it shall return –1 and set *errno* to
41415 indicate the error.

41416 **ERRORS**

41417 The *shmctl*( ) function shall fail if:

41418 [EACCES] The argument *cmd* is equal to IPC_STAT and the calling process does not have |
41419 read permission; see Section 2.7 (on page 541).

41420 [EINVAL] The value of *shmid* is not a valid shared memory identifier, or the value of *cmd* |
41421 is not a valid command.

41422 [EPERM] The argument *cmd* is equal to IPC_RMID or IPC_SET and the effective user ID |
41423 of the calling process is not equal to that of a process with appropriate
41424 privileges and it is not equal to the value of *shm_perm.cuid* or *shm_perm.uid* in
41425 the data structure associated with *shmid*.

The *shmctl*( ) function may fail if:

[EOVERFLOW] The *cmd* argument is IPC_STAT and the *gid* or *uid* value is too large to be stored in the structure pointed to by the *buf* argument.

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 (on page 541) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

shmat( ), *shmdt*( ), *shmget*( ), *shm_open*( ), *shm_unlink*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/shm.h>**, Section 2.7 (on page 541)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 4

The function is no longer marked as OPTIONAL FUNCTIONALITY.

The **<sys/types.h>** and **<sys/ipc.h>** headers are removed from the SYNOPSIS section.

The [ENOSYS] error is removed from the ERRORS section.

A FUTURE DIRECTIONS section is added warning application developers about migration to IEEE 1003.4 interfaces for interprocess communication.

Issue 4, Version 2

The ERRORS section is updated for X/OPEN UNIX conformance to include [EOVERFLOW] as an optional error.

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.

41458 **NAME**

41459 shmdt — XSI shared memory detach operation

41460 **SYNOPSIS**

41461 XSI
```
#include <sys/shm.h>

int shmdt(const void *shmaddr);
```
41462

41463

41464 **DESCRIPTION**

41465 The *shmdt*( ) function operates on XSI shared memory (see the Base Definitions volume of |
41466 IEEE Std. 1003.1-200x, Section 3.342, Shared Memory Object). It is unspecified whether this |
41467 function interoperates with the realtime interprocess communication facilities defined in Section
41468 2.8 (on page 543).

41469 The *shmdt*( ) function detaches the shared memory segment located at the address specified by
41470 *shmaddr* from the address space of the calling process.

41471 **RETURN VALUE**

41472 Upon successful completion, *shmdt*( ) shall decrement the value of *shm_nattch* in the data
41473 structure associated with the shared memory ID of the attached shared memory segment and
41474 return 0.

41475 Otherwise, the shared memory segment shall not be detached, *shmdt*( ) shall return –1, and *errno*
41476 shall be set to indicate the error.

41477 **ERRORS**

41478 The *shmdt*( ) function shall fail if:

41479 [EINVAL] The value of *shmaddr* is not the data segment start address of a shared |
41480 memory segment.

41481 **EXAMPLES**

41482 None.

41483 **APPLICATION USAGE**

41484 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
41485 Application developers who need to use IPC should design their applications so that modules
41486 using the IPC routines described in Section 2.7 (on page 541) can be easily modified to use the
41487 alternative interfaces.

41488 **RATIONALE**

41489 None.

41490 **FUTURE DIRECTIONS**

41491 None.

41492 **SEE ALSO**

41493 *exec*, *exit*( ), *fork*( ), *shmat*( ), *shmctl*( ), *shmget*( ), *shm_open*( ), *shm_unlink*( ), the Base Definitions |
41494 volume of IEEE Std. 1003.1-200x, **<sys/shm.h>**, Section 2.7 (on page 541) |

41495 **CHANGE HISTORY**

41496 First released in Issue 2. Derived from Issue 2 of the SVID. |

41497 **Issue 4**

41498 The function is no longer marked as OPTIONAL FUNCTIONALITY.

41499 The **<sys/types.h>** and **<sys/ipc.h>** headers are removed from the SYNOPSIS section.

41500 The type of argument *shmaddr* is changed from **char*** to **const void***.

The DESCRIPTION is clarified in several places.

The [ENOSYS] error is removed from the ERRORS section.

A FUTURE DIRECTIONS section is added warning application developers about migration to IEEE 1003.4 interfaces for interprocess communication.

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.

41509 **NAME**

41510 shmget — get XSI shared memory segment

41511 **SYNOPSIS**

41512 XSI
```
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```
41513

41514

41515 **DESCRIPTION**

41516 The *shmget*( ) function operates on XSI shared memory (see the Base Definitions volume of |
41517 IEEE Std. 1003.1-200x, Section 3.342, Shared Memory Object). It is unspecified whether this |
41518 function interoperates with the realtime interprocess communication facilities defined in Section
41519 2.8 (on page 543).

41520 The *shmget*( ) function shall return the shared memory identifier associated with *key*.

41521 A shared memory identifier, associated data structure, and shared memory segment of at least
41522 *size* bytes (see **<sys/shm.h>**) are created for *key* if one of the following is true:

41523 • The argument *key* is equal to IPC_PRIVATE.

41524 • The argument *key* does not already have a shared memory identifier associated with it and
41525 (*shmflg* &IPC_CREAT) is non-zero.

41526 Upon creation, the data structure associated with the new shared memory identifier shall be
41527 initialized as follows:

41528 • The values of *shm_perm.cuid*, *shm_perm.uid*, *shm_perm.cgid*, and *shm_perm.gid* are set equal to
41529 the effective user ID and effective group ID, respectively, of the calling process.

41530 • The low-order nine bits of *shm_perm.mode* are set equal to the low-order nine bits of *shmflg*.
41531 The value of *shm_segsz* is set equal to the value of *size*.

41532 • The values of *shm_lpid*, *shm_nattch*, *shm_atime*, and *shm_dtime* are set equal to 0.

41533 • The value of *shm_ctime* is set equal to the current time.

41534 When the shared memory segment is created, it shall be initialized with all zero values.

41535 **RETURN VALUE**

41536 Upon successful completion, *shmget*( ) shall return a non-negative integer, namely a shared
41537 memory identifier; otherwise, it shall return –1 and set *errno* to indicate the error.

41538 **ERRORS**

41539 The *shmget*( ) function shall fail if:

41540 [EACCES] A shared memory identifier exists for *key* but operation permission as |
41541 specified by the low-order nine bits of *shmflg* would not be granted; see
41542 Section 2.7 (on page 541).

41543 [EEXIST] A shared memory identifier exists for the argument *key* but (*shmflg* |
41544 &IPC_CREAT) &&(*shmflg* &IPC_EXCL) is non-zero.

41545 [EINVAL] The value of *size* is less than the system-imposed minimum or greater than the |
41546 system-imposed maximum, or a shared memory identifier exists for the
41547 argument *key* but the size of the segment associated with it is less than *size* and
41548 *size* is not 0.

41549 [ENOENT] A shared memory identifier does not exist for the argument *key* and (*shmflg* |
41550 &IPC_CREAT) is 0.

[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.

[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.

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 (on page 541) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

shmat( ), *shmctl*( ), *shmdt*( ), *shm_open*( ), *shm_unlink*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/shm.h>**, Section 2.7 (on page 541)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 4

The function is no longer marked as OPTIONAL FUNCTIONALITY.

The **<sys/types.h>** and **<sys/ipc.h>** headers are removed from the SYNOPSIS section.

The [ENOSYS] error is removed from the ERRORS section.

A FUTURE DIRECTIONS section is added warning application developers about migration to IEEE 1003.4 interfaces for interprocess communication.

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.

41583 **NAME**

41584 shutdown — shut down socket send and receive operations

41585 **SYNOPSIS**

41586 `#include <sys/socket.h>`

41587 `int shutdown(int socket, int how);`

41588 **DESCRIPTION**

41589 The *shutdown*( ) function shall cause all or part of a full-duplex connection on the socket
41590 associated with the file descriptor *socket* to be shut down.

41591 The *shutdown*( ) function takes the following arguments:

41592 *socket* Specifies the file descriptor of the socket.

41593 *how* Specifies the type of shutdown. The values are as follows:

41594 SHUT_RD Disables further receive operations.

41595 SHUT_WR Disables further send operations.

41596 SHUT_RDWR Disables further send and receive operations.

41597 The *shutdown*( ) function disables subsequent send and/or receive operations on a socket,
41598 depending on the value of the *how* argument.

41599 **RETURN VALUE**

41600 Upon successful completion, *shutdown*( ) shall return 0; otherwise, –1 shall be returned and *errno*
41601 set to indicate the error.

41602 **ERRORS**

41603 The *shutdown*( ) function shall fail if:

41604 [EBADF] The *socket* argument is not a valid file descriptor.

41605 [EINVAL] The *how* argument is invalid.

41606 [ENOTCONN] The socket is not connected.

41607 [ENOTSOCK] The *socket* argument does not refer to a socket.

41608 The *shutdown*( ) function may fail if:

41609 [ENOBUFS] Insufficient resources were available in the system to perform the operation. |

41610 **EXAMPLES**

41611 None.

41612 **APPLICATION USAGE**

41613 None.

41614 **RATIONALE**

41615 None.

41616 **FUTURE DIRECTIONS**

41617 None.

41618 **SEE ALSO**

41619 *getsockopt*( ), *read*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *select*( ), *send*( ), *sendto*( ), *setsockopt*( ), *socket*( ),
41620 *write*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/socket.h>** |

41621 **CHANGE HISTORY**

41622 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

41623 **NAME**

41624 sigaction — examine and change signal action

41625 **SYNOPSIS**

41626 `#include <signal.h>`

41627 `int sigaction(int sig, const struct sigaction *restrict act,` |
41628 `struct sigaction *restrict oact);` |

41629 **DESCRIPTION** |

41630 The *sigaction*( ) function allows the calling process to examine and/or specify the action to be
41631 associated with a specific signal. The argument *sig* specifies the signal; acceptable values are
41632 defined in **<signal.h>**.

41633 The structure **sigaction**, used to describe an action to be taken, is defined in the header
41634 **<signal.h>** to include at least the following members:

41635

41636

Member Type	Member Name	Description
void(*) (int)	*sa_handler*	SIG_DFL, SIG_IGN, or pointer to a function.
sigset_t	*sa_mask*	Additional set of signals to be blocked during execution of signal-catching function.
int	*sa_flags*	Special flags to affect behavior of signal.
void(*) (int, siginfo_t *, void *)	*sa_sigaction*	Signal-catching function.

41644 If the argument *act* is not a null pointer, it points to a structure specifying the action to be
41645 associated with the specified signal. If the argument *oact* is not a null pointer, the action
41646 previously associated with the signal is stored in the location pointed to by the argument *oact*. If
41647 the argument *act* is a null pointer, signal handling is unchanged; thus, the call can be used to
41648 enquire about the current handling of a given signal. The SIGKILL and SIGSTOP signals shall
41649 not be added to the signal mask using this mechanism; this restriction shall be enforced by the
41650 system without causing an error to be indicated.

41651 If the SA_SIGINFO flag (see below) is cleared in the *sa_flags* field of the **sigaction** structure, the
41652 *sa_handler* field identifies the action to be associated with the specified signal. If the
41653 SA_SIGINFO flag is set in the *sa_flags* field, and the implementation supports the Realtime
41654 Signals Extension option or the X/Open System Interfaces Extension option, the *sa_sigaction*
41655 field specifies a signal-catching function. If the SA_SIGINFO bit is cleared and the *sa_handler*
41656 field specifies a signal-catching function, or if the SA_SIGINFO bit is set, the *sa_mask* field
41657 identifies a set of signals that shall be added to the signal mask of the thread before the signal-
41658 catching function is invoked. If the *sa_handler* field specifies a signal-catching function, the
41659 *sa_mask* field identifies a set of signals that shall be added to the process' signal mask before the
41660 signal-catching function is invoked.

41661 The *sa_flags* field can be used to modify the behavior of the specified signal.

41662 The following flags, defined in the header **<signal.h>**, can be set in *sa_flags*:

41663 SA_NOCLDSTOP Do not generate SIGCHLD when children stop. |

41664 If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is not set in *sa_flags*, and |
41665 the implementation supports the SIGCHLD signal, then a SIGCHLD |
41666 signal shall be generated for the calling process whenever any of its child |
41667 processes stop. If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is set in |
41668 *sa_flags*, then the implementation shall not generate a SIGCHLD signal in |

this way.

XSI SA_ONSTACK If set and an alternate signal stack has been declared with *sigaltstack*() or *sigstack*(), the signal shall be delivered to the calling process on that stack. Otherwise, the signal shall be delivered on the current stack.

XSI 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.

Note: SIGILL and SIGTRAP cannot be automatically reset when delivered; the system silently enforces this restriction.

Otherwise, the disposition of the signal shall not be modified on entry to the signal handler.

In addition, if this flag is set, *sigaction*() behaves as if the SA_NODEFER flag were also set.

XSI SA_RESTART This flag affects the behavior of interruptible functions; that is, those specified to fail with *errno* 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 the flag is not set, interruptible functions interrupted by this signal shall fail with *errno* set to [EINTR].

SA_SIGINFO If cleared and the signal is caught, the signal-catching function shall be entered as:

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

where *signo* is the only argument to the signal catching function. In this case, the application shall use the *sa_handler* member to describe the signal catching function and the application shall not modify the *sa_sigaction* member.

XSI | RTS If SA_SIGINFO is set and the signal is caught, the signal-catching function shall be entered as:

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

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 process' context that was interrupted when the signal was delivered. In this case, the application shall use the *sa_sigaction* member to describe the signal catching function and the application shall not modify the *sa_handler* member.

The *si_signo* member contains the system-generated signal number.

XSI The *si_errno* 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.

The *si_code* member contains a code identifying the cause of the signal. If the value of *si_code* is less than or equal to 0, then the signal was generated by a process and *si_pid* and *si_uid*, respectively, indicate the process ID and the real user ID of the sender. The **<signal.h>** header description contains information about the signal specific contents of the

elements of the **siginfo_t** type.

XSI SA_NOCLDWAIT If set, and *sig* equals SIGCHLD, 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]. Otherwise, terminating child processes shall be transformed into zombie processes, unless SIGCHLD is set to SIG_IGN.

XSI SA_NODEFER If set and *sig* is caught, *sig* shall not be added to the process' signal mask on entry to the signal handler unless it is included in *sa_mask*. Otherwise, *sig* shall always be added to the process' signal mask on entry to the signal handler.

When a signal is caught by a signal-catching function installed by *sigaction*(), a new signal mask is calculated and installed for the duration of the signal-catching function (or until a call to either *sigprocmask*() or *sigsuspend*() is made). This mask is formed by taking the union of the current XSI signal mask and the value of the *sa_mask* for the signal being delivered unless SA_NODEFER or SA_RESETHAND is set, and then including the signal being delivered. If and when the user's signal handler returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it remains installed until another action is XSI explicitly requested (by another call to *sigaction*()), until the SA_RESETHAND flag causes resetting of the handler, or until one of the *exec* functions is called.

If the previous action for *sig* had been established by *signal*(), the values of the fields returned in the structure pointed to by *oact* are unspecified, and in particular *oact->sa_handler* is not necessarily the same value passed to *signal*(). However, if a pointer to the same structure or a copy thereof is passed to a subsequent call to *sigaction*() via the *act* argument, handling of the signal shall be as if the original call to *signal*() were repeated.

If *sigaction*() fails, no new signal handler is installed.

It is unspecified whether an attempt to set the action for a signal that cannot be caught or ignored to SIG_DFL is ignored or causes an error to be returned with *errno* set to [EINVAL].

If SA_SIGINFO is not set in *sa_flags*, then the disposition of subsequent occurrences of *sig* when it is already pending is implementation-defined; the signal-catching function shall be invoked RTS with a single argument. If the implementation supports the Realtime Signals Extension option, and if SA_SIGINFO is set in *sa_flags*, then subsequent occurrences of *sig* generated by *sigqueue*() or as a result of any signal-generating function that supports the specification of an application-defined value (when *sig* is already pending) shall be queued in FIFO order until delivered or accepted; the signal-catching function shall be invoked with three arguments. The application specified value is passed to the signal-catching function as the *si_value* member of the **siginfo_t** structure.

The result of the use of *sigaction*() and a *sigwait*() function concurrently within a process on the same signal is unspecified.

RETURN VALUE

Upon successful completion, *sigaction*() shall return 0; otherwise, –1 shall be returned, *errno* shall be set to indicate the error, and no new signal-catching function shall be installed.

41758 **ERRORS**

41759 The *sigaction*( ) function shall fail if:

41760 [EINVAL] The *sig* argument is not a valid signal number or an attempt is made to catch a |
41761 signal that cannot be caught or ignore a signal that cannot be ignored.

41762 [ENOTSUP] The SA_SIGINFO bit flag is set in the *sa_flags* field of the **sigaction** structure,
41763 and the implementation does not support either the Realtime Signals
41764 Extension option, or the X/Open System Interfaces Extension option.

41765 The *sigaction*( ) function may fail if:

41766 [EINVAL] An attempt was made to set the action to SIG_DFL for a signal that cannot be |
41767 caught or ignored (or both).

41768 **EXAMPLES**

41769 None.

41770 **APPLICATION USAGE**

41771 The *sigaction*( ) function supersedes the *signal*( ) function, and should be used in preference. In
41772 particular, *sigaction*( ) and *signal*( ) should not be used in the same process to control the same
41773 signal. The behavior of reentrant functions, as defined in the DESCRIPTION, is as specified by
41774 this volume of IEEE Std. 1003.1-200x, regardless of invocation from a signal-catching function.
41775 This is the only intended meaning of the statement that reentrant functions may be used in
41776 signal-catching functions without restrictions. Applications must still consider all effects of such
41777 functions on such things as data structures, files, and process state. In particular, application
41778 writers need to consider the restrictions on interactions when interrupting *sleep*( ) and
41779 interactions among multiple handles for a file description. The fact that any specific function is
41780 listed as reentrant does not necessarily mean that invocation of that function from a signal-
41781 catching function is recommended.

41782 In order to prevent errors arising from interrupting non-reentrant function calls, applications
41783 should protect calls to these functions either by blocking the appropriate signals or through the
41784 use of some programmatic semaphore (see *semget*( ), *sem_init*( ), *sem_open*( ), and so on). Note in
41785 particular that even the ‘‘safe’’ functions may modify *errno*; the signal-catching function, if not
41786 executing as an independent thread, may want to save and restore its value. Naturally, the same
41787 principles apply to the reentrancy of application routines and asynchronous data access. Note
41788 that *longjmp*( ) and *siglongjmp*( ) are not in the list of reentrant functions. This is because the code
41789 executing after *longjmp*( ) and *siglongjmp*( ) can call any unsafe functions with the same danger as
41790 calling those unsafe functions directly from the signal handler. Applications that use *longjmp*( )
41791 and *siglongjmp*( ) from within signal handlers require rigorous protection in order to be portable.
41792 Many of the other functions that are excluded from the list are traditionally implemented using
41793 either *malloc*( ) or *free*( ) functions or the standard I/O library, both of which traditionally use
41794 data structures in a non-reentrant manner. Because any combination of different functions using
41795 a common data structure can cause reentrancy problems, this volume of IEEE Std. 1003.1-200x
41796 does not define the behavior when any unsafe function is called in a signal handler that
41797 interrupts an unsafe function.

41798 If the signal occurs other than as the result of calling *abort*( ), *kill*( ), or *raise*( ), the behavior is
41799 undefined if the signal handler calls any function in the standard library other than one of the
41800 functions listed in the table above or refers to any object with static storage duration other than
41801 by assigning a value to a static storage duration variable of type **volatile sig_atomic_t**.
41802 Furthermore, if such a call fails, the value of *errno* is indeterminate.

41803 Usually, the signal is executed on the stack that was in effect before the signal was delivered. An
41804 alternate stack may be specified to receive a subset of the signals being caught.

When the signal handler returns, the receiving process 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 by the process.

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 portable applications that catch SIGCHLD are expected to install signal-catching functions that repeatedly call the *waitpid*( ) function with the WNOHANG flag set, acting on each child for which status is returned, until *waitpid*( ) returns zero. If stopped children are not of interest, the use of the SA_NOCLDSTOP flag can prevent the overhead from invoking the signal-catching routine when they stop.

Some historical implementations also define other mechanisms for stopping processes, such as the *ptrace*( ) function. These implementations usually do not generate a SIGCHLD signal when processes stop due to this mechanism; however, that is beyond the scope of this volume of IEEE Std. 1003.1-200x.

This volume of IEEE Std. 1003.1-200x requires that calls to *sigaction*( ) that supply a NULL *act* argument succeed, even in the case of signals that cannot be caught or ignored (that is, SIGKILL or SIGSTOP). The System V *signal*( ) and BSD *sigvec*( ) functions return [EINVAL] in these cases and, in this respect, their behavior varies from *sigaction*( ).

This volume of IEEE Std. 1003.1-200x requires that *sigaction*( ) properly save and restore a signal action set up by the ISO C standard *signal*( ) function. However, there is no guarantee that the reverse is true, nor could there be given the greater amount of information conveyed by the **sigaction** structure. Because of this, applications should avoid using both functions for the same signal in the same process. Since this cannot always be avoided in case of general-purpose library routines, they should always be implemented with *sigaction*( ).

It was intended that the *signal*( ) function should be implementable as a library routine using *sigaction*( ).

The POSIX Realtime Extension extends the *sigaction*( ) function as specified by the POSIX.1-1990 standard to allow the application to request on a per-signal basis via an additional signal action

flag that the extra parameters, including the application-defined signal value, if any, be passed to the signal-catching function.

FUTURE DIRECTIONS

The *fpathconf*( ) function is marked as an extension in the list of safe functions because it is not included in the corresponding list in the ISO POSIX-1 standard, but it is expected to be added in a future version.

SEE ALSO

Section 2.4 (on page 528), *bsd_signal*( ), *kill*( ), *_longjmp*( ), *longjmp*( ), *raise*( ), *semget*( ), *sem_init*( ), *sem_open*( ), *sigaddset*( ), *sigaltstack*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ), *signal*( ), *sigprocmask*( ), *sigsuspend*( ), *wait*( ), *waitid*( ), *waitpid*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<signal.h>**, **<ucontext.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The *raise*( ) and *signal*( ) functions are added to the list of functions that are either reentrant or not interruptible by signals; *fpathconf*( ) is also added to this list and marked as an extension; *ustat*( ) is removed from the list, as this function is withdrawn from the interface definition. It is no longer specified whether *abort*( ), *exit*( ), and *longjmp*( ) also fall into this category of functions.

The APPLICATION USAGE section is added. Most of this text is moved from the DESCRIPTION in Issue 3.

The FUTURE DIRECTIONS section is added.

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument *act* is changed from **struct sigaction*** to **const struct sigaction***.
- A statement is added to the DESCRIPTION indicating that the consequence of attempting to set SIG_DFL for a signal that cannot be caught or ignored is unspecified. The [EINVAL] error, describing one possible reaction to this condition, is added to the ERRORS section.

Issue 4, Version 2

The following changes are incorporated for X/OPEN UNIX conformance:

- The DESCRIPTION describes *sa_sigaction*, the member of the **sigaction** structure that is the signal-catching function.
- The DESCRIPTION describes the SA_ONSTACK, SA_RESETHAND, SA_RESTART, SA_SIGINFO, SA_NOCLDWAIT, and SA_NODEFER settings of *sa_flags*. The text describes the implications of the use of SA_SIGINFO for the number of arguments passed to the signal-catching function. The text also describes the effects of the SA_NODEFER and SA_RESETHAND flags on the delivery of a signal and on the permanence of an installed action.
- The DESCRIPTION specifies the effect if the action for the SIGCHLD signal is set to SIG_IGN.
- In the DESCRIPTION, additional text describes the effect if the action is a pointer to a function. A new bullet covers the case where SA_SIGINFO is set. SIGBUS is given as an additional signal for which the behavior of a process is undefined following a normal return from the signal-catching function.

- The APPLICATION USAGE section is updated to describe use of an alternate signal stack; resumption of the process receiving the signal; coding for compatibility with POSIX.4-1993; and implementation of signal-handling functions in BSD.

Issue 5

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

In the DESCRIPTION, the second argument to *func* when SA_SIGINFO is set is no longer permitted to be NULL, and the description of permitted **siginfo_t** contents is expanded by reference to **<signal.h>**.

Because the X/OPEN UNIX Extension functionality is now folded into the BASE, the [ENOTSUP] error is deleted.

Issue 6

The Open Group corrigenda item U028/7 has been applied. In the paragraph entitled ‘‘Signal Effects on Other Functions’’, a reference to *sigpending*( ) is added.

In the DESCRIPTION, the text ‘‘Signal Generation and Delivery’’ is moved to a separate section of this volume of IEEE Std. 1003.1-200x.

Text describing functionality from the Realtime Signals option is marked.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENOTSUP] error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements. |

The **restrict** keyword is added to the *sigaction*( ) prototype for alignment with the ISO/IEC 9899: 1999 standard. | |

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

41918 **NAME**

41919 sigaddset — add a signal to a signal set

41920 **SYNOPSIS**

41921 `#include <signal.h>`

41922 `int sigaddset(sigset_t *set, int signo);`

41923 **DESCRIPTION**

41924 The *sigaddset*( ) function adds the individual signal specified by the *signo* to the signal set pointed
41925 to by *set*.

41926 Applications shall call either *sigemptyset*( ) or *sigfillset*( ) at least once for each object of type
41927 **sigset_t** prior to any other use of that object. If such an object is not initialized in this way, but is
41928 nonetheless supplied as an argument to any of *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigismember*( ),
41929 *sigpending*( ), or *sigprocmask*( ), the results are undefined.

41930 **RETURN VALUE**

41931 Upon successful completion, *sigaddset*( ) shall return 0; otherwise, it shall return –1 and set *errno*
41932 to indicate the error.

41933 **ERRORS**

41934 The *sigaddset*( ) function may fail if:

41935 [EINVAL] The value of the *signo* argument is an invalid or unsupported signal number. |

41936 **EXAMPLES**

41937 None.

41938 **APPLICATION USAGE**

41939 None.

41940 **RATIONALE**

41941 None.

41942 **FUTURE DIRECTIONS**

41943 None.

41944 **SEE ALSO**

41945 Section 2.4 (on page 528), *sigaction*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ),
41946 *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
41947 **<signal.h>**

41948 **CHANGE HISTORY**

41949 First released in Issue 3.

41950 Entry included for alignment with the POSIX.1-1988 standard.

41951 **Issue 4**

41952 The word ‘‘will’’ is replaced by the word ‘‘may’’ in the ERRORS section.

41953 **Issue 5**

41954 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
41955 previous issues.

41956 **Issue 6**

41957 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

41958 **NAME**

41959 sigaltstack — set and get signal alternate stack context

41960 **SYNOPSIS**

41961 XSI
```
#include <signal.h>

int sigaltstack(const stack_t *restrict ss, stack_t *restrict oss);
```
41962 |
41963 |

41964 **DESCRIPTION**

41965 The *sigaltstack*( ) function allows a process to define and examine the state of an alternate stack
41966 for signal handlers. Signals that have been explicitly declared to execute on the alternate stack
41967 shall be delivered on the alternate stack.

41968 If *ss* is not a null pointer, it points to a **stack_t** structure that specifies the alternate signal stack
41969 that shall take effect upon return from *sigaltstack*( ). The *ss_flags* member specifies the new stack
41970 state. If it is set to SS_DISABLE, the stack is disabled and *ss_sp* and *ss_size* are ignored.
41971 Otherwise, the stack shall be enabled, and the *ss_sp* and *ss_size* members specify the new address
41972 and size of the stack.

41973 The range of addresses starting at *ss_sp* up to but not including *ss_sp*+*ss_size*, is available to the
41974 implementation for use as the stack. This function makes no assumptions regarding which end
41975 is the stack base and in which direction the stack grows as items are pushed.

41976 If *oss* is not a null pointer, on successful completion it shall point to a **stack_t** structure that
41977 specifies the alternate signal stack that was in effect prior to the call to *sigaltstack*( ). The *ss_sp*
41978 and *ss_size* members specify the address and size of that stack. The *ss_flags* member specifies the
41979 stack's state, and may contain one of the following values:

41980 SS_ONSTACK The process is currently executing on the alternate signal stack. Attempts to
41981 modify the alternate signal stack while the process is executing on it fail. This |
41982 flag shall not be modified by processes. |

41983 SS_DISABLE The alternate signal stack is currently disabled.

41984 The value SIGSTKSZ is a system default specifying the number of bytes that would be used to
41985 cover the usual case when manually allocating an alternate stack area. The value MINSIGSTKSZ
41986 is defined to be the minimum stack size for a signal handler. In computing an alternate stack
41987 size, a program should add that amount to its stack requirements to allow for the system
41988 implementation overhead. The constants SS_ONSTACK, SS_DISABLE, SIGSTKSZ, and
41989 MINSIGSTKSZ are defined in **<signal.h>**.

41990 After a successful call to one of the *exec* functions, there are no alternate signal stacks in the new
41991 process image.

41992 In some implementations, a signal (whether or not indicated to execute on the alternate stack)
41993 shall always execute on the alternate stack if it is delivered while another signal is being caught
41994 using the alternate stack.

41995 Use of this function by library threads that are not bound to kernel-scheduled entities results in
41996 undefined behavior.

41997 **RETURN VALUE**

41998 Upon successful completion, *sigaltstack*( ) shall return 0; otherwise, it shall return –1 and set *errno*
41999 to indicate the error.

42000 **ERRORS**

42001 The *sigaltstack*( ) function shall fail if:

42002 [EINVAL] The *ss* argument is not a null pointer, and the *ss_flags* member pointed to by *ss* |
42003 contains flags other than SS_DISABLE.

42004 [ENOMEM] The size of the alternate stack area is less than MINSIGSTKSZ. |

42005 [EPERM] An attempt was made to modify an active stack. |

42006 **EXAMPLES**

42007 **Allocating Memory for an Alternate Stack**

42008 The following example illustrates a method for allocating memory for an alternate stack.

```
42009 #include <signal.h>
42010 ...
42011 if ((sigstk.ss_sp = malloc(SIGSTKSZ)) == NULL)
42012     /* Error return. */
42013 sigstk.ss_size = SIGSTKSZ;
42014 sigstk.ss_flags = 0;
42015 if (sigaltstack(&sigstk,(stack_t *)0) < 0)
42016     perror("sigaltstack");
```

42017 **APPLICATION USAGE**

42018 On some implementations, stack space is automatically extended as needed. On those
42019 implementations, automatic extension is typically not available for an alternate stack. If the stack
42020 overflows, the behavior is undefined.

42021 **RATIONALE**

42022 None.

42023 **FUTURE DIRECTIONS**

42024 None.

42025 **SEE ALSO**

42026 Section 2.4 (on page 528), *sigaction*( ), *sigsetjmp*( ), the Base Definitions volume of |
42027 IEEE Std. 1003.1-200x, **<signal.h>** |

42028 **CHANGE HISTORY**

42029 First released in Issue 4, Version 2.

42030 **Issue 5**

42031 Moved from X/OPEN UNIX extension to BASE.

42032 The last sentence of the DESCRIPTION was included as an APPLICATION USAGE note in
42033 previous issues.

42034 **Issue 6**

42035 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements. |

42036 The **restrict** keyword is added to the *sigaltstack*( ) prototype for alignment with the |
42037 ISO/IEC 9899:1999 standard. |

42038 **NAME**

42039 sigdelset — delete a signal from a signal set

42040 **SYNOPSIS**

42041 `#include <signal.h>`

42042 `int sigdelset(sigset_t *set, int signo);`

42043 **DESCRIPTION**

42044 The *sigdelset*( ) function deletes the individual signal specified by *signo* from the signal set
42045 pointed to by *set*.

42046 Applications should call either *sigemptyset*( ) or *sigfillset*( ) at least once for each object of type
42047 **sigset_t** prior to any other use of that object. If such an object is not initialized in this way, but is
42048 nonetheless supplied as an argument to any of *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigismember*( ),
42049 *sigpending*( ), or *sigprocmask*( ), the results are undefined.

42050 **RETURN VALUE**

42051 Upon successful completion, *sigdelset*( ) shall return 0; otherwise, it shall return –1 and set *errno*
42052 to indicate the error.

42053 **ERRORS**

42054 The *sigdelset*( ) function may fail if:

42055 [EINVAL] The *signo* argument is not a valid signal number, or is an unsupported signal |
42056 number.

42057 **EXAMPLES**

42058 None.

42059 **APPLICATION USAGE**

42060 None.

42061 **RATIONALE**

42062 None.

42063 **FUTURE DIRECTIONS**

42064 None.

42065 **SEE ALSO**

42066 Section 2.4 (on page 528), *sigaction*( ), *sigaddset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ),
42067 *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
42068 **<signal.h>**

42069 **CHANGE HISTORY**

42070 First released in Issue 3.

42071 Entry included for alignment with the POSIX.1-1988 standard.

42072 **Issue 4**

42073 The word ‘‘will’’ is replaced by the word ‘‘may’’ in the ERRORS section.

42074 **Issue 5**

42075 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
42076 previous issues.

NAME

sigemptyset — initialize and empty a signal set

SYNOPSIS

```
#include <signal.h>

int sigemptyset(sigset_t *set);
```

DESCRIPTION

The *sigemptyset*( ) function initializes the signal set pointed to by *set*, such that all signals defined in this volume of IEEE Std. 1003.1-200x are excluded.

RETURN VALUE

Upon successful completion, *sigemptyset*( ) shall return 0; otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The implementation of the *sigemptyset*( ) (or *sigfillset*( )) function could quite trivially clear (or set) all the bits in the signal set. Alternatively, it would be reasonable to initialize part of the structure, such as a version field, to permit binary-compatibility between releases where the size of the set varies. For such reasons, either *sigemptyset*( ) or *sigfillset*( ) must be called prior to any other use of the signal set, even if such use is read-only (for example, as an argument to *sigpending*( )). This function is not intended for dynamic allocation.

The *sigfillset*( ) and *sigemptyset*( ) functions require that the resulting signal set include (or exclude) all the signals defined in this volume of IEEE Std. 1003.1-200x. Although it is outside the scope of this volume of IEEE Std. 1003.1-200x to place this requirement on signals that are implemented as extensions, it is recommended that implementation-defined signals also be affected by these functions. However, there may be a good reason for a particular signal not to be affected. For example, blocking or ignoring an implementation-defined signal may have undesirable side effects, whereas the default action for that signal is harmless. In such a case, it would be preferable for such a signal to be excluded from the signal set returned by *sigfillset*( ).

In early proposals there was no distinction between invalid and unsupported signals (the names of optional signals that were not supported by an implementation were not defined by that implementation). The [EINVAL] error was thus specified as a required error for invalid signals. With that distinction, it is not necessary to require implementations of these functions to determine whether an optional signal is actually supported, as that could have a significant performance impact for little value. The error could have been required for invalid signals and optional for unsupported signals, but this seemed unnecessarily complex. Thus, the error is optional in both cases.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 528), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigfillset*( ), *sigismember*( ), *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<signal.h>**

42123 **CHANGE HISTORY**

42124 First released in Issue 3.

42125 Entry included for alignment with the POSIX.1-1988 standard.

42126 **NAME**

42127 sigfillset — initialize and fill a signal set

42128 **SYNOPSIS**

42129 `#include <signal.h>`

42130 `int sigfillset(sigset_t *set);`

42131 **DESCRIPTION**

42132 The *sigfillset*( ) function initializes the signal set pointed to by *set*, such that all signals defined in
42133 this volume of IEEE Std. 1003.1-200x are included.

42134 **RETURN VALUE**

42135 Upon successful completion, *sigfillset*( ) shall return 0; otherwise, it shall return –1 and set *errno*
42136 to indicate the error.

42137 **ERRORS**

42138 No errors are defined.

42139 **EXAMPLES**

42140 None.

42141 **APPLICATION USAGE**

42142 None.

42143 **RATIONALE**

42144 Refer to *sigemptyset*( ) (on page 1863).

42145 **FUTURE DIRECTIONS**

42146 None.

42147 **SEE ALSO**

42148 Section 2.4 (on page 528), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigemptyset*( ), *sigismember*( ),
42149 *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std. 1003.1-200x,
42150 **<signal.h>**

42151 **CHANGE HISTORY**

42152 First released in Issue 3.

42153 Entry included for alignment with the POSIX.1-1988 standard.

42154 **NAME**

42155 sighold, sigignore — add a signal to the signal mask or set a signal disposition to be ignored

42156 **SYNOPSIS**

42157 XSI
```
#include <signal.h>

int sighold(int sig);
int sigignore(int sig);
```
42158

42159

42160

42161 **DESCRIPTION**

42162 Refer to *signal*( ).

NAME

siginterrupt — allow signals to interrupt functions

SYNOPSIS

XSI
```
#include <signal.h>

int siginterrupt(int sig, int flag);
```

DESCRIPTION

The *siginterrupt*( ) function is used to change the restart behavior when a function is interrupted by the specified signal. The function *siginterrupt*(*sig*, *flag*) has an effect as if implemented as:

```
siginterrupt(int sig, int flag) {
    int ret;
    struct sigaction act;

    (void) sigaction(sig, NULL, &act);
    if (flag)
        act.sa_flags &= ~SA_RESTART;
    else
        act.sa_flags |= SA_RESTART;
    ret = sigaction(sig, &act, NULL);
    return ret;
}
```

RETURN VALUE

Upon successful completion, *siginterrupt*( ) shall return 0; otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *siginterrupt*( ) function shall fail if:

[EINVAL] The *sig* argument is not a valid signal number. |

EXAMPLES

None.

APPLICATION USAGE

The *siginterrupt*( ) function supports programs written to historical system interfaces. A portable application, when being written or rewritten, should use *sigaction*( ) with the SA_RESTART flag instead of *siginterrupt*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 528), *sigaction*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<signal.h>** |

CHANGE HISTORY

First released in Issue 4, Version 2.

42204 **Issue 5**

42205 Moved from X/OPEN UNIX extension to BASE.

NAME

sigismember — test for a signal in a signal set

SYNOPSIS

```
#include <signal.h>

int sigismember(const sigset_t *set, int signo);
```

DESCRIPTION

The *sigismember*( ) function tests whether the signal specified by *signo* is a member of the set pointed to by *set*.

Applications should call either *sigemptyset*( ) or *sigfillset*( ) at least once for each object of type **sigset_t** prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigismember*( ), *sigpending*( ), or *sigprocmask*( ), the results are undefined.

RETURN VALUE

Upon successful completion, *sigismember*( ) shall return 1 if the specified signal is a member of the specified set, or 0 if it is not. Otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *sigismember*( ) function may fail if:

[EINVAL] The *signo* argument is not a valid signal number, or is an unsupported signal number. |

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 528), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigfillset*( ), *sigemptyset*( ), *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<signal.h>** |

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The following changes are incorporated for alignment with the ISO C standard:

- The type of the argument *set* is changed from **sigset_t*** to type **const sigset_t***.
- The word ‘‘will’’ is replaced by the word ‘‘may’’ in the ERRORS section.

Issue 5

The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

42247 **NAME**

42248 siglongjmp — non-local goto with signal handling

42249 **SYNOPSIS**

42250 `#include <setjmp.h>`

42251 `void siglongjmp(sigjmp_buf env, int val);`

42252 **DESCRIPTION**

42253 The *siglongjmp*( ) function restores the environment saved by the most recent invocation of
42254 *sigsetjmp*( ) in the same thread, with the corresponding **sigjmp_buf** argument. If there is no such
42255 invocation, or if the function containing the invocation of *sigsetjmp*( ) has terminated execution in
42256 the interim, the behavior is undefined.

42257 All accessible objects have values as of the time *siglongjmp*( ) was called, except that the values of |
42258 objects of automatic storage duration which are local to the function containing the invocation of
42259 the corresponding *sigsetjmp*( ) which do not have volatile-qualified type and which are changed
42260 between the *sigsetjmp*( ) invocation and *siglongjmp*( ) call are indeterminate.

42261 As it bypasses the usual function call and return mechanisms, *siglongjmp*( ) shall execute
42262 correctly in contexts of interrupts, signals, and any of their associated functions. However, if
42263 *siglongjmp*( ) is invoked from a nested signal handler (that is, from a function invoked as a result
42264 of a signal raised during the handling of another signal), the behavior is undefined.

42265 The *siglongjmp*( ) function shall restore the saved signal mask if and only if the *env* argument was
42266 initialized by a call to *sigsetjmp*( ) with a non-zero *savemask* argument.

42267 The effect of a call to *siglongjmp*( ) where initialization of the **jmp_buf** structure was not
42268 performed in the calling thread is undefined.

42269 **RETURN VALUE**

42270 After *siglongjmp*( ) is completed, program execution shall continue as if the corresponding
42271 invocation of *sigsetjmp*( ) had just returned the value specified by *val*. The *siglongjmp*( ) function
42272 shall not cause *sigsetjmp*( ) to return 0; if *val* is 0, *sigsetjmp*( ) shall return the value 1.

42273 **ERRORS**

42274 No errors are defined.

42275 **EXAMPLES**

42276 None.

42277 **APPLICATION USAGE**

42278 The distinction between *setjmp*( ) or *longjmp*( ) and *sigsetjmp*( ) or *siglongjmp*( ) is only significant
42279 for programs which use *sigaction*( ), *sigprocmask*( ), or *sigsuspend*( ).

42280 **RATIONALE**

42281 None.

42282 **FUTURE DIRECTIONS**

42283 None.

42284 **SEE ALSO**

42285 *longjmp*( ), *setjmp*( ), *sigprocmask*( ), *sigsetjmp*( ), *sigsuspend*( ), the Base Definitions volume of |
42286 IEEE Std. 1003.1-200x, **<setjmp.h>** |

42287 **CHANGE HISTORY**

42288 First released in Issue 3.

42289 Entry included for alignment with the ISO POSIX-1 standard.

Issue 4

The APPLICATION USAGE section is amended.

An ERRORS section is added.

Issue 5

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

42295 **NAME**

42296 sighold, sigignore, signal, sigpause, sigrelse, sigset — signal management

42297 **SYNOPSIS**

42298 `#include <signal.h>`

42299 `void (*signal(int sig, void (*func)(int)))(int);`

42300 XSI `int sighold(int sig);`

42301 `int sigignore(int sig);`

42302 `int sigpause(int sig);`

42303 `int sigrelse(int sig);`

42304 `void (*sigset(int sig, void (*disp)(int)))(int);`

42305

42306 **DESCRIPTION**

42307 CX For *signal*( ): The functionality described on this reference page is aligned with the ISO C
42308 standard. Any conflict between the requirements described here and the ISO C standard is
42309 unintentional. This volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

42310 CX Use of any of these functions is unspecified in a multi-threaded process.

42311 The *signal*( ) function chooses one of three ways in which receipt of the signal number *sig* is to be
42312 subsequently handled. If the value of *func* is SIG_DFL, default handling for that signal shall
42313 occur. If the value of *func* is SIG_IGN, the signal shall be ignored. Otherwise, the application
42314 shall ensure that *func* points to a function to be called when that signal occurs. An invocation of |
42315 such a function because of a signal, or (recursively) of any further functions called by that |
42316 invocation (other than functions in the standard library), is called a ‘‘signal handler’’. |

42317 When a signal occurs, if *func* points to a function, first the equivalent of a:

42318 `signal(sig, SIG_DFL);`

42319 is executed or an implementation-defined blocking of the signal is performed. (If the value of *sig* |
42320 is SIGILL, whether the reset to SIG_DFL occurs is implementation-defined.) Next the equivalent |
42321 of:

42322 `(*func)(sig);`

42323 is executed. The *func* function may terminate by executing a **return** statement or by calling
42324 *abort*( ), *exit*( ), or *longjmp*( ). If *func* executes a **return** statement and the value of *sig* was SIGFPE |
42325 or any other implementation-defined value corresponding to a computational exception, the |
42326 behavior is undefined. Otherwise, the program shall resume execution at the point it was |
42327 interrupted. If the signal occurs as the result of calling the *abort*( ) or *raise*( ) function, the signal |
42328 handler shall not call the *raise*( ) function. |

42329 If the signal occurs other than as the result of calling *abort*( ), *kill*( ), or *raise*( ), the behavior is |
42330 undefined if the signal handler refers to any object with static storage duration other than by |
42331 assigning a value to an object declared as volatile **sig_atomic_t**, or if the signal handler calls any |
42332 function in the standard library other than one of the functions listed in Section 2.4 (on page 528) |
42333 or refers to any object with static storage duration other than by assigning a value to a static |
42334 storage duration variable of type volatile **sig_atomic_t**. Furthermore, if such a call fails, the |
42335 value of *errno* is indeterminate.

42336 At program start-up, the equivalent of:

42337 `signal(sig, SIG_IGN);`

42338 is executed for some signals, and the equivalent of:

```
signal(sig, SIG_DFL);
```

is executed for all other signals (see *exec*).

XSI The *sighold*(), *sigignore*(), *sigpause*(), *sigrelse*(), and *sigset*() functions provide simplified signal management.

The *sigset*() function is used to modify signal dispositions. The *sig* argument specifies the signal, which may be any signal except SIGKILL and SIGSTOP. The *disp* argument specifies the signal's disposition, which may be SIG_DFL, SIG_IGN, or the address of a signal handler. If *sigset*() is used, and *disp* is the address of a signal handler, the system shall add *sig* to the calling process' signal mask before executing the signal handler; when the signal handler returns, the system shall restore the calling process' signal mask to its state prior the delivery of the signal. In addition, if *sigset*() is used, and *disp* is equal to SIG_HOLD, *sig* shall be added to the calling process' signal mask and *sig*'s disposition shall remain unchanged. If *sigset*() is used, and *disp* is not equal to SIG_HOLD, *sig* shall be removed from the calling process' signal mask.

The *sighold*() function adds *sig* to the calling process' signal mask.

The *sigrelse*() function removes *sig* from the calling process' signal mask.

The *sigignore*() function sets the disposition of *sig* to SIG_IGN.

The *sigpause*() function removes *sig* from the calling process' signal mask and suspends the calling process until a signal is received. The *sigpause*() function restores the process' signal mask to its original state before returning.

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].

RETURN VALUE

If the request can be honored, *signal*() shall return the value of *func* for the most recent call to *signal*() for the specified signal *sig*. Otherwise, SIG_ERR shall be returned and a positive value shall be stored in *errno*.

XSI Upon successful completion, *sigset*() shall return SIG_HOLD if the signal had been blocked and the signal's previous disposition if it had not been blocked. Otherwise, SIG_ERR shall be returned and *errno* set to indicate the error.

The *sigpause*() function shall suspend execution of the thread until a signal is received, whereupon it shall return –1 and set *errno* to [EINTR].

For all other functions, upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *signal*() function shall fail if:

CX [EINVAL] The *sig* argument is not a valid signal number or an attempt is made to catch a signal that cannot be caught or ignore a signal that cannot be ignored.

The *signal*() function may fail if:

CX [EINVAL] An attempt was made to set the action to SIG_DFL for a signal that cannot be caught or ignored (or both).

The *sighold*(), *sigignore*(), *sigpause*(), *sigrelse*(), and *sigset*() functions shall fail if:

42382 XSI [EINVAL] The *sig* argument is an illegal signal number.

42383 The *sigset*( ) and *sigignore*( ) functions shall fail if:

42384 XSI [EINVAL] An attempt is made to catch a signal that cannot be caught, or to ignore a
42385 signal that cannot be ignored.

42386 **EXAMPLES**

42387 None.

42388 **APPLICATION USAGE**

42389 The *sigaction*( ) function provides a more comprehensive and reliable mechanism for controlling
42390 signals; new applications should use *sigaction*( ) rather than *signal*( ).

42391 The *sighold*( ) function, in conjunction with *sigrelse*( ) or *sigpause*( ), may be used to establish
42392 critical regions of code that require the delivery of a signal to be temporarily deferred.

42393 The *sigsuspend*( ) function should be used in preference to *sigpause*( ) for broader portability.

42394 **RATIONALE**

42395 None.

42396 **FUTURE DIRECTIONS**

42397 None.

42398 **SEE ALSO**

42399 Section 2.4 (on page 528), *exec*, *pause*( ), *sigaction*( ), *sigsuspend*( ), *waitid*( ), the Base Definitions
42400 volume of IEEE Std. 1003.1-200x, **<signal.h>**

42401 **CHANGE HISTORY**

42402 First released in Issue 1. Derived from Issue 1 of the SVID.

42403 **Issue 4**

42404 The APPLICATION USAGE section is added.

42405 The following changes are incorporated for alignment with the ISO C standard:

42406 • The function is no longer marked as an extension.

42407 • The argument **int** is added to the definition of *func* in the SYNOPSIS section.

42408 • In Issue 3, this function cross-referred to *sigaction*( ). This issue provides a complete
42409 description of the function as defined in ISO C standard.

42410 **Issue 4, Version 2**

42411 The following changes are incorporated for X/OPEN UNIX conformance:

42412 • The *sighold*( ), *sigignore*( ), *sigpause*( ), *sigrelse*( ), and *sigset*( ) functions are added to the
42413 SYNOPSIS.

42414 • The DESCRIPTION is updated to describe semantics of the above functions.

42415 • Additional text is added to the RETURN VALUE section to describe possible returns from
42416 the *sigset*( ) function specifically, and all of the above functions in general.

42417 • The ERRORS section is restructured to describe possible error returns from each of the above
42418 functions individually.

42419 • The APPLICATION USAGE section is updated to describe certain programming
42420 considerations associated with the X/OPEN UNIX functions.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is updated to indicate that the *sigpause*() function restores the process' signal mask to its original state before returning.

The RETURN VALUE section is updated to indicate that the *sigpause*() function suspends execution of the process until a signal is received, whereupon it returns –1 and sets *errno* to [EINTR].

Issue 6

Extensions beyond the ISO C standard are now marked.

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.

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

42433 **NAME**

42434 signbit — test sign

42435 **SYNOPSIS**

42436 `#include <math.h>`

42437 `int signbit(real-floating x);`

42438 **DESCRIPTION**

42439 CX The functionality described on this reference page is aligned with the ISO C standard. Any
42440 conflict between the requirements described here and the ISO C standard is unintentional. This
42441 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

42442 The *signbit*( ) macro shall determine whether the sign of its argument value is negative.

42443 **RETURN VALUE**

42444 The *signbit*( ) macro shall return a non-zero value if and only if the sign of its argument value is
42445 negative.

42446 **ERRORS**

42447 No errors are defined.

42448 **EXAMPLES**

42449 None.

42450 **APPLICATION USAGE**

42451 None.

42452 **RATIONALE**

42453 None.

42454 **FUTURE DIRECTIONS**

42455 None.

42456 **SEE ALSO**

42457 *fpclassify*( ), *isfinite*( ), *isinf*( ), *isnan*( ), *isnormal*( ), the Base Definitions volume of
42458 IEEE Std. 1003.1-200x, **<math.h>**

42459 **CHANGE HISTORY**

42460 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

42461 **NAME**

42462 sigpause — remove a signal from the signal mask and suspend the thread

42463 **SYNOPSIS**

42464 XSI
```
#include <signal.h>
```

42465
```
int sigpause(int sig);
```

42466

42467 **DESCRIPTION**

42468 Refer to *signal*( ).

42469 **NAME**

42470 sigpending — examine pending signals

42471 **SYNOPSIS**

42472 `#include <signal.h>`

42473 `int sigpending(sigset_t *set);`

42474 **DESCRIPTION**

42475 The *sigpending*( ) function stores, in the location referenced by the *set* argument, the set of signals
42476 that are blocked from delivery to the calling thread and that are pending on the process or the
42477 calling thread.

42478 **RETURN VALUE**

42479 Upon successful completion, *sigpending*( ) shall return 0; otherwise, –1 shall be returned and
42480 *errno* set to indicate the error.

42481 **ERRORS**

42482 No errors are defined.

42483 **EXAMPLES**

42484 None.

42485 **APPLICATION USAGE**

42486 None.

42487 **RATIONALE**

42488 None.

42489 **FUTURE DIRECTIONS**

42490 None.

42491 **SEE ALSO**

42492 *sigaddset*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ), *sigprocmask*( ), the Base Definitions |
42493 volume of IEEE Std. 1003.1-200x, **<signal.h>** |

42494 **CHANGE HISTORY**

42495 First released in Issue 3.

42496 **Issue 5**

42497 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

42498 **NAME**

42499 sigprocmask — examine and change blocked signals

42500 **SYNOPSIS**

42501 `#include <signal.h>`

42502 `int sigprocmask(int` *`how`*`, const sigset_t *restrict` *`set`*`,` |

42503 `sigset_t *restrict` *`oset`*`);` |

42504 **DESCRIPTION** |

42505 Refer to *pthread_sigmask*( ).

42506 **NAME**

42507 sigqueue — queue a signal to a process (**REALTIME**)

42508 **SYNOPSIS**

42509 RTS
```
#include <signal.h>

int sigqueue(pid_t pid, int signo, const union sigval value);
```
42510

42511

42512 **DESCRIPTION**

42513 The *sigqueue*( ) function shall cause the signal specified by *signo* to be sent with the value
42514 specified by *value* to the process specified by *pid*. If *signo* is zero (the null signal), error checking
42515 is performed but no signal is actually sent. The null signal can be used to check the validity of
42516 *pid*.

42517 The conditions required for a process to have permission to queue a signal to another process
42518 are the same as for the *kill*( ) function.

42519 The *sigqueue*( ) function returns immediately. If SA_SIGINFO is set for *signo* and if the resources
42520 were available to queue the signal, the signal is queued and sent to the receiving process. If
42521 SA_SIGINFO is not set for *signo*, then *signo* is sent at least once to the receiving process; it is
42522 unspecified whether *value* shall be sent to the receiving process as a result of this call.

42523 If the value of *pid* causes *signo* to be generated for the sending process, and if *signo* is not blocked
42524 for the calling thread and if no other thread has *signo* unblocked or is waiting in a *sigwait*( )
42525 function for *signo*, either *signo* or at least the pending, unblocked signal shall be delivered to the
42526 calling thread before the *sigqueue*( ) function returns. Should any multiple pending signals in the
42527 range SIGRTMIN to SIGRTMAX be selected for delivery, it shall be the lowest numbered one.
42528 The selection order between realtime and non-realtime signals, or between multiple pending
42529 non-realtime signals, is unspecified.

42530 **RETURN VALUE**

42531 Upon successful completion, the specified signal shall have been queued, and the *sigqueue*( )
42532 function shall return a value of zero. Otherwise, the function shall return a value of –1 and set
42533 *errno* to indicate the error.

42534 **ERRORS**

42535 The *sigqueue*( ) function shall fail if:

42536 [EAGAIN] No resources available to queue the signal. The process has already queued |
42537 SIGQUEUE_MAX signals that are still pending at the receiver(s), or a system-
42538 wide resource limit has been exceeded.

42539 [EINVAL] The value of the *signo* argument is an invalid or unsupported signal number. |

42540 [EPERM] The process does not have the appropriate privilege to send the signal to the |
42541 receiving process.

42542 [ESRCH] The process *pid* does not exist. |

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

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 function to *kill*( ).

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 function.

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.

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.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.8.1 (on page 543), the Base Definitions volume of IEEE Std. 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Issue 6

The *sigqueue*( ) function is marked as part of the Realtime Signals Extension option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Realtime Signals Extension option.

42583 **NAME**

42584 sigrelse, sigset — remove a signal from signal mask or modify signal disposition

42585 **SYNOPSIS**

42586 XSI
```
#include <signal.h>

int sigrelse(int sig);
void (*sigset(int sig, void (*disp)(int)))(int);
```
42587

42588

42589

42590 **DESCRIPTION**

42591 Refer to *signal*( ).

NAME

sigsetjmp — set jump point for a non-local goto

SYNOPSIS

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
```

DESCRIPTION

A call to *sigsetjmp*( ) saves the calling environment in its *env* argument for later use by *siglongjmp*( ). It is unspecified whether *sigsetjmp*( ) is a macro or a function. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the name *sigsetjmp*, the behavior is undefined.

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.

All accessible objects have values as of the time *siglongjmp*( ) was called, except that the values of objects of automatic storage duration which are local to the function containing the invocation of the corresponding *sigsetjmp*( ) which do not have volatile-qualified type and which are changed between the *sigsetjmp*( ) invocation and *siglongjmp*( ) call are indeterminate.

The application shall ensure that an invocation of *sigsetjmp*( ) appears in one of the following contexts only:

- The entire controlling expression of a selection or iteration statement
- One operand of a relational or equality operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement
- The operand of a unary (`'!'`) operator with the resulting expression being the entire controlling expression of a selection or iteration
- The entire expression of an expression statement (possibly cast to **void**)

RETURN VALUE

If the return is from a successful direct invocation, *sigsetjmp*( ) shall return 0. If the return is from a call to *siglongjmp*( ), *sigsetjmp*( ) shall return a non-zero value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The distinction between *setjmp*( )/*longjmp*( ) and *sigsetjmp*( )/*siglongjmp*( ) is only significant for programs which use *sigaction*( ), *sigprocmask*( ), or *sigsuspend*( ).

RATIONALE

The ISO C standard specifies various restrictions on the usage of the *setjmp*( ) macro in order to permit implementors to recognize the name in the compiler and not implement an actual function. These same restrictions apply to the *sigsetjmp*( ) macro.

There are processors that cannot easily support these calls, but this was not considered a sufficient reason to exclude them.

4.2 BSD and 4.3 BSD systems provide functions named *_setjmp*( ) and *_longjmp*( ) that, together with *setjmp*( ) and *longjmp*( ), provide the same functionality as *sigsetjmp*( ) and *siglongjmp*( ). On those systems, *setjmp*( ) and *longjmp*( ) save and restore signal masks, while *_setjmp*( ) and

42636 *_longjmp*( ) do not. On System V, Release 3 and in corresponding issues of the SVID, *setjmp*( ) and
42637 *longjmp*( ) are explicitly defined not to save and restore signal masks. In order to permit existing
42638 practice in both cases, the relation of *setjmp*( ) and *longjmp*( ) to signal masks is not specified, and
42639 a new set of functions is defined instead.

42640 The *longjmp*( ) and *siglongjmp*( ) functions operate as in the previous issue provided the matching
42641 *setjmp*( ) or *sigsetjmp*( ) has been performed in the same thread. Non-local jumps into contexts
42642 saved by other threads would be at best a questionable practice and were not considered worthy
42643 of standardization.

42644 **FUTURE DIRECTIONS**

42645 None.

42646 **SEE ALSO**

42647 *siglongjmp*( ), *signal*( ), *sigprocmask*( ), *sigsuspend*( ), the Base Definitions volume of |
42648 IEEE Std. 1003.1-200x, **<setjmp.h>** |

42649 **CHANGE HISTORY**

42650 First released in Issue 3.

42651 Entry included for alignment with the POSIX.1-1988 standard.

42652 **Issue 4**

42653 The DESCRIPTION states that *sigsetjmp*( ) is a macro or a function. Issue 3 states that it is a
42654 macro. Warnings are also added about the suppression of a *sigsetjmp*( ) macro definition.

42655 A statement is added to the DESCRIPTION about the accessibility of objects after a *siglongjmp*( )
42656 call.

42657 Text is added to the DESCRIPTION describing the contexts in which calls to *sigsetjmp*( ) are
42658 valid.

42659 **Issue 5**

42660 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

42661 **Issue 6**

42662 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

42663 **NAME**

42664 sigsuspend — wait for a signal

42665 **SYNOPSIS**

42666 `#include <signal.h>`

42667 `int sigsuspend(const sigset_t *sigmask);`

42668 **DESCRIPTION**

42669 The *sigsuspend*( ) function replaces the current signal mask of the calling thread with the set of
42670 signals pointed to by *sigmask* and then suspends the thread until delivery of a signal whose
42671 action is either to execute a signal-catching function or to terminate the process. This shall not
42672 cause any other signals that may have been pending on the process to become pending on the
42673 thread.

42674 If the action is to terminate the process then *sigsuspend*( ) shall never return. If the action is to
42675 execute a signal-catching function, then *sigsuspend*( ) shall return after the signal-catching
42676 function returns, with the signal mask restored to the set that existed prior to the *sigsuspend*( )
42677 call.

42678 It is not possible to block signals that cannot be ignored. This is enforced by the system without
42679 causing an error to be indicated.

42680 **RETURN VALUE**

42681 Since *sigsuspend*( ) suspends thread execution indefinitely, there is no successful completion
42682 return value. If a return occurs, –1 shall be returned and *errno* set to indicate the error.

42683 **ERRORS**

42684 The *sigsuspend*( ) function shall fail if:

42685 [EINTR] A signal is caught by the calling process and control is returned from the
42686 signal-catching function.

42687 **EXAMPLES**

42688 None.

42689 **APPLICATION USAGE**

42690 Normally, at the beginning of a critical code section, a specified set of signals is blocked using
42691 the *sigprocmask*( ) function. When the thread has completed the critical section and needs to wait
42692 for the previously blocked signal(s), it pauses by calling *sigsuspend*( ) with the mask that was
42693 returned by the *sigprocmask*( ) call.

42694 **RATIONALE**

42695 None.

42696 **FUTURE DIRECTIONS**

42697 None.

42698 **SEE ALSO**

42699 Section 2.4 (on page 528), *pause*( ), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), the
42700 Base Definitions volume of IEEE Std. 1003.1-200x, **<signal.h>**

42701 **CHANGE HISTORY**

42702 First released in Issue 3.

42703 Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The term ‘‘signal handler’’ is changed to ‘‘signal-catching function’’.

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The type of the argument *sigmask* is changed from **sigset_t*** to **const sigset_t***.

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.

NAME

sigtimedwait, sigwaitinfo — wait for queued signals (**REALTIME**)

SYNOPSIS

RTS
```
#include <signal.h>

int sigtimedwait(const sigset_t *restrict set, siginfo_t *restrict info,
    const struct timespec *restrict timeout);
int sigwaitinfo(const sigset_t *restrict set, siginfo_t *restrict info);
```

DESCRIPTION

The *sigtimedwait*( ) function behaves the same as *sigwaitinfo*( ) except that if none of the signals specified by *set* are pending, *sigtimedwait*( ) waits for the time interval specified in the **timespec** structure referenced by *timeout*. If the **timespec** structure pointed to by *timeout* is zero-valued and if none of the signals specified by *set* are pending, then *sigtimedwait*( ) returns immediately with an error. If *timeout* is the NULL pointer, the behavior is unspecified. If the Monotonic Clock option is supported, the CLOCK_MONOTONIC clock shall be used to measure the time interval specified by the *timeout* argument.

MON

The *sigwaitinfo*( ) function selects the pending signal from the set specified by *set*. Should any of multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it shall be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified. If no signal in *set* is pending at the time of the call, the calling thread is suspended until one or more signals in *set* become pending or until it is interrupted by an unblocked, caught signal.

The *sigwaitinfo*( ) function behaves the same as the *sigwait*( ) function if the *info* argument is NULL. If the *info* argument is non-NULL, the *sigwaitinfo*( ) function behaves the same as *sigwait*( ), except that the selected signal number shall be stored in the *si_signo* member, and the cause of the signal shall be stored in the *si_code* member. If any value is queued to the selected signal, the first such queued value shall be dequeued and, if the *info* argument is non-NULL, the value shall be stored in the *si_value* member of *info*. The system resource used to queue the signal shall be released and returned to the system for other use. If no value is queued, the content of the *si_value* member is undefined. If no further signals are queued for the selected signal, the pending indication for that signal shall be reset.

RETURN VALUE

Upon successful completion (that is, one of the signals specified by *set* is pending or is generated) *sigwaitinfo*( ) and *sigtimedwait*( ) shall return the selected signal number. Otherwise, the function shall return a value of –1 and set *errno* to indicate the error.

ERRORS

The *sigtimedwait*( ) function shall fail if:

Notes to Reviewers

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

D1, XSH, ERN 345 proposes that the [EAGAIN] error condition ought to be [ETIMEDOUT] as per the same condition on *pthread_cond_timedwait*( ).

[EAGAIN] No signal specified by *set* was generated within the specified timeout period.

The *sigtimedwait*( ) and *sigwaitinfo*( ) functions may fail if:

[EINTR] The wait was interrupted by an unblocked, caught signal. It shall be documented in system documentation whether this error causes these

functions to fail.

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 only checks for this error if no signal is pending in *set* and it is necessary to wait.

EXAMPLES

None.

APPLICATION USAGE

None.

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 cancelation 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*.

The *sigwait* functions provide a synchronous mechanism for threads to wait for asynchronously generated signals. One important question was how many threads that are suspended in a call to a *sigwait*() function for a signal should return from the call when the signal is sent. Four choices were considered:

1. Return an error for multiple simultaneous calls to *sigwait* functions for the same signal.
2. One or more threads return.
3. All waiting threads return.
4. Exactly one thread returns.

Prohibiting multiple calls to *sigwait*() for the same signal was felt to be overly restrictive. The ''one or more'' behavior made implementation of conforming packages easy at the expense of forcing POSIX threads clients to protect against multiple simultaneous calls to *sigwait*() in application code in order to achieve predictable behavior. There was concern that the ''all waiting threads'' behavior would result in ''signal broadcast storms'', consuming excessive CPU resources by replicating the signals in the general case. Furthermore, no convincing examples could be presented that delivery to all was either simpler or more powerful than delivery to one.

Thus, the consensus was that exactly one thread that was suspended in a call to a *sigwait* function for a signal should return when that signal occurs. This is not an onerous restriction as:

- A multi-way signal wait can be built from the single-way wait.
- Signals should only be handled by application-level code, as library routines cannot guess what the application wants to do with signals generated for the entire process.
- Applications can thus arrange for a single thread to wait for any given signal and call any needed routines upon its arrival.

In an application that is using signals for XSI interprocess communication, signal processing is typically done in one place. Alternatively, if the signal is being caught so that process cleanup can be done, the signal handler thread can call separate process cleanup routines for each portion of the application. Since the application main line started each portion of the application, it is at the right abstraction level to tell each portion of the application to clean up.

Certainly, there exist programming styles where it is logical to consider waiting for a single signal in multiple threads. A simple *sigwait_multiple*() routine can be constructed to achieve this goal. A possible implementation would be to have each *sigwait_multiple*() caller registered as having expressed interest in a set of signals. The caller then waits on a thread-specific condition variable. A single server thread calls a *sigwait*() function on the union of all registered signals. When the *sigwait*() function returns, the appropriate state is set and condition variables are broadcast. New *sigwait_multiple*() callers may cause the pending *sigwait*() call to be canceled and reissued in order to update the set of signals being waited for.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.8.1 (on page 543), *pause*(), *pthread_sigmask*(), *sigaction*(), *sigpending*(), *sigsuspend*(), *sigwait*(), the Base Definitions volume of IEEE Std. 1003.1-200x, **<signal.h>**, **<time.h>**

42847 **CHANGE HISTORY**

42848 First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the
42849 POSIX Threads Extension.

42850 **Issue 6**

42851 These functions are marked as part of the Realtime Signals Extension option. |

42852 The Open Group corrigenda item U035/3 has been applied. The SYNOPSIS of the *sigwaitinfo*()
42853 function has been corrected so that the second argument is of type **siginfo_t***.

42854 The [ENOSYS] error condition has been removed as stubs need not be provided if an
42855 implementation does not support the Realtime Signals Extension option. |

42856 The DESCRIPTION is updated for alignment with IEEE Std. 1003.1j-2000 by specifying that the
42857 CLOCK_MONOTONIC clock, if supported, is used to measure timeout intervals. |

42858 The **restrict** keyword is added to the *sigtimedwait*() and *sigwaitinfo*() prototypes for alignment |
42859 with the ISO/IEC 9899:1999 standard. |

42860 **NAME**

42861 sigwait — wait for queued signals

42862 **SYNOPSIS**

42863 `#include <signal.h>`

42864 `int sigwait(const sigset_t *restrict set, int *restrict sig);`

42865 **DESCRIPTION**

42866 The *sigwait*( ) function selects a pending signal from *set*, atomically clears it from the system's set
42867 of pending signals, and returns that signal number in the location referenced by *sig*. If prior to
42868 the call to *sigwait*( ) there are multiple pending instances of a single signal number, it is
42869 implementation-defined whether upon successful return there are any remaining pending
42870 RTS signals for that signal number. If the implementation supports queued signals and there are
42871 multiple signals queued for the signal number selected, the first such queued signal shall cause a
42872 return from *sigwait*( ) and the remainder shall remain queued. If no signal in *set* is pending at the
42873 time of the call, the thread is suspended until one or more becomes pending. The signals defined
42874 by *set* shall have been blocked at the time of the call to *sigwait*( ); otherwise, the behavior is
42875 undefined. The effect of *sigwait*( ) on the signal actions for the signals in *set* is unspecified.

42876 If more than one thread is using *sigwait*( ) to wait for the same signal, no more than one of these
42877 threads shall return from *sigwait*( ) with the signal number. Which thread returns from *sigwait*( )
42878 if more than a single thread is waiting is unspecified.

42879 RTS Should any of the multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it
42880 shall be the lowest numbered one. The selection order between realtime and non-realtime
42881 signals, or between multiple pending non-realtime signals, is unspecified.

42882 **RETURN VALUE**

42883 Upon successful completion, *sigwait*( ) shall store the signal number of the received signal at the
42884 location referenced by *sig* and return zero. Otherwise, an error number shall be returned to
42885 indicate the error.

42886 **ERRORS**

42887 The *sigwait*( ) function may fail if:

42888 [EINVAL] The *set* argument contains an invalid or unsupported signal number.

42889 **EXAMPLES**

42890 None.

42891 **APPLICATION USAGE**

42892 None.

42893 **RATIONALE**

42894 To provide a convenient way for a thread to wait for a signal, this volume of
42895 IEEE Std. 1003.1-200x provides the *sigwait*( ) function. For most cases where a thread has to wait
42896 for a signal, the *sigwait*( ) function should be quite convenient, efficient, and adequate.

42897 However, requests were made for a lower-level primitive than *sigwait*( ) and for semaphores that
42898 could be used by threads. After some consideration, threads were allowed to use semaphores
42899 and *sem_post*( ) was defined to be async-signal and async-cancel-safe.

42900 In summary, when it is necessary for code run in response to an asynchronous signal to notify a
42901 thread, *sigwait*( ) should be used to handle the signal. Alternatively, if the implementation
42902 provides semaphores, they also can be used, either following *sigwait*( ) or from within a signal
42903 handling routine previously registered with *sigaction*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 528), Section 2.8.1 (on page 543), *pause*( ), *pthread_sigmask*( ), *sigaction*( ), *sigpending*( ), *sigsuspend*( ), *sigwaitinfo*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<signal.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Issue 6

The RATIONALE section is added.

The **restrict** keyword is added to the *sigwait*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

42917 **NAME**

42918 sigwaitinfo — wait for queued signals (**REALTIME**)

42919 **SYNOPSIS**

42920 RTS
```
#include <signal.h>

int sigwaitinfo(const sigset_t *restrict set, siginfo_t *restrict info);
```

42923 **DESCRIPTION**

42924 Refer to *sigtimedwait*( ).

NAME

sin, sinf, sinl — sine function

SYNOPSIS

```
#include <math.h>

double sin(double x);
float sinf(float x);
long double sinl(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 sine of its argument *x*, measured in radians.

An application wishing to check for error situations should set *errno* to 0 before calling *sin*(). If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

The *sin*() function may lose accuracy when its argument is far from 0.0.

RETURN VALUE

Upon successful completion, these functions shall return the sine of *x*.

XSI If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

XSI If *x* is ±Inf, either 0.0 shall be returned and *errno* set to [EDOM], or NaN shall be returned and *errno* may be set to [EDOM].

If the correct result would cause underflow, 0.0 shall be returned and *errno* may be set to [ERANGE].

ERRORS

These functions may fail if:

XSI [EDOM] The value of *x* is NaN, or *x* is ±Inf.

[ERANGE] The result underflows.

XSI No other errors shall occur.

EXAMPLES

Taking the Sine of a 45-Degree Angle

```
#include <math.h>
...
double radians = 45.0 * M_PI / 180;
double result;
...
result = sin(radians);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asin( ), *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 4

References to *matherr*( ) are removed.

The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the ISO C standard and to rationalize error handling in the mathematics functions.

The return value specified for [EDOM] is marked as an extension.

Issue 5

The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes in previous issues.

Issue 6

The *sinf*( ) and *sinl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

42980 **NAME**

42981 sinh, sinhf, sinhl — hyperbolic sine function

42982 **SYNOPSIS**

42983
```
#include <math.h>
```

42984
```
double sinh(double x);
```
42985
```
float sinhf(float x);
```
42986
```
long double sinhl(long double x);
```

42987 **DESCRIPTION**

42988 CX The functionality described on this reference page is aligned with the ISO C standard. Any
42989 conflict between the requirements described here and the ISO C standard is unintentional. This
42990 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

42991 These functions shall compute the hyperbolic sine of *x*.

42992 An application wishing to check for error situations should set *errno* to 0 before calling *sinh*(). If
42993 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

42994 **RETURN VALUE**

42995 Upon successful completion, these functions shall return the hyperbolic sine of *x*.

42996 If the result would cause an overflow, ±HUGE_VAL shall be returned and *errno* set to
42997 [ERANGE].

42998 If the result would cause underflow, 0.0 shall be returned and *errno* may be set to [ERANGE].

42999 XSI If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

43000 **ERRORS**

43001 These functions shall fail if:

43002 [ERANGE] The result would cause overflow.

43003 These functions may fail if:

43004 XSI [EDOM] The value of *x* is NaN.

43005 [ERANGE] The result would cause underflow.

43006 XSI No other errors shall occur.

43007 **EXAMPLES**

43008 None.

43009 **APPLICATION USAGE**

43010 None.

43011 **RATIONALE**

43012 None.

43013 **FUTURE DIRECTIONS**

43014 None.

43015 **SEE ALSO**

43016 *asinh*(), *cosh*(), *isnan*(), *tanh*(), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

43017 **CHANGE HISTORY**

43018 First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 4

References to *matherr*( ) are removed.

The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the ISO C standard and to rationalize error handling in the mathematics functions.

The return value specified for [EDOM] is marked as an extension.

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 *sinhf*( ) and *sinhl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

43029 **NAME**

43030 sleep — suspend execution for an interval of time

43031 **SYNOPSIS**

43032 `#include <unistd.h>`

43033 `unsigned sleep(unsigned seconds);`

43034 **DESCRIPTION**

43035 The *sleep*( ) function shall cause the calling thread to be suspended from execution until either
43036 the number of realtime seconds specified by the argument *seconds* has elapsed or a signal is
43037 delivered to the calling thread and its action is to invoke a signal-catching function or to
43038 terminate the process. The suspension time may be longer than requested due to the scheduling
43039 of other activity by the system.

43040 If a SIGALRM signal is generated for the calling process during execution of *sleep*( ) and if the
43041 SIGALRM signal is being ignored or blocked from delivery, it is unspecified whether *sleep*( )
43042 returns when the SIGALRM signal is scheduled. If the signal is being blocked, it is also
43043 unspecified whether it remains pending after *sleep*( ) returns or it is discarded.

43044 If a SIGALRM signal is generated for the calling process during execution of *sleep*( ), except as a
43045 result of a prior call to *alarm*( ), and if the SIGALRM signal is not being ignored or blocked from
43046 delivery, it is unspecified whether that signal has any effect other than causing *sleep*( ) to return.

43047 If a signal-catching function interrupts *sleep*( ) and examines or changes either the time a
43048 SIGALRM is scheduled to be generated, the action associated with the SIGALRM signal, or
43049 whether the SIGALRM signal is blocked from delivery, the results are unspecified.

43050 If a signal-catching function interrupts *sleep*( ) and calls *siglongjmp*( ) or *longjmp*( ) to restore an
43051 environment saved prior to the *sleep*( ) call, the action associated with the SIGALRM signal and
43052 the time at which a SIGALRM signal is scheduled to be generated are unspecified. It is also
43053 unspecified whether the SIGALRM signal is blocked, unless the process' signal mask is restored
43054 as part of the environment.

43055 XSI Interactions between *sleep*( ) and any of *setitimer*( ), *ualarm*( ), or *usleep*( ) are unspecified.

43056 **RETURN VALUE**

43057 If *sleep*( ) returns because the requested time has elapsed, the value returned shall be 0. If *sleep*( )
43058 returns because of premature arousal due to delivery of a signal, the return value shall be the
43059 ''unslept'' amount (the requested time minus the time actually slept) in seconds.

43060 **ERRORS**

43061 No errors are defined.

43062 **EXAMPLES**

43063 None.

43064 **APPLICATION USAGE**

43065 None.

43066 **RATIONALE**

43067 There are two general approaches to the implementation of the *sleep*( ) function. One is to use the
43068 *alarm*( ) function to schedule a SIGALRM signal and then suspend the process waiting for that
43069 signal. The other is to implement an independent facility. This volume of IEEE Std. 1003.1-200x
43070 permits either approach.

43071 In order to comply with the requirement that no primitive shall change a process attribute unless
43072 explicitly described by this volume of IEEE Std. 1003.1-200x, an implementation using SIGALRM
43073 must carefully take into account any SIGALRM signal scheduled by previous *alarm*( ) calls, the

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*( ) can avoid these problems.

Despite all reasonable care, there are several very subtle, but detectable and unavoidable, differences between the two types of implementations. These are the cases mentioned in this volume of IEEE Std. 1003.1-200x where some other activity relating to SIGALRM takes place, and the results are stated to be unspecified. All of these cases are sufficiently unusual as not to be of concern to most applications.

See also the discussion of the term *realtime* in *alarm*( ).

Because *sleep*( ) can be implemented using *alarm*( ), the discussion about alarms occurring early under *alarm*( ) applies to *sleep*( ) as well.

Application writers should note that the type of the argument *seconds* and the return value of *sleep*( ) is **unsigned**. That means that a Strictly Conforming POSIX System Interfaces Application cannot pass a value greater than the minimum guaranteed value for {UINT_MAX}, which the ISO C standard sets as 65 535, and any application passing a larger value is restricting its portability. A different type was considered, but historical implementations, including those with a 16-bit **int** type, consistently use either **unsigned** or **int**.

Scheduling delays may cause the process to return from the *sleep*( ) function significantly after the requested time. In such cases, the return value should be set to zero, since the formula (requested time minus the time actually spent) yields a negative number and *sleep*( ) returns an **unsigned**.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *getitimer*( ), *nanosleep*( ), *pause*( ), *sigaction*( ), *sigsetjmp*( ), *ualarm*( ), *usleep*( ), 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 4

The **<unistd.h>** header is added to the SYNOPSIS section.

Issue 4, Version 2

The DESCRIPTION is updated to indicate possible interactions with the *setitimer*( ), *ualarm*( ), and *usleep*( ) functions.

43117 **Issue 5**

43118 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

43119 **NAME**

43120 socket — create an endpoint for communication

43121 **SYNOPSIS**

43122 `#include <sys/socket.h>`

43123 `int socket(int` *`domain`*`, int` *`type`*`, int` *`protocol`*`);`

43124 **DESCRIPTION**

43125 The *socket*( ) function shall create an unbound socket in a communications domain, and return a
43126 file descriptor that can be used in later function calls that operate on sockets.

43127 The *socket*( ) function takes the following arguments:

43128 *domain* Specifies the communications domain in which a socket is to be created.

43129 *type* Specifies the type of socket to be created.

43130 *protocol* Specifies a particular protocol to be used with the socket. Specifying a *protocol*
43131 of 0 causes *socket*( ) to use an unspecified default protocol appropriate for the
43132 requested socket type.

43133 The *domain* argument specifies the address family used in the communications domain. The
43134 address families supported by the system are implementation-defined.

43135 Symbolic constants that can be used for the domain argument are defined in the **<sys/socket.h>**
43136 header.

43137 The *type* argument specifies the socket type, which determines the semantics of communication
43138 over the socket. The socket types supported by the system are implementation-defined. Possible
43139 socket types include:

43140 SOCK_STREAM Provides sequenced, reliable, bidirectional, connection-mode byte
43141 streams, and may provide a transmission mechanism for out-of-band
43142 data.

43143 SOCK_DGRAM Provides datagrams, which are connectionless-mode, unreliable messages
43144 of fixed maximum length.

43145 SOCK_SEQPACKET Provides sequenced, reliable, bidirectional, connection-mode
43146 transmission path for records. A record can be sent using one or more
43147 output operations and received using one or more input operations, but a
43148 single operation never transfers part of more than one record. Record
43149 boundaries are visible to the receiver via the MSG_EOR flag.

43150 If the *protocol* argument is non-zero, it shall specify a protocol that is supported by the address
43151 family. The protocols supported by the system are implementation-defined.

43152 The process may need to have appropriate privileges to use the *socket*( ) function or to create
43153 some sockets.

43154 **RETURN VALUE**

43155 Upon successful completion, *socket*( ) shall return a non-negative integer, the socket file
43156 descriptor. Otherwise, a value of –1 shall be returned and *errno* set to indicate the error.

43157 **ERRORS**

43158 The *socket*( ) function shall fail if:

43159 [EAFNOSUPPORT]
43160 The implementation does not support the specified address family.

[EMFILE] No more file descriptors are available for this process.

[ENFILE] No more file descriptors are available for the system.

[EPROTONOSUPPORT]
The protocol is not supported by the address family, or the protocol is not supported by the implementation.

[EPROTOTYPE] The socket type is not supported by the protocol.

The *socket*( ) function may fail if:

[EACCES] The process does not have appropriate privileges.

[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 documentation for specific address families specifies which protocols each address family supports. The documentation for specific protocols specifies which socket types each protocol supports.

The application can determine if an address family is supported by trying to create a socket with *domain* set to the protocol in question.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

accept( ), *bind*( ), *connect*( ), *getsockname*( ), *getsockopt*( ), *listen*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *send*( ), *sendmsg*( ), *setsockopt*( ), *shutdown*( ), *socketpair*( ), 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.

NAME

socketpair — create a pair of connected sockets

SYNOPSIS

```
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol,
    int socket_vector[2]);
```

DESCRIPTION

The *socketpair*( ) function creates an unbound pair of connected sockets in a specified *domain*, of a specified *type*, under the protocol optionally specified by the *protocol* argument. The two sockets are identical. The file descriptors used in referencing the created sockets are returned in *socket_vector*[0] and *socket_vector*[1].

The *socketpair*( ) function takes the following arguments:

domain Specifies the communications domain in which the sockets are to be created.

type Specifies the type of sockets to be created.

protocol Specifies a particular protocol to be used with the sockets. Specifying a *protocol* of 0 causes *socketpair*( ) to use an unspecified default protocol appropriate for the requested socket type.

socket_vector Specifies a 2-integer array to hold the file descriptors of the created socket pair.

The *type* argument specifies the socket type, which determines the semantics of communications over the socket. The socket types supported by the system are implementation-defined. Possible socket types include:

SOCK_STREAM Provides sequenced, reliable, bidirectional, connection-mode byte streams, and may provide a transmission mechanism for out-of-band data.

SOCK_DGRAM Provides datagrams, which are connectionless-mode, unreliable messages of fixed maximum length.

SOCK_SEQPACKET Provides sequenced, reliable, bidirectional, connection-mode transmission paths for records. A record can be sent using one or more output operations and received using one or more input operations, but a single operation never transfers part of more than one record. Record boundaries are visible to the receiver via the MSG_EOR flag.

If the *protocol* argument is non-zero, it shall specify a protocol that is supported by the address family. The protocols supported by the system are implementation-defined.

The process may need to have appropriate privileges to use the *socketpair*( ) function or to create some sockets.

RETURN VALUE

Upon successful completion, this function shall return 0; otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *socketpair*( ) function shall fail if:

[EAFNOSUPPORT]
The implementation does not support the specified address family.

43232 [EMFILE] No more file descriptors are available for this process.

43233 [ENFILE] No more file descriptors are available for the system.

43234 [EOPNOTSUPP] The specified protocol does not permit creation of socket pairs.

43235 [EPROTONOSUPPORT]
43236 The protocol is not supported by the address family, or the protocol is not
43237 supported by the implementation.

43238 [EPROTOTYPE] The socket type is not supported by the protocol.

43239 The *socketpair*( ) function may fail if:

43240 [EACCES] The process does not have appropriate privileges.

43241 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

43242 [ENOMEM] Insufficient memory was available to fulfill the request. |

43243 **EXAMPLES**

43244 None.

43245 **APPLICATION USAGE**

43246 The documentation for specific address families specifies which protocols each address family
43247 supports. The documentation for specific protocols specifies which socket types each protocol
43248 supports.

43249 The *socketpair*( ) function is used primarily with UNIX domain sockets and need not be
43250 supported for other domains.

43251 **RATIONALE**

43252 None.

43253 **FUTURE DIRECTIONS**

43254 None.

43255 **SEE ALSO**

43256 *socket*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/socket.h>** |

43257 **CHANGE HISTORY**

43258 First released in Issue 6. Derived from the XNS, Issue 5.2 specification. |

43259 **NAME**

43260 sprintf, snprintf — print formatted output

43261 **SYNOPSIS**

43262 `#include <stdio.h>`

43263 `int snprintf(char *restrict s, size_t n,` |
43264 `    const char *restrict format, /* args */ ...);` |
43265 `int sprintf(char *restrict s, const char *restrict format, ...);` |

43266 **DESCRIPTION** |

43267 Refer to *fprintf*( ).

NAME

sqrt, sqrtf, sqrtl — square root function

SYNOPSIS

```
#include <math.h>

double sqrt(double x);
float sqrtf(float x);
long double sqrtl(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 square root of x, $\sqrt{x}$.

An application wishing to check for error situations should set *errno* to 0 before calling *sqrt*( ). If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the square root of x.

XSI If x is NaN, NaN shall be returned and *errno* may be set to [EDOM].

XSI If x is negative, 0.0 or NaN shall be returned and *errno* shall be set to [EDOM].

ERRORS

These functions shall fail if:

[EDOM] The value of x is negative.

These functions may fail if:

XSI [EDOM] The value of x is NaN.

XSI No other errors shall occur.

EXAMPLES

Taking the Square Root of 9.0

```
#include <math.h>
...
double x = 9.0;
double result;
...
result = sqrt(x);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isnan( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**, **<stdio.h>**

CHANGE HISTORY

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

Issue 4

References to *matherr*( ) are removed.

The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the ISO C standard and to rationalize error handling in the mathematics functions.

The return value specified for [EDOM] is marked as an extension.

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 *sqrtf*( ) and *sqrtl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

43320 **NAME**

43321 srand — pseudo-random number generator

43322 **SYNOPSIS**

43323 `#include <stdlib.h>`

43324 `void srand(unsigned` *`seed`*`);` |

43325 **DESCRIPTION** |

43326 Refer to *rand*( ).

43327 **NAME**

43328 srand48 — seed uniformly distributed double-precision pseudo-random number generator

43329 **SYNOPSIS**

43330 XSI
```
#include <stdlib.h>

void srand48(long seedval);
```
43331 |

43332 |

43333 **DESCRIPTION**

43334 Refer to *drand48*( ).

43335 **NAME**

43336 srandom — seed pseudo-random number generator

43337 **SYNOPSIS**

43338 XSI
```
#include <stdlib.h>

void srandom(unsigned seed);
```
43339 |

43340 |

43341 **DESCRIPTION**

43342 Refer to *initstate*( ).

43343 **NAME**

43344 sscanf — convert formatted input

43345 **SYNOPSIS**

43346 `#include <stdio.h>`

43347 `int sscanf(const char *restrict s, const char *restrict format, ...);` |

43348 **DESCRIPTION** |

43349 Refer to *fscanf*( ).

43350 **NAME**

43351 stat — get file status

43352 **SYNOPSIS**

43353 `#include <sys/stat.h>`

43354 `int stat(const char *restrict path, struct stat *restrict buf);`

43355 **DESCRIPTION**

43356 The *stat*( ) function obtains information about the named file and writes it to the area pointed to
43357 by the *buf* argument. The *path* argument points to a path name naming a file. Read, write, or
43358 execute permission of the named file is not required. An implementation that provides
43359 additional or alternate file access control mechanisms may, under implementation-defined
43360 conditions, cause *stat*( ) to fail. In particular, the system may deny the existence of the file
43361 specified by *path*.

43362 If the named file is a symbolic link, the *stat*( ) function shall continue path name resolution using
43363 the contents of the symbolic link, and shall return information pertaining to the resulting file if
43364 the file exists.

43365 The *buf* argument is a pointer to a **stat** structure, as defined in the header **<sys/stat.h>**, into
43366 which information is placed concerning the file.

43367 The *stat*( ) function updates any time-related fields (as described in the definition of **File Times**
43368 **Update** in the Base Definitions volume of IEEE Std. 1003.1-200x), before writing into the **stat**
43369 structure.

43370 The structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime*, and *st_mtime*
43371 shall have meaningful values for all file types defined in this volume of IEEE Std. 1003.1-200x.
43372 The value of the member *st_nlink* shall be set to the number of links to the file.

43373 **RETURN VALUE**

43374 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
43375 indicate the error.

43376 **ERRORS**

43377 The *stat*( ) function shall fail if:

43378 [EACCES] Search permission is denied for a component of the path prefix.

43379 [EIO] An error occurred while reading from the file system.

43380 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
43381 argument.

43382 [ENAMETOOLONG]
43383 The length of the *path* argument exceeds {PATH_MAX} or a path name
43384 component is longer than {NAME_MAX}.

43385 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

43386 [ENOTDIR] A component of the path prefix is not a directory.

43387 [EOVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file
43388 serial number cannot be represented correctly in the structure pointed to by
43389 *buf*.

43390 The *stat*( ) function may fail if:

43391 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
43392 resolution of the *path* argument.

[ENAMETOOLONG] |
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted path name string exceeded {PATH_MAX}. |

[EOVERFLOW] A value to be stored would overflow one of the members of the **stat** structure. |

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>

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) {

    /* Get entry's information. */
    if (stat(dp->d_name, &statbuf) == -1)
        continue;

    /* Print out type, permissions, and number of links. */
    printf("%10.10s", sperm (statbuf.st_mode));
    printf("%4d", statbuf.st_nlink);
```

```
        /* Print out owners name if it is found using getpwuid(). */
        if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
            printf(" %-8.8s", pwd->pw_name);
        else
            printf(" %-8d", statbuf.st_uid);
        /* Print out group name if it's found using getgrgid(). */
        if ((grp = getgrgid(statbuf.st_gid)) != NULL)
            printf(" %-8.8s", grp->gr_name);
        else
            printf(" %-8d", statbuf.st_gid);
        /* Print size of file. */
        printf("%9ld", statbuf.st_size);
        tm = localtime(&statbuf.st_mtime);
        /* Get localized date string. */
        strftime(datestring, sizeof(datestring), nl_langinfo(D_T_FMT), tm);
        printf(" %s %s\n", datestring, dp->d_name);
    }
```

APPLICATION USAGE

None.

RATIONALE

The intent of the paragraph describing ''additional or alternate file access control mechanisms'' is to allow a secure implementation where a process with a label that does not dominate the file's label cannot perform a *stat*( ) function. This is not related to read permission; a process with a label that dominates the file's label does not need read permission. An implementation that supports write-up operations could fail *fstat*( ) function calls even though it has a valid file descriptor open for writing.

FUTURE DIRECTIONS

None.

SEE ALSO

fstat( ), *lstat*( ), *readlink*( ), *symlink*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

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

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument *path* is changed from **char*** to **const char***.
- In the DESCRIPTION is changed as follows:
  - — Statements indicating the purpose of this function and a paragraph defining the contents of **stat** structure members are added.
  - — The words ''extended security controls'' are replaced by ''additional or alternate file access control mechanisms''.

The following change is incorporated for alignment with the FIPS requirements:

- In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path name component is larger than {NAME_MAX} is now defined as mandatory and marked as an extension.

Issue 4, Version 2

The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- In the mandatory section, [EIO] is added to indicate that a physical I/O error has occurred, and [ELOOP] to indicate that too many symbolic links were encountered during path name resolution.
- In the optional section, a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of path name resolution of a symbolic link.
- In the optional section, [EOVERFLOW] is added to indicate that a value to be stored in a member of the **stat** structure would cause overflow.

Issue 5

Large File Summit extensions are added.

Issue 6

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

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 [EOVERFLOW] mandatory error condition is added. This change is to support large files.
- The [ENAMETOOLONG] and the second [EOVERFLOW] 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 DESCRIPTION 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.

43516 **NAME**

43517 statvfs — get file system information

43518 **SYNOPSIS**

43519 XSI
```
#include <sys/statvfs.h>

int statvfs(const char *restrict path, struct statvfs *restrict buf);
```

43520 |

43521 |

43522 **DESCRIPTION**

43523 Refer to *fstatvfs*( ).

NAME

stderr, stdin, stdout — standard I/O streams

SYNOPSIS

```
#include <stdio.h>

extern FILE *stderr, *stdin, *stdout;
```

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.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type **FILE**. The *fopen*( ) function creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the **<stdio.h>** header and associated with the standard open files.

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

CX The following symbolic values in **<unistd.h>** define the file descriptors that shall be associated with the C-language *stdin*, *stdout*, and *stderr* when the application is started:

STDIN_FILENO	Standard input value, *stdin*. Its value is 0.
STDOUT_FILENO	Standard output value, *stdout*. Its value is 1.
STDERR_FILENO	Standard error value, *stderr*. Its value is 2.

stderr is expected to be open for reading and writing.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fclose( ), *feof*( ), *ferror*( ), *fileno*( ), *fopen*( ), *fread*( ), *fseek*( ), *getc*( ), *gets*( ), *popen*( ), *printf*( ), *putc*( ), *puts*( ), *read*( ), *scanf*( ), *setbuf*( ), *setvbuf*( ), *tmpfile*( ), *ungetc*( ), *vprintf*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdio.h>**, **<unistd.h>**

43565 **CHANGE HISTORY**

43566 First released in Issue 1.

43567 **Issue 6**

43568 Extensions beyond the ISO C standard are now marked.

NAME

strcasecmp, strncasecmp — case-insensitive string comparisons

SYNOPSIS

XSI
```
#include <strings.h>

int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, size_t n);
```

DESCRIPTION

The *strcasecmp*( ) function compares, while ignoring differences in case, the string pointed to by *s1* to the string pointed to by *s2*. The *strncasecmp*( ) function compares, while ignoring differences in case, not more than *n* bytes from the string pointed to by *s1* to the string pointed to by *s2*.

In the POSIX locale, *strcasecmp*( ) and *strncasecmp*( ) do upper to lower conversions, then a byte comparison. The results are unspecified in other locales.

RETURN VALUE

Upon completion, *strcasecmp*( ) shall return an integer greater than, equal to, or less than 0, if the string pointed to by *s1* is, ignoring case, greater than, equal to, or less than the string pointed to by *s2*, respectively.

Upon successful completion, *strncasecmp*( ) shall return an integer greater than, equal to, or less than 0, if the possibly null-terminated array pointed to by *s1* is, ignoring case, greater than, equal to, or less than the possibly null-terminated array pointed to by *s2*, respectively.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std. 1003.1-200x, **<strings.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

43606 **NAME**

43607 strcat — concatenate two strings

43608 **SYNOPSIS**

43609 `#include <string.h>`

43610 `char *strcat(char *restrict s1, const char *restrict s2);`

43611 **DESCRIPTION**

43612 CX The functionality described on this reference page is aligned with the ISO C standard. Any
43613 conflict between the requirements described here and the ISO C standard is unintentional. This
43614 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

43615 The *strcat*( ) function shall append a copy of the string pointed to by *s2* (including the
43616 terminating null byte) to the end of the string pointed to by *s1*. The initial byte of *s2* overwrites
43617 the null byte at the end of *s1*. If copying takes place between objects that overlap, the behavior is
43618 undefined.

43619 **RETURN VALUE**

43620 The *strcat*( ) function shall return *s1*; no return value is reserved to indicate an error.

43621 **ERRORS**

43622 No errors are defined.

43623 **EXAMPLES**

43624 None.

43625 **APPLICATION USAGE**

43626 This issue is aligned with the ISO C standard; this does not affect compatibility with XPG3
43627 applications. Reliable error detection by this function was never guaranteed.

43628 **RATIONALE**

43629 None.

43630 **FUTURE DIRECTIONS**

43631 None.

43632 **SEE ALSO**

43633 *strncat*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<string.h>**

43634 **CHANGE HISTORY**

43635 First released in Issue 1. Derived from Issue 1 of the SVID.

43636 **Issue 4**

43637 The DESCRIPTION is changed to make it clear that the function manipulates bytes rather than
43638 (possibly multi-byte) characters.

43639 The following change is incorporated for alignment with the ISO C standard:

43640 • The type of argument *s2* is changed from **char*** to **const char***.

43641 **Issue 6**

43642 The *strcat*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

NAME

strchr — string scanning operation

SYNOPSIS

```
#include <string.h>

char *strchr(const char *s, int c);
```

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.

CX The *strchr*( ) function shall locate the first occurrence of *c* (converted to an **unsigned char**) in the string pointed to by *s*. The terminating null byte is considered to be part of the string.

RETURN VALUE

Upon completion, *strchr*( ) shall return a pointer to the byte, or a null pointer if the byte was not found.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strrchr( ), 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 4

The DESCRIPTION and RETURN VALUE sections are changed to make it clear that the function manipulates bytes rather than (possibly multi-byte) characters.

The APPLICATION USAGE section is removed.

The following change is incorporated for alignment with the ISO C standard:

- The type of argument *s* is changed from **char*** to **const char***.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

strcmp — compare two strings

SYNOPSIS

```
#include <string.h>

int strcmp(const char *s1, const char *s2);
```

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 *strcmp*( ) function shall compare the string pointed to by *s1* to the string pointed to by *s2*.

The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the strings being compared.

RETURN VALUE

Upon completion, *strcmp*( ) shall return an integer greater than, equal to, or less than 0, if the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*, respectively.

ERRORS

No errors are defined.

EXAMPLES

Checking a Password Entry

The following example compares the information read from standard input to the value of the name of the user entry. If the *strcmp*( ) function returns 0 (indicating a match), a further check will be made to see if the user entered the proper old password. The *crypt*( ) function is used to encrypt the old password entered by the user, using the value of the encrypted password in the **passwd** structure as the salt. If this value matches the value of the encrypted **passwd** in the structure, the entered password *oldpasswd* is the correct user's password. Finally, the program encrypts the new password so that it can store the imformation in the **passwd** structure.

```
#include <string.h>
#include <unistd.h>
#include <stdio.h>
...
int valid_change;
struct passwd *p;
char user[100];
char oldpasswd[100];
char newpasswd[100];
char savepasswd[100];
...
if (strcmp(p->pw_name, user) == 0) {
    if (strcmp(p->pw_passwd, crypt(oldpasswd, p->pw_passwd)) == 0) {
        strcpy(savepasswd, crypt(newpasswd, user));
        p->pw_passwd = savepasswd;
        valid_change = 1;
    }
    else {
```

```
43725                fprintf(stderr, "Old password is not valid\n");
43726            }
43727        }
43728        ...
```

43729 **APPLICATION USAGE**

43730 None.

43731 **RATIONALE**

43732 None.

43733 **FUTURE DIRECTIONS**

43734 None.

43735 **SEE ALSO**

43736 *strncmp*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<string.h>**

43737 **CHANGE HISTORY**

43738 First released in Issue 1. Derived from Issue 1 of the SVID.

43739 **Issue 4**

43740 The DESCRIPTION is changed to make it clear that *strcmp*( ) compares bytes rather than
43741 (possibly multi-byte) characters.

43742 The following change is incorporated for alignment with the ISO C standard:

43743 • The type of arguments *s1* and *s2* is changed from **char*** to **const char***.

43744 **Issue 6**

43745 Extensions beyond the ISO C standard are now marked.

NAME

strcoll — string comparison using collating information

SYNOPSIS

```
#include <string.h>

int strcoll(const char *s1, const char *s2);
```

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 *strcoll*( ) function shall compare the string pointed to by *s1* to the string pointed to by *s2*, both interpreted as appropriate to the *LC_COLLATE* category of the current locale.

CX The *strcoll*( ) function shall not change the setting of *errno* if successful.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to 0, then call *strcoll*( ), then check *errno*.

RETURN VALUE

Upon successful completion, *strcoll*( ) shall return an integer greater than, equal to, or less than 0, according to whether the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2* when both are interpreted as appropriate to the current locale. On error, *strcoll*( ) may set *errno*, but no return value is reserved to indicate an error.

ERRORS

The *strcoll*( ) function may fail if:

CX [EINVAL] The *s1* or *s2* arguments contain characters outside the domain of the collating sequence.

EXAMPLES

Comparing Nodes

The following example uses an application-defined function, *node_compare*( ), to compare two nodes based on an alphabetical ordering of the *string* field.

```
#include <string.h>
...
struct node { /* These are stored in the table. */
    char *string;
    int length;
};
...
int node_compare(const void *node1, const void *node2)
{
    return strcoll(((const struct node *)node1)->string,
        ((const struct node *)node2)->string);
}
...
```

APPLICATION USAGE

The *strxfrm*( ) and *strcmp*( ) functions should be used for sorting large lists.

43788 **RATIONALE**

43789 None.

43790 **FUTURE DIRECTIONS**

43791 None.

43792 **SEE ALSO**

43793 *strcmp*( ), *strxfrm*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<string.h>**

43794 **CHANGE HISTORY**

43795 First released in Issue 3.

43796 **Issue 4**

43797 A paragraph describing how the sign of the return value should be determined is removed from
43798 the DESCRIPTION.

43799 The [EINVAL] error is marked as an extension.

43800 The following changes are incorporated for alignment with the ISO C standard:

43801 • The function is no longer marked as an extension.

43802 • The type of arguments *s1* and *s2* are changed from **char*** to **const char***.

43803 **Issue 5**

43804 The DESCRIPTION is updated to indicate that *errno* does not be changed if the function is
43805 successful.

43806 **Issue 6**

43807 Extensions beyond the ISO C standard are now marked.

43808 The following new requirements on POSIX implementations derive from alignment with the
43809 Single UNIX Specification:

43810 • The [EINVAL] optional error condition is added.

NAME

strcpy — copy a string

SYNOPSIS

```
#include <string.h>

char *strcpy(char *restrict s1, const char *restrict s2);
```

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 *strcpy*( ) function shall copy the string pointed to by *s2* (including the terminating null byte) into the array pointed to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *strcpy*( ) function shall return *s1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

Initializing a String

The following example copies the string "----------" into the *permstring* variable.

```
#include <string.h>
...
static char permstring[11];
...
strcpy(permstring, "----------");
...
```

Storing a Key and Data

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***.)

```
#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);
```

```
43855       ...
43856       if ((curtbl->key = malloc(strlen(key) + 1)) == NULL) {
43857           perror("malloc"); dbfree(tbl, count); return NULL;
43858       }
43859       strcpy(curtbl->key, key);
43860       if ((curtbl->data = malloc(strlen(data) + 1)) == NULL) {
43861           perror("malloc"); free(curtbl->key); dbfree(tbl, count); return NULL;
43862       }
43863       strcpy(curtbl->data, data);
43864       ...
```

APPLICATION USAGE

Character movement is performed differently in different implementations. Thus, overlapping moves may yield surprises.

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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strncpy( ), 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 4

The DESCRIPTION is changed to make it clear that the function manipulates bytes rather than (possibly multi-byte) characters.

The following change is incorporated for alignment with the ISO C standard:

- The type of argument *s2* is changed from **char*** to **const char***.

Issue 6

The *strcpy*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

43885 **NAME**

43886 strcspn — get length of a complementary substring

43887 **SYNOPSIS**

43888
```
#include <string.h>
```

43889
```
size_t strcspn(const char *s1, const char *s2);
```

43890 **DESCRIPTION**

43891 CX The functionality described on this reference page is aligned with the ISO C standard. Any
43892 conflict between the requirements described here and the ISO C standard is unintentional. This
43893 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

43894 The *strcspn*( ) function shall compute the length of the maximum initial segment of the string
43895 pointed to by *s1* which consists entirely of bytes *not* from the string pointed to by *s2*.

43896 **RETURN VALUE**

43897 The *strcspn*( ) function shall return the length of the computed segment of the string pointed to
43898 by *s1*; no return value is reserved to indicate an error.

43899 **ERRORS**

43900 No errors are defined.

43901 **EXAMPLES**

43902 None.

43903 **APPLICATION USAGE**

43904 None.

43905 **RATIONALE**

43906 None.

43907 **FUTURE DIRECTIONS**

43908 None.

43909 **SEE ALSO**

43910 *strspn*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<string.h>** |

43911 **CHANGE HISTORY**

43912 First released in Issue 1. Derived from Issue 1 of the SVID. |

43913 **Issue 4**

43914 The DESCRIPTION is changed to make it clear that the function manipulates bytes rather than
43915 (possibly multi-byte) characters.

43916 The following change is incorporated for alignment with the ISO C standard:

43917 • The type of arguments *s1* and *s2* is changed from **char*** to **const char***.

43918 **Issue 5**

43919 The RETURN VALUE section is updated to indicated that *strcspn*( ) returns the length of *s1*, and
43920 not *s1* itself as was previously stated.

43921 **Issue 6**

43922 The Open Group corrigenda item U030/1 has been applied. The text of the RETURN VALUE
43923 section is updated to indicate that the computed segment length is returned, not the *s1* length.

NAME

strdup — duplicate a string

SYNOPSIS

XSI
```
#include <string.h>

char *strdup(const char *s1);
```

DESCRIPTION

The *strdup*( ) function shall return a pointer to a new string, which is a duplicate of the string pointed to by *s1*. The returned pointer can be passed to *free*( ). A null pointer is returned if the new string cannot be created.

RETURN VALUE

The *strdup*( ) function shall return a pointer to a new string on success. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

ERRORS

The *strdup*( ) function may fail if:

[ENOMEM] Storage space available is insufficient.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

free( ), *malloc*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

43954 **NAME**

43955 strerror, strerror_r — get error message string

43956 **SYNOPSIS**

43957
```
#include <string.h>
```

43958
```
char *strerror(int errnum);
```

43959
```
int strerror_r(int errnum, char *strerrbuf, size_t buflen);
```

43960 **DESCRIPTION**

43961 The *strerror*( ) function maps the error number in *errnum* to a locale-dependent error message
43962 string and returns a pointer to it. The string pointed to must not be modified by the application,
43963 but may be overwritten by a subsequent call to *strerror*( ) or *perror*( ).

43964 The contents of the error message strings returned by *strerror*( ) should be determined by the
43965 setting of the *LC_MESSAGES* category in the current locale.

43966 The implementation shall behave as if no function defined in this volume of
43967 IEEE Std. 1003.1-200x calls *strerror*( ).

43968 The *strerror*( ) function shall not change the setting of *errno* if successful.

43969 Because no return value is reserved to indicate an error, an application wishing to check for error
43970 situations should set *errno* to 0, then call *strerror*( ), then check *errno*.

43971 The *strerror*( ) function need not be reentrant. A function that is not required to be reentrant is
43972 not required to be thread-safe.

43973 The *strerror_r*( ) function maps the error number in *errnum* to a locale-dependent error message
43974 string and returns the string in the buffer pointed to by *strerrbuf*, with length *buflen*.

43975 **RETURN VALUE**

43976 Upon successful completion, *strerror*( ) shall return a pointer to the generated message string. On
43977 error *errno* may be set, but no return value is reserved to indicate an error.

43978 Upon successful completion, *strerror_r*( ) returns 0. Otherwise, an error number is returned to
43979 indicate the error.

43980 **ERRORS**

43981 These functions may fail if:

43982 [EINVAL] The value of *errnum* is not a valid error number.

43983 The *strerror_r*( ) function may fail if:

43984 [ERANGE] Insufficient storage was supplied via *strerrbuf* and *buflen* to contain the
43985 generated message string.

43986 **EXAMPLES**

43987 None.

43988 **APPLICATION USAGE**

43989 None.

43990 **RATIONALE**

43991 None.

43992 **FUTURE DIRECTIONS**

43993 None.

SEE ALSO

perror( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 4

The DESCRIPTION is changed as follows:

- The term ‘‘language-dependent’’ is replaced by ‘‘locale-dependent’’.
- A statement about the use of the *LC_MESSAGES* category for determining the language of error messages is added and marked as an extension.

The fact that *strerror*( ) can return a null pointer on failure and set *errno* is marked as an extension.

The [EINVAL] error is marked as an extension.

The FUTURE DIRECTIONS section is removed.

The following change is incorporated for alignment with the ISO C standard:

- The function is no longer marked as an extension.

Issue 5

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, the fact that *errno* may be set is added.
- The [EINVAL] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The *strerror_r*( ) function is added in response to IEEE PASC Interpretation 1003.1c #39.

NAME

strfmon — convert monetary value to a string

SYNOPSIS

XSI
```
#include <monetary.h>

ssize_t strfmon(char *restrict s, size_t maxsize,
    const char *restrict format, ...);
```

DESCRIPTION

The *strfmon*( ) function shall place characters into the array pointed to by *s* as controlled by the string pointed to by *format*. No more than *maxsize* bytes are placed into the array.

The format is a character string that contains two types of objects: *plain characters*, which are simply copied to the output stream, and *conversion specifications*, each of which results in the fetching of zero or more arguments which are converted and formatted. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are simply ignored.

The application shall ensure that a conversion specification consists of the following sequence:

- A '%' character
- Optional flags
- Optional field width
- Optional left precision
- Optional right precision
- A required conversion character that determines the conversion to be performed

Flags

One or more of the following optional flags can be specified to control the conversion:

=f An '=' followed by a single character *f* which is used as the numeric fill character. The application shall ensure that the fill character is representable in a single byte in order to work with precision and width counts. The default numeric fill character is the <space> character. This flag does not affect field width filling which always uses the <space> character. This flag is ignored unless a left precision (see below) is specified.

^ Do not format the currency amount with grouping characters. The default is to insert the grouping characters if defined for the current locale.

+ or (Specify the style of representing positive and negative currency amounts. Only one of '+' or '(' may be specified. If '+' is specified, the locale's equivalent of '+' and '−' are used (for example, in the U.S., the empty string if positive and '−' if negative). If '(' is specified, negative amounts are enclosed within parentheses. If neither flag is specified, the '+' style is used.

! Suppress the currency symbol from the output conversion.

− Specify the alignment. If this flag is present all fields are left-justified (padded to the right) rather than right-justified.

Field Width

w A decimal digit string *w* specifying a minimum field width in bytes in which the result of the conversion is right-justified (or left-justified if the flag '−' is specified). The default is 0.

Left Precision

#n A '#' followed by a decimal digit string *n* specifying a maximum number of digits expected to be formatted to the left of the radix character. This option can be used to keep the formatted output from multiple calls to the *strfmon*( ) function aligned in the same columns. It can also be used to fill unused positions with a special character as in "$***123.45". This option causes an amount to be formatted as if it has the number of digits specified by *n*. If more than *n* digit positions are required, this conversion specification is ignored. Digit positions in excess of those actually required are filled with the numeric fill character (see the *=f* flag above).

If grouping has not been suppressed with the '^' flag, and it is defined for the current locale, grouping separators are inserted before the fill characters (if any) are added. Grouping separators are not applied to fill characters even if the fill character is a digit.

To ensure alignment, any characters appearing before or after the number in the formatted output such as currency or sign symbols are padded as necessary with <space> characters to make their positive and negative formats an equal length.

Right Precision

.p A period followed by a decimal digit string *p* specifying the number of digits after the radix character. If the value of the right precision *p* is 0, no radix character appears. If a right precision is not included, a default specified by the current locale is used. The amount being formatted is rounded to the specified number of digits prior to formatting.

Conversion Characters

The conversion characters and their meanings are:

i The **double** argument is formatted according to the locale's international currency format (for example, in the U.S.: USD 1,234.56).

n The **double** argument is formatted according to the locale's national currency format (for example, in the U.S.: $1,234.56).

% Convert to a '%'; no argument is converted. The entire conversion specification shall be "%%".

Locale Information

The *LC_MONETARY* category of the program's locale 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: 1995 standard.

If the value of *maxsize* is greater than {SSIZE_MAX}, the result is implementation-defined.

RETURN VALUE

If the total number of resulting bytes including the terminating null byte is not more than *maxsize*, *strfmon*( ) shall return the number of bytes placed into the array pointed to by *s*, not including the terminating null byte. Otherwise, –1 shall be returned, the contents of the array are indeterminate, and *errno* shall be set to indicate the error.

ERRORS

The *strfmon*( ) function shall fail if:

[E2BIG] Conversion stopped due to lack of space in the buffer.

EXAMPLES

Given a locale for the U.S. and the values 123.45, –123.45, and 3456.781:

Conversion Specification	Output	Comments
%*n*	$123.45 -$123.45 $3,456.78	Default formatting.
%11*n*	$123.45 -$123.45 $3,456.78	Right align within an 11 character field.
%#5*n*	$ 123.45 -$ 123.45 $ 3,456.78	Aligned columns for values up to 99,999.
%=*#5*n*	$***123.45 -$***123.45 $*3,456.78	Specify a fill character.
%=0#5*n*	$000123.45 -$000123.45 $03,456.78	Fill characters do not use grouping even if the fill character is a digit.
%^#5*n*	$ 123.45 -$ 123.45 $ 3456.78	Disable the grouping separator.
%^#5.0*n*	$ 123 -$ 123 $ 3457	Round off to whole units.
%^#5.4*n*	$ 123.4500 -$ 123.4500 $ 3456.7810	Increase the precision.
%(#5*n*	123.45 ($ 123.45) $ 3,456.78	Use an alternative pos/neg style.
%(!#5*n*	123.45 (123.45) 3,456.78	Disable the currency symbol.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

Lowercase conversion characters are reserved for future standards use and uppercase for implementation-defined use.

SEE ALSO

localeconv( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<monetary.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 5

Moved from ENHANCED I18N to BASE.

The [ENOSYS] error is removed.

A sentence is added to the DESCRIPTION warning about values of *maxsize* that are greater than {SSIZE_MAX}.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *strfmon*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

44161 **NAME**

44162 strftime — convert date and time to a string

44163 **SYNOPSIS**

44164 `#include <time.h>`

44165 `size_t strftime(char *restrict s, size_t maxsize,`
44166 `    const char *restrict format, const struct tm *restrict timptr);`

44167 **DESCRIPTION**

44168 CX The functionality described on this reference page is aligned with the ISO C standard. Any
44169 conflict between the requirements described here and the ISO C standard is unintentional. This
44170 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

44171 The *strftime*( ) function shall place bytes into the array pointed to by *s* as controlled by the string
44172 pointed to by *format*. The *format* string consists of zero or more conversion specifications and
44173 ordinary characters. A conversion specification consists of a '%' character, possibly followed by
44174 an *E* or *O* modifier, and a terminating conversion character that determines the conversion
44175 specification's behavior. All ordinary characters (including the terminating null byte) are copied
44176 unchanged into the array. If copying takes place between objects that overlap, the behavior is
44177 undefined. No more than *maxsize* bytes are placed into the array. Each conversion specification
44178 is replaced by appropriate characters as described in the following list. The appropriate
44179 characters are determined by the program's locale and by the values contained in the structure
44180 pointed to by *timptr*.

44181 CX Local timezone information is used as though *strftime*( ) called *tzset*( ).

44182 %*a* Replaced by the locale's abbreviated weekday name.

44183 %*A* Replaced by the locale's full weekday name.

44184 %*b* Replaced by the locale's abbreviated month name.

44185 %*B* Replaced by the locale's full month name.

44186 %*c* Replaced by the locale's appropriate date and time representation.

44187 %*C* Replaced by the century number (the year divided by 100 and truncated to an integer)
44188 as a decimal number [00-99].

44189 %*d* Replaced by the day of the month as a decimal number [01,31].

44190 %*D* Same as %*m*/%*d*/%*y*.

44191 %*e* Replaced by the day of the month as a decimal number [1,31]; a single digit is preceded
44192 by a space.

44193 %*F* Equivalent to %*Y*-%*m*-%*d* (the ISO 8601:1988 standard date format).

44194 %*g* Replaced by the last 2 digits of the week-based year (see below) as a decimal number
44195 (00-99).

44196 %*G* Replaced by the week-based year (see below) as a decimal number (for example, 1977).

44197 %*h* Same as %*b*.

44198 %*H* Replaced by the hour (24-hour clock) as a decimal number [00,23].

44199 %*I* Replaced by the hour (12-hour clock) as a decimal number [01,12].

44200 %*j* Replaced by the day of the year as a decimal number [001,366].

%*m* Replaced by the month as a decimal number [01,12].

%*M* Replaced by the minute as a decimal number [00,59].

%*n* Replaced by a <newline> character.

%*p* Replaced by the locale's equivalent of either a.m. or p.m.

%*r* Replaced by the time in a.m. and p.m. notation; in the POSIX locale this is equivalent to %*I*:%*M*:%*S* %*p*.

%*R* Replaced by the time in 24 hour notation (%*H*:%*M*).

%*S* Replaced by the second as a decimal number [00,61].

%*t* Replaced by a <tab> character.

%*T* Replaced by the time (%*H*:%*M*:%*S*).

%*u* Replaced by the weekday as a decimal number [1,7], with 1 representing Monday.

%*U* Replaced by the week number of the year (Sunday as the first day of the week) as a decimal number [00,53].

%*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.

%*w* Replaced by the weekday as a decimal number [0,6], with 0 representing Sunday.

%*W* Replaced by the week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.

%*x* Replaced by the locale's appropriate date representation.

%*X* Replaced by the locale's appropriate time representation.

%*y* Replaced by the year without century as a decimal number [00,99].

%*Y* Replaced by the year with century as a decimal number.

%*z* Replaced by the offset from UTC in the ISO 8601:1988 standard format –0430 (meaning 4 hours 30 minutes behind UTC, west of Greenwich), or by no characters if no timezone is determinable.

%*Z* Replaced by the timezone name or abbreviation, or by no bytes if no timezone information exists.

%% Replaced by '%'.

If a conversion specification does not correspond to any of the above, the behavior is undefined.

Modified Conversion Specifiers

CX Some conversion specifiers can be modified by the *E* or *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 for the current locale, (see ERA in the Base Definitions volume of IEEE Std. 1003.1-200x, Section 7.3.5, LC_TIME) the behavior shall be as if the unmodified conversion specification were used.

%*Ec* Replaced by the locale's alternative appropriate date and time representation.

%*EC*	Replaced by the name of the base year (period) in the locale's alternative representation.
%*Ex*	Replaced by the locale's alternative date representation.
%*EX*	Replaced by the locale's alternative time representation.
%*Ey*	Replaced by the offset from %*EC* (year only) in the locale's alternative representation.
%*EY*	Replaced by the full alternative year representation.
%*Od*	Replaced by the day of the month, using the locale's alternative numeric symbols, filled as needed with leading zeros if there is any alternative symbol for zero; otherwise, with leading spaces.
%*Oe*	Replaced by the day of the month, using the locale's alternative numeric symbols, filled as needed with leading spaces.
%*OH*	Replaced by the hour (24-hour clock) using the locale's alternative numeric symbols.
%*OI*	Replaced by the hour (12-hour clock) using the locale's alternative numeric symbols.
%*Om*	Replaced by the month using the locale's alternative numeric symbols.
%*OM*	Replaced by the minutes using the locale's alternative numeric symbols.
%*OS*	Replaced by the seconds using the locale's alternative numeric symbols.
%*Ou*	Replaced by the weekday as a number in the locale's alternative representation (Monday=1).
%*OU*	Replaced by the week number of the year (Sunday as the first day of the week, rules corresponding to %*U*) using the locale's alternative numeric symbols.
%*OV*	Replaced by the week number of the year (Monday as the first day of the week, rules corresponding to %*V*) using the locale's alternative numeric symbols.
%*Ow*	Replaced by the number of the weekday (Sunday=0) using the locale's alternative numeric symbols.
%*OW*	Replaced by the week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.
%*Oy*	Replaced by the year (offset from %*C*) using the locale's alternative numeric symbols.

%*g*, %*G*, and %*V* give values according to the ISO 8601:1988 standard week-based year. In this system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th, which is also the week that includes the first Thursday of the year, and is also the first week that contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January 1999, %*G* is replaced by 1998 and %*V* is replaced by 53. If December 29th, 30th, or 31st is a Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday 30th December 1997, %*G* is replaced by 1998 and %*V* is replaced by 01.

If a conversion specifier is not one of the above, the behavior is undefined.

RETURN VALUE

If the total number of resulting bytes including the terminating null byte is not more than *maxsize*, *strftime*( ) shall return the number of bytes placed into the array pointed to by *s*, not including the terminating null byte. Otherwise, 0 shall be returned and the contents of the array are indeterminate.

ERRORS

No errors are defined.

EXAMPLES

Getting a Localized Date String

The following example first sets the locale to the user's default. The locale information will be used in the *nl_langinfo*( ) and *strftime*( ) functions. The *nl_langinfo*( ) function returns the localized date string which specifies how the date is laid out. The *strftime*( ) function takes this information and, using the **tm** structure for values, places the date and time information into *datestring*.

```
#include <time.h>
#include <locale.h>
#include <langinfo.h>
...
struct tm *tm;
char datestring[256];
...
setlocale (LC_ALL, "");
...
strftime (datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm);
...
```

APPLICATION USAGE

The range of values for %*S* is [00,61] rather than [00,59] to allow for the occasional leap second and even more infrequent double leap second.

Some of the conversion specifications marked EX 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*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strptime*( ), *time*( ), *utime*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 4

The DESCRIPTION is expanded to describe modified conversion specifiers.

%*C*, %*e*, %*R*, %*u*, and %*V* are added to the list of valid conversion specifications.

The DESCRIPTION and RETURN VALUE sections are changed to make it clear when the function uses byte values rather than (possibly multi-byte) character values.

The following changes are incorporated for alignment with the ISO C standard:

- The type of argument *format* is changed from **char*** to **const char***, and the type of argument *timptr* is changed from **struct tm*** to **const struct tm***.
- In the description of the %*Z* conversion specification, the words ‘‘or abbreviation’’ are added to indicate that *strftime*( ) does not necessarily return a full timezone name.

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 now marked.

The Open Group corrigenda item U033/8 has been 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:

- The *strftime*( ) prototype is updated.
- The DESCRIPTION is extensively revised.

NAME

strlen — get string length

SYNOPSIS

```
#include <string.h>

size_t strlen(const char *s);
```

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 *strlen*( ) function shall compute the number of bytes in the string to which *s* points, not including the terminating null byte.

RETURN VALUE

The *strlen*( ) function shall return the length of *s*; no return value shall be reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

Getting String Lengths

The following example sets the maximum length of *key* and *data* by using *strlen*( ) to get the lengths of those strings.

```
#include <string.h>
...
struct element {
    char *key;
    char *data;
};
...
char *key, *data;
int len;

*keylength = *datalength = 0;
...
if ((len = strlen(key)) > *keylength)
    *keylength = len;
if ((len = strlen(data)) > *datalength)
    *datalength = len;
...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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 4

The DESCRIPTION is changed to make it clear that the function works in units of bytes rather than (possibly multi-byte) characters.

The following change is incorporated for alignment with the ISO C standard:

- The type of argument *s* is changed from **char*** to **const char***.

Issue 5

The RETURN VALUE section is updated to indicate that *strlen*( ) returns the length of *s*, and not *s* itself as was previously stated.

NAME

strncasecmp — case-insensitive string comparison

SYNOPSIS

XSI

```
#include <strings.h>

int strncasecmp(const char *s1, const char *s2, size_t n);
```

DESCRIPTION

Refer to *strcasecmp*( ).

NAME

strncat — concatenate a string with part of another

SYNOPSIS

```
#include <string.h>

char *strncat(char *restrict s1, const char *restrict s2, 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 *strncat*( ) function shall append not more than *n* bytes (a null byte and bytes that follow it are not appended) from the array pointed to by *s2* to the end of the string pointed to by *s1*. The initial byte of *s2* overwrites the null byte at the end of *s1*. A terminating null byte is always appended to the result. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *strncat*( ) function shall return *s1*; no return value shall be reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strcat( ), 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 4

The DESCRIPTION is changed to make it clear that the function manipulates bytes rather than (possibly multi-byte) characters.

The following change is incorporated for alignment with the ISO C standard:

- The type of argument *s2* is changed from **char*** to **const char***.

Issue 6

The *strncat*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

NAME

strncmp — compare part of two strings

SYNOPSIS

```
#include <string.h>

int strncmp(const char *s1, const char *s2, 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 *strncmp*( ) function shall compare not more than *n* bytes (bytes that follow a null byte are not compared) from the array pointed to by *s1* to the array pointed to by *s2*.

The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the strings being compared.

RETURN VALUE

Upon successful completion, *strncmp*( ) shall return an integer greater than, equal to, or less than 0, if the possibly null-terminated array pointed to by *s1* is greater than, equal to, or less than the possibly null-terminated array pointed to by *s2* respectively.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strcmp( ), 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 4

The DESCRIPTION is changed to make it clear that the function manipulates bytes rather than (possibly multi-byte) characters.

The following change is incorporated for alignment with the ISO C standard:

- The type of arguments *s1* and *s2* are changed from **char*** to **const char***.

Issue 6

Extensions beyond the ISO C standard are now marked.

44492 **NAME**

44493 strncpy — copy part of a string

44494 **SYNOPSIS**

44495 `#include <string.h>`

44496 `char *strncpy(char *restrict s1, const char *restrict s2, size_t n);`

44497 **DESCRIPTION**

44498 CX The functionality described on this reference page is aligned with the ISO C standard. Any
44499 conflict between the requirements described here and the ISO C standard is unintentional. This
44500 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

44501 The *strncpy*( ) function shall copy not more than *n* bytes (bytes that follow a null byte are not
44502 copied) from the array pointed to by *s2* to the array pointed to by *s1*. If copying takes place
44503 between objects that overlap, the behavior is undefined.

44504 If the array pointed to by *s2* is a string that is shorter than *n* bytes, null bytes shall be appended
44505 to the copy in the array pointed to by *s1*, until *n* bytes in all are written.

44506 **RETURN VALUE**

44507 The *strncpy*( ) function shall return *s1*; no return value is reserved to indicate an error.

44508 **ERRORS**

44509 No errors are defined.

44510 **EXAMPLES**

44511 None.

44512 **APPLICATION USAGE**

44513 Character movement is performed differently in different implementations. Thus, overlapping
44514 moves may yield surprises.

44515 If there is no null byte in the first *n* bytes of the array pointed to by *s2*, the result is not null-
44516 terminated.

44517 **RATIONALE**

44518 None.

44519 **FUTURE DIRECTIONS**

44520 None.

44521 **SEE ALSO**

44522 *strcpy*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<string.h>**

44523 **CHANGE HISTORY**

44524 First released in Issue 1. Derived from Issue 1 of the SVID.

44525 **Issue 4**

44526 The DESCRIPTION is changed to make it clear that the function manipulates bytes rather than
44527 (possibly multi-byte) characters.

44528 The following change is incorporated for alignment with the ISO C standard:

44529 • The type of argument *s2* is changed from **char*** to **const char***.

44530 **Issue 6**

44531 The *strncpy*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

strpbrk — scan string for byte

SYNOPSIS

```
#include <string.h>

char *strpbrk(const char *s1, const char *s2);
```

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 *strpbrk*( ) function shall locate the first occurrence in the string pointed to by *s1* of any byte from the string pointed to by *s2*.

RETURN VALUE

Upon successful completion, *strpbrk*( ) shall return a pointer to the byte or a null pointer if no byte from *s2* occurs in *s1*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strchr( ), *strrchr*( ), 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 4

The DESCRIPTION and RETURN VALUE sections are changed to make it clear that the function works in units of bytes rather than (possibly multi-byte) characters.

The following change is incorporated for alignment with the ISO C standard:

- The type of arguments *s1* and *s2* is changed from **char*** to **const char***.

44565 **NAME**

44566 strptime — date and time conversion

44567 **SYNOPSIS**

44568 XSI
```
#include <time.h>

char *strptime(const char *restrict buf, const char *restrict format,
    struct tm *restrict tm);
```

44572 **DESCRIPTION**

44573 The *strptime*( ) function shall convert the character string pointed to by *buf* to values which are
44574 stored in the **tm** structure pointed to by *tm*, using the format specified by *format*.

44575 The *format* is composed of zero or more directives. Each directive is composed of one of the
44576 following: one or more white-space characters (as specified by *isspace*( )); an ordinary character
44577 (neither '%' nor a white-space character); or a conversion specification. Each conversion
44578 specification is composed of a '%' character followed by a conversion character which specifies
44579 the replacement required. The application shall ensure that there is white-space or other non-
44580 alphanumeric characters between any two conversion specifications. The following conversion
44581 specifications are supported:

44582 %*a* The day of the week, using the locale's weekday names; either the abbreviated or full
44583 name may be specified.

44584 %*A* The same as %*a*.

44585 %*b* The month, using the locale's month names; either the abbreviated or full name may be
44586 specified.

44587 %*B* The same as %*b*.

44588 %*c* Replaced by the locale's appropriate date and time representation.

44589 %*C* The century number [0,99]; leading zeros are permitted but not required.

44590 %*d* The day of the month [1,31]; leading zeros are permitted but not required.

44591 %*D* The date as %*m*/%*d*/%*y*.

44592 %*e* The same as %*d*.

44593 %*h* The same as %*b*.

44594 %*H* The hour (24-hour clock) [0,23]; leading zeros are permitted but not required.

44595 %*I* The hour (12-hour clock) [1,12]; leading zeros are permitted but not required.

44596 %*j* The day number of the year [1,366]; leading zeros are permitted but not required.

44597 %*m* The month number [1,12]; leading zeros are permitted but not required.

44598 %*M* The minute [0-59]; leading zeros are permitted but not required.

44599 %*n* Any white space.

44600 %*p* The locale's equivalent of a.m or p.m.

44601 %*r* 12-hour clock time using the AM/PM notation if **t_fmt_ampm** is not an empty string in
44602 the LC_TIME portion of the current locale; in the POSIX locale, this is equivalent to
44603 %*I*:%*M*:%*S* %*p*.

44604 %*R* The time as %*H*:%*M*.

%*S* The seconds [0,61]; leading zeros are permitted but not required.

%*t* Any white space.

%*T* The time as %*H*:%*M*:%*S*.

%*U* The week number of the year (Sunday as the first day of the week) as a decimal number [00,53]; leading zeros are permitted but not required.

%*w* The weekday as a decimal number [0,6], with 0 representing Sunday; leading zeros are permitted but not required.

%*W* The week number of the year (Monday as the first day of the week) as a decimal number [00,53]; leading zeros are permitted but not required.

%*x* The date, using the locale's date format.

%*X* The time, using the locale's time format.

%*y* The year within century. When a century is not otherwise specified, values in the range 69-99 refer to years in the twentieth century (1969 to 1999 inclusive); values in the range 00-68 refer to years in the twenty-first century (2000 to 2068 inclusive); leading zeros are permitted but not required.

%*Y* The year, including the century (for example, 1988).

%% Replaced by '%'.

Modified Directives

Some directives 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 directive. If the alternative format or specification does not exist in the current locale, the behavior shall be as if the unmodified directive were used.

%*Ec* The locale's alternative appropriate date and time representation.

%*EC* The name of the base year (period) in the locale's alternative representation.

%*Ex* The locale's alternative date representation.

%*EX* The locale's alternative time representation.

%*Ey* The offset from %*EC* (year only) in the locale's alternative representation.

%*EY* The full alternative year representation.

%*Od* The day of the month using the locale's alternative numeric symbols; leading zeros are permitted but not required.

%*Oe* The same as %*Od*.

%*OH* The hour (24-hour clock) using the locale's alternative numeric symbols.

%*OI* The hour (12-hour clock) using the locale's alternative numeric symbols.

%*Om* The month using the locale's alternative numeric symbols.

%*OM* The minutes using the locale's alternative numeric symbols.

%*OS* The seconds using the locale's alternative numeric symbols.

%*OU* The week number of the year (Sunday as the first day of the week) using the locale's alternative numeric symbols.

%*Ow* The number of the weekday (Sunday=0) using the locale's alternative numeric symbols.

%*OW* The week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.

%*Oy* The year (offset from %*C*) using the locale's alternative numeric symbols.

A directive 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.

A directive 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.

A series of directives 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.

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 *buf* such as month or weekday names. If no match is found, *strptime*( ) fails and no more characters are scanned.

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

None.

APPLICATION USAGE

Several ''same as'' formats and the special processing of white-space characters are provided in order to ease the use of identical *format* strings for *strftime*( ) and *strptime*( ).

Applications should use %*Y* (4-digit years) 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. Portable 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( ), *strftime*( ), *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 are clarified in the DESCRIPTION.

Issue 6

The Open Group corrigenda item U033/5 has been applied. The %*r* specifier description is reworded.

The DESCRIPTION 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 corrigenda item U047/2 has been applied.

NAME

strrchr — string scanning operation

SYNOPSIS

```
#include <string.h>

char *strrchr(const char *s, int c);
```

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.

CX The *strrchr*( ) function shall locate the last occurrence of *c* (converted to an **unsigned char**) in the string pointed to by *s*. The terminating null byte is considered to be part of the string.

RETURN VALUE

Upon successful completion, *strrchr*( ) shall return a pointer to the byte or a null pointer if *c* does not occur in the string.

ERRORS

No errors are defined.

EXAMPLES

Finding the Base Name of a File

The following example uses *strrchr*( ) to get a pointer to the base name of a file. The *strrchr*( ) function searches backwards through the name of the file to find the last '/' character in *name*. This pointer (plus one) will point to the base name of the file.

```
#include <string.h>
...
const char *name;
char *basename;
...
basename = strrchr(name, '/') + 1;
...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strchr( ), 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 4

The DESCRIPTION and RETURN VALUE sections are changed to make it clear that the function works in units of bytes rather than (possibly multi-byte) characters.

The following change is incorporated for alignment with the ISO C standard:

44740 • The type of argument *s* is changed from **char*** to **const char***.

NAME

strspn — get length of a substring

SYNOPSIS

```
#include <string.h>

size_t strspn(const char *s1, const char *s2);
```

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 *strspn*( ) function shall compute the length of the maximum initial segment of the string pointed to by *s1* which consists entirely of bytes from the string pointed to by *s2*.

RETURN VALUE

The *strspn*( ) function shall return the length of *s1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strcspn( ), 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 4

The DESCRIPTION is changed to make it clear that the function works in units of bytes rather than (possibly multi-byte) characters.

The following change is incorporated for alignment with the ISO C standard:

- The type of arguments *s1* and *s2* are changed from **char*** to **const char***.

Issue 5

The RETURN VALUE section is updated to indicate that *strspn*( ) returns the length of *s*, and not *s* itself as was previously stated.

NAME

strstr — find a substring

SYNOPSIS

```
#include <string.h>

char *strstr(const char *s1, const char *s2);
```

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 *strstr*( ) function shall locate the first occurrence in the string pointed to by *s1* of the sequence of bytes (excluding the terminating null byte) in the string pointed to by *s2*.

RETURN VALUE

Upon successful completion, *strstr*( ) shall return a pointer to the located string or a null pointer if the string is not found.

If *s2* points to a string with zero length, the function shall return *s1*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strchr( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the ANSI C standard.

Issue 4

The DESCRIPTION is changed to make it clear that the function works in units of bytes rather than (possibly multi-byte) characters.

The following change is incorporated for alignment with the ISO C standard:

- The type of arguments *s1* and *s2* are changed from **char*** to **const char***.

NAME

strtod, strtof, strtold — convert string to a double-precision number

SYNOPSIS

```
#include <stdlib.h>

double strtod(const char *restrict nptr, char **restrict endptr);
float strtof(const char *restrict nptr, char **restrict endptr);
long double strtold(const char *restrict nptr, char **restrict endptr);
```

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 convert the initial portion of the string pointed to by *nptr* to **double**, **float**, and **long double** representation, respectively. First, they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace*( ))
2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string

Then it attempts to convert the subject sequence to a floating-point number, and returns the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character, then an optional exponent part
- A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix character, then an optional binary exponent part
- One of INF or INFINITY, ignoring case
- One of NAN or NAN(*n-char-sequence*$_{opt}$), *ignoring case in the NAN part, where:*

```
n-char-sequence:
    digit
    nondigit
    n-char-sequence digit
    n-char-sequence nondigit
```

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant of the C language, except that the radix character is used in place of a period, and that if neither an exponent part nor a radix character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated. A character sequence INF or INFINITY is interpreted as an infinity, if

representable in the return type, else like a floating constant that is too large for the range of the return type. A character sequence NAN or NAN(*n-char-sequence*$_{opt}$*), is interpreted as a quiet NaN, if supported in the return type, else like a subject sequence part that does not have the expected form; the meaning of the n*-char sequences is implementation-defined. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence has the hexadecimal form and FLT_RADIX is a power of 2, the value resulting from the conversion is correctly rounded.

CX The radix character is defined in the program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period ('.').

CX In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

The *strtod*( ) function shall not change the setting of *errno* if successful.

Because 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set *errno* to 0, then call *strtod*( ), then check *errno*.

RETURN VALUE

Upon successful completion, these functions shall return the converted value. If no conversion could be performed, 0 shall be returned, and *errno* may be set to [EINVAL].

If the correct value is outside the range of representable values, HUGE_VAL, HUGE_VALF, or HUGE_VALL shall be returned (according to the sign of the value), and *errno* shall be set to [ERANGE].

If the correct value would cause an underflow, a value whose magnitude is no greater than the smallest normalized positive number in the return type shall be returned and *errno* set to [ERANGE].

ERRORS

The *strtod*( ) function shall fail if:

CX [ERANGE] The value to be returned would cause overflow or underflow.

The *strtod*( ) function may fail if:

CX [EINVAL] No conversion could be performed.

Notes to Reviewers

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

There is a query outstanding over the question of [EINVAL] being an allowable extension to C99. This error may be removed in a future draft.

EXAMPLES

None.

APPLICATION USAGE

If the subject sequence has the hexadecimal form and FLT_RADIX is not a power of 2, 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.

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 <= D <= 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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isspace( ), *localeconv*( ), *scanf*( ), *setlocale*( ), *strtol*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<float.h>**, **<stdlib.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

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

Issue 4

The DESCRIPTION is changed to make it clear when the function manipulates bytes and when it manipulates characters.

The [EINVAL] error is added to the ERRORS section and marked as an extension.

The following changes are incorporated for alignment with the ISO C standard:

- The function is no longer marked as an extension.
- The type of argument *str* is changed from **char*** to **const char***.
- The name of the second argument is changed from *ptr* to *endptr*.
- The precise conditions under which the [ERANGE] error can be set have been defined in the RETURN VALUE section.

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 now 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.

NAME

strtoimax, strtoumax — convert string to integer type

SYNOPSIS

```
#include <inttypes.h>

intmax_t strtoimax(const char *restrict nptr, char **restrict endptr,
    int base);
uintmax_t strtoumax(const char *restrict nptr, char **restrict endptr,
    int base);
```

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 be equivalent to the *strtol*( ), *strtoll*( ), *strtoul*( ), and *strtoull*( ) functions, except that the initial portion of the string shall be converted to **intmax_t** and **uintmax_t** representation, respectively.

RETURN VALUE

These functions shall return the converted value, if any.

If no conversion could be performed, zero shall be returned.

If the correct value is outside the range of representable values, {INTMAX_MAX}, {INTMAX_MIN}, or {UINTMAX_MAX} shall be returned (according to the return type and sign of the value, if any), and *errno* shall be set to [ERANGE].

ERRORS

These functions shall fail if:

[ERANGE] The value to be returned is not representable.

These functions may fail if:

[EINVAL] The value of *base* is not supported.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strtol( ), *strtoul*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<inttypes.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

strtok, strtok_r — split string into tokens

SYNOPSIS

```
#include <string.h>

char *strtok(char *restrict s1, const char *restrict s2);
TSF char *strtok_r(char *restrict s, const char *restrict sep,
        char **restrict lasts);
```

DESCRIPTION

CX For *strtok*( ): 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.

A sequence of calls to *strtok*( ) breaks the string pointed to by *s1* into a sequence of tokens, each of which is delimited by a byte from the string pointed to by *s2*. The first call in the sequence has *s1* as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by *s2* may be different from call to call.

The first call in the sequence searches the string pointed to by *s1* for the first byte that is *not* contained in the current separator string pointed to by *s2*. If no such byte is found, then there are no tokens in the string pointed to by *s1* and *strtok*( ) returns a null pointer. If such a byte is found, it is the start of the first token.

The *strtok*( ) function then searches from there for a byte that *is* contained in the current separator string. If no such byte is found, the current token extends to the end of the string pointed to by *s1*, and subsequent searches for a token shall return a null pointer. If such a byte is found, it is overwritten by a null byte, which terminates the current token. The *strtok*( ) function saves a pointer to the following byte, from which the next search for a token shall start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The implementation shall behave as if no function defined in this volume of IEEE Std. 1003.1-200x calls *strtok*( ).

CX The *strtok*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *strtok_r*( ) function considers the null-terminated string *s* as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *sep*. The argument *lasts* points to a user-provided pointer which points to stored information necessary for *strtok_r*( ) to continue scanning the same string.

In the first call to *strtok_r*( ), *s* points to a null-terminated string, *sep* to a null-terminated string of separator characters, and the value pointed to by *lasts* is ignored. The *strtok_r*( ) function returns a pointer to the first character of the first token, writes a null character into *s* immediately following the returned token, and updates the pointer to which *lasts* points.

In subsequent calls, *s* is a NULL pointer and *lasts* shall be unchanged from the previous call so that subsequent calls shall move through the string *s*, returning successive tokens until no tokens remain. The separator string *sep* may be different from call to call. When no token remains in *s*, a NULL pointer is returned.

RETURN VALUE

Upon successful completion, *strtok*( ) shall return a pointer to the first byte of a token. Otherwise, if there is no token, *strtok*( ) shall return a null pointer.

TSF The *strtok_r*( ) function shall return a pointer to the token found, or a NULL pointer when no token is found.

ERRORS

No errors are defined.

EXAMPLES

Searching for Word Separators

The following example searches for tokens separated by space characters.

```
#include <string.h>
...
char *token;
char *line = "LINE TO BE SEPARATED";
char *search = " ";

/* Token will point to "LINE". */
token = strtok(line, search);

/* Token will point to "TO". */
token = strtok(NULL, search);
```

Breaking a Line

The following example uses *strtok*( ) to break a line into two character strings separated by any combination of <space>s, <tab>s, or <newline>s.

```
#include <string.h>
...
struct element {
    char *key;
    char *data;
};
...
char line[LINE_MAX];
char *key, *data;
...
key = strtok(line, " \n");
data = strtok(NULL, " \n");
...
```

APPLICATION USAGE

The *strtok_r*( ) function is thread-safe and stores its state in a user-supplied buffer instead of possibly using a static data area that may be overwritten by an unrelated call from another thread.

RATIONALE

The *strtok*( ) function searches for a separator string within a larger string. It returns a pointer to the last substring between separator strings. This function uses static storage to keep track of the current string position between calls. The new function, *strtok_r*( ), takes an additional argument, *lasts*, to keep track of the current position in the string.

FUTURE DIRECTIONS

None.

SEE ALSO

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 4

The DESCRIPTION is changed to make it clear that the function manipulates bytes rather than (possibly multi-byte) characters.

The following changes are incorporated for alignment with the ISO C standard:

- The function is no longer marked as an extension.
- The type of argument *s2* is changed from **char*** to **const char***.

Issue 5

The *strtok_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *strtok*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

Extensions beyond the ISO C standard are now marked.

The *strtok_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.

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.

The **restrict** keyword is added to the *strtok*( ) and *strtok_r*( ) prototypes for alignment with the ISO/IEC 9899:1999 standard.

45087 **NAME**

45088 strtol, strtoll — convert string to a long integer

45089 **SYNOPSIS**

45090 `#include <stdlib.h>`

45091 `long strtol(const char *restrict str, char **restrict endptr, int base);`
45092 `long long strtoll(const char *restrict str, char **restrict endptr,`
45093 `int base)`

45094 **DESCRIPTION**

45095 CX The functionality described on this reference page is aligned with the ISO C standard. Any
45096 conflict between the requirements described here and the ISO C standard is unintentional. This
45097 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

45098 These functions shall convert the initial portion of the string pointed to by *str* to a type **long** and
45099 **long long** representation, respectively. First, they decompose the input string into three parts:

45100 1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace*( ))

45101 2. A subject sequence interpreted as an integer represented in some radix determined by the
45102 value of *base*

45103 3. A final string of one or more unrecognized characters, including the terminating null byte
45104 of the input string.

45105 Then it attempts to convert the subject sequence to an integer, and returns the result.

45106 If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant,
45107 octal constant, or hexadecimal constant, any of which may be preceded by a `'+'` or `'−'` sign. A
45108 decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An
45109 octal constant consists of the prefix `'0'` optionally followed by a sequence of the digits `'0'` to
45110 `'7'` only. A hexadecimal constant consists of the prefix `"0x"` or `"0X"` followed by a sequence of
45111 the decimal digits and letters `'a'` (or `'A'`) to `'f'` (or `'F'`) with values 10 to 15 respectively.

45112 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
45113 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
45114 by a `'+'` or `'−'` sign. The letters from `'a'` (or `'A'`) to `'z'` (or `'Z'`) inclusive are ascribed the
45115 values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the
45116 value of *base* is 16, the characters `"0x"` or `"0X"` may optionally precede the sequence of letters
45117 and digits, following the sign if present.

45118 The subject sequence is defined as the longest initial subsequence of the input string, starting
45119 with the first non-white-space character that is of the expected form. The subject sequence
45120 contains no characters if the input string is empty or consists entirely of white-space characters,
45121 or if the first non-white-space character is other than a sign or a permissible letter or digit.

45122 If the subject sequence has the expected form and the value of *base* is 0, the sequence of
45123 characters starting with the first digit is interpreted as an integer constant. If the subject
45124 sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base
45125 for conversion, ascribing to each letter its value as given above. If the subject sequence begins
45126 with a minus sign, the value resulting from the conversion is negated. A pointer to the final
45127 string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

45128 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
45129 accepted.

45130 If the subject sequence is empty or does not have the expected form, no conversion is performed;
45131 the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null

pointer.

The *strtol*( ) function shall not change the setting of *errno* if successful.

Because 0, {LONG_MIN} or {LLONG_MIN}, and {LONG_MAX} or {LLONG_MAX} are returned on error and are also valid returns on success, an application wishing to check for error situations should set *errno* to 0, then call *strtol*( ), then check *errno*.

RETURN VALUE

Upon successful completion, these functions shall return the converted value, if any. If no CX conversion could be performed, 0 shall be returned and *errno* may be set to [EINVAL].

If the correct value is outside the range of representable values, {LONG_MIN}, {LONG_MAX}, {LLONG_MIN} or {LLONG_MAX} shall be returned (according to the sign of the value), and *errno* set to [ERANGE].

ERRORS

The *strtol*( ) function shall fail if:

[ERANGE] The value to be returned is not representable.

The *strtol*( ) function may fail if:

CX [EINVAL] The value of *base* is not supported.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalpha( ), *scanf*( ), *strtod*( ), 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 4

The DESCRIPTION is changed to make it clear when the function manipulates bytes and when it manipulates characters.

In the RETURN VALUE section, text indicating that *errno* is set when 0 is returned is marked as an extension.

The ERRORS section is updated in line with the RETURN VALUE section.

The following changes are incorporated for alignment with the ISO C standard:

- The function is no longer marked as an extension.
- The type of argument *str* is changed from **char*** to **const char***.
- The name of the second argument is changed from *ptr* to *endptr*.
- The DESCRIPTION is changed to indicate permitted forms of the subject sequence when *base* is 0.

- The RETURN VALUE section is changed to indicate that {LONG_MAX} or {LONG_MIN} is returned if the converted value is too large or too small.

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 now 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 *strtol*( ) prototype is updated.
- The *strtoll*( ) function is added.

NAME

strtoul, strtoull — convert string to an unsigned long

SYNOPSIS

```
#include <stdlib.h>

long strtoul(const char *restrict str, char **restrict endptr, int base);
long long strtoull(const char*restrict str, char **restrict endptr,
    int base);
```

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 convert the initial portion of the string pointed to by *str* to a type **long** and **long long** representation, respectively. First, they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace*( ))
2. A subject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string

Then it attempts to convert the subject sequence to an unsigned integer, and returns the result.

If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant, any of which may be preceded by a '+' or '−' sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7' only. A hexadecimal constant consists of the prefix "0x" or "0X" followed by a sequence of the decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a '+' or '−' sign. The letters from 'a' (or 'A') to 'z' (or 'Z') inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters "0x" or "0X" may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white-space characters, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is 0, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

CX In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null

pointer.

The *strtoul*( ) function shall not change the setting of *errno* if successful.

Because 0, {ULONG_MAX}, and {ULLONG_MAX} are returned on error and are also valid returns on success, an application wishing to check for error situations should set *errno* to 0, then call *strtoul*( ), then check *errno*.

RETURN VALUE

Upon successful completion, these functions shall return the converted value, if any. If no conversion could be performed, 0 shall be returned and *errno* may be set to [EINVAL]. If the correct value is outside the range of representable values, {ULONG_MAX} or {ULLONG_MAX} shall be returned and *errno* set to [ERANGE].

ERRORS

The *strtoul*( ) function shall fail if:

CX [EINVAL] The value of *base* is not supported.

[ERANGE] The value to be returned is not representable.

The *strtoul*( ) function may fail if:

CX [EINVAL] No conversion could be performed.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalpha( ), *scanf*( ), *strtod*( ), *strtol*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

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 now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EINVAL] error condition is added for when the value of *base* is not supported.

  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 *strtoul*( ) prototype is updated.

45270 • The *strtoull*( ) function is added.

45271 **NAME**

45272 strxfrm — string transformation

45273 **SYNOPSIS**

45274 `#include <string.h>`

45275 `size_t strxfrm(char *restrict s1, const char *restrict s2, size_t n);`

45276 **DESCRIPTION**

45277 CX The functionality described on this reference page is aligned with the ISO C standard. Any
45278 conflict between the requirements described here and the ISO C standard is unintentional. This
45279 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

45280 The *strxfrm*( ) function shall transform the string pointed to by *s2* and place the resulting string
45281 into the array pointed to by *s1*. The transformation is such that if *strcmp*( ) is applied to two
45282 transformed strings, it returns a value greater than, equal to, or less than 0, corresponding to the
45283 result of *strcoll*( ) applied to the same two original strings. No more than *n* bytes are placed into
45284 the resulting array pointed to by *s1*, including the terminating null byte. If *n* is 0, *s1* is permitted
45285 to be a null pointer. If copying takes place between objects that overlap, the behavior is
45286 undefined.

45287 CX The *strxfrm*( ) function shall not change the setting of *errno* if successful.

45288 Because no return value is reserved to indicate an error, an application wishing to check for error
45289 situations should set *errno* to 0, then call *strcoll*( ), then check *errno*.

45290 **RETURN VALUE**

45291 Upon successful completion, *strxfrm*( ) shall return the length of the transformed string (not
45292 including the terminating null byte). If the value returned is *n* or more, the contents of the array
45293 pointed to by *s1* are indeterminate.

45294 CX On error, *strxfrm*( ) may set *errno* but no return value is reserved to indicate an error.

45295 **ERRORS**

45296 The *strxfrm*( ) function may fail if:

45297 CX [EINVAL] The string pointed to by the *s2* argument contains characters outside the
45298 domain of the collating sequence.

45299 **EXAMPLES**

45300 None.

45301 **APPLICATION USAGE**

45302 The transformation function is such that two transformed strings can be ordered by *strcmp*( ) as
45303 appropriate to collating sequence information in the program's locale (category *LC_COLLATE*).

45304 The fact that when *n* is 0 *s1* is permitted to be a null pointer is useful to determine the size of the
45305 *s1* array prior to making the transformation.

45306 **RATIONALE**

45307 None.

45308 **FUTURE DIRECTIONS**

45309 None.

45310 **SEE ALSO**

45311 *strcmp*( ), *strcoll*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<string.h>**

45312 **CHANGE HISTORY**

45313 First released in Issue 3.

45314 Entry included for alignment with the ISO C standard.

45315 **Issue 4**

45316 The DESCRIPTION is changed to make it clear when the function manipulates byte values and
45317 when it manipulates characters.

45318 The sentence describing error returns in the RETURN VALUE section is marked as an extension,
45319 as is the [EINVAL] error.

45320 The APPLICATION USAGE section is expanded.

45321 The following changes are incorporated for alignment with the ISO C standard:

45322 • The function is no longer marked as an extension.

45323 • The type of argument *s2* is changed from **char*** to **const char***.

45324 **Issue 5**

45325 The DESCRIPTION is updated to indicate that *errno* does not changed if the function is
45326 successful.

45327 **Issue 6**

45328 Extensions beyond the ISO C standard are now marked.

45329 The following new requirements on POSIX implementations derive from alignment with the
45330 Single UNIX Specification:

45331 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
45332 added if no conversion could be performed.

45333 The *strxfrm*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard. |

45334 **NAME**

45335 swab — swap bytes

45336 **SYNOPSIS**

45337 XSI
```
#include <unistd.h>

void swab(const void *restrict src, void *restrict dest,
    ssize_t nbytes);
```
45340

45341 **DESCRIPTION**

45342 The *swab*( ) function shall copy *nbytes* bytes, which are pointed to by *src*, to the object pointed to
45343 by *dest*, exchanging adjacent bytes. The *nbytes* argument should be even. If *nbytes* is odd, *swab*( )
45344 copies and exchanges *nbytes*−1 bytes and the disposition of the last byte is unspecified. If
45345 copying takes place between objects that overlap, the behavior is undefined. If *nbytes* is
45346 negative, *swab*( ) does nothing.

45347 **RETURN VALUE**

45348 None.

45349 **ERRORS**

45350 No errors are defined.

45351 **EXAMPLES**

45352 None.

45353 **APPLICATION USAGE**

45354 None.

45355 **RATIONALE**

45356 None.

45357 **FUTURE DIRECTIONS**

45358 None.

45359 **SEE ALSO**

45360 The Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>**

45361 **CHANGE HISTORY**

45362 First released in Issue 1. Derived from Issue 1 of the SVID.

45363 **Issue 4**

45364 The **<unistd.h>** header is added to the SYNOPSIS section.

45365 The type of argument *src* is changed from **char*** to **const void***, *dest* is changed from **char*** to
45366 **void***, and *nbytes* is changed from **int** to **ssize_t**.

45367 The DESCRIPTION is changed as follows:

45368 - States explicitly that copying between overlapping objects results in undefined behavior.
45369 - Takes account of the type change to *nbyte*; that is, previously it was defined as **int** and could
45370 be positive or negative, whereas now it is defined as an **unsigned** type.
45371 - A statement about overlapping objects is added.

45372 The APPLICATION USAGE section is removed.

Issue 6

The **restrict** keyword is added to the *swab*() prototype for alignment with the ISO/IEC 9899:1999 standard.

45376 **NAME**

45377 swapcontext — swap user context

45378 **SYNOPSIS**

45379 XSI
```
#include <ucontext.h>

int swapcontext(ucontext_t *restrict oucp,
    const ucontext_t *restrict ucp);
```
45380 |
45381 |
45382 |

45383 **DESCRIPTION**

45384 Refer to *makecontext*( ).

45385 **NAME**

45386 swprintf — print formatted wide-character output

45387 **SYNOPSIS**

45388 `#include <stdio.h>`

45389 `#include <wchar.h>`

45390 `int swprintf(wchar_t *ws, size_t n, const wchar_t *format, ...);` |

45391 **DESCRIPTION** |

45392 Refer to *fwprintf*( ).

45393 **NAME**

45394 swscanf — convert formatted wide-character input

45395 **SYNOPSIS**

45396 `#include <stdio.h>`

45397 `#include <wchar.h>`

45398 `int swscanf(const wchar_t *ws, const wchar_t *format, ... );` |

45399 **DESCRIPTION** |

45400 Refer to *fwscanf*( ).

45401 **NAME**

45402 symlink — make symbolic link to a file

45403 **SYNOPSIS**

45404 `#include <unistd.h>`

45405 `int symlink(const char *`*`path1`*`, const char *`*`path2`*`);`

45406 **DESCRIPTION**

45407 The *symlink*( ) function shall create a symbolic link called *path2* that contains the string pointed
45408 to by *path1* (*path2* is the name of the symbolic link created, *path1* is the string contained in the
45409 symbolic link).

45410 The string pointed to by *path1* shall be treated only as a character string and shall not be
45411 validated as a path name.

45412 If the *symlink*( ) function fails for any reason other than [EIO], any file named by *path2* shall be |
45413 unaffected.

45414 **RETURN VALUE**

45415 Upon successful completion, *symlink*( ) shall return 0; otherwise, it shall return –1 and set *errno* to
45416 indicate the error.

45417 **ERRORS**

45418 The *symlink*( ) function shall fail if:

45419 [EACCES] Write permission is denied in the directory where the symbolic link is being |
45420 created, or search permission is denied for a component of the path prefix of
45421 *path2*.

45422 [EEXIST] The *path2* argument names an existing file or symbolic link. |

45423 [EIO] An I/O error occurs while reading from or writing to the file system. |

45424 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path2* |
45425 argument.

45426 [ENAMETOOLONG] |
45427 The length of the *path2* argument exceeds {PATH_MAX} or a path name |
45428 component is longer than {NAME_MAX} or the length of the *path1* argument |
45429 is longer than {SYMLINK_MAX}. |

45430 [ENOENT] A component of *path2* does not name an existing file or *path2* is an empty |
45431 string.

45432 [ENOSPC] The directory in which the entry for the new symbolic link is being placed |
45433 cannot be extended because no space is left on the file system containing the
45434 directory, or the new symbolic link cannot be created because no space is left
45435 on the file system which shall contain the link, or the file system is out of file-
45436 allocation resources.

45437 [ENOTDIR] A component of the path prefix of *path2* is not a directory. |

45438 [EROFS] The new symbolic link would reside on a read-only file system. |

45439 The *symlink*( ) function may fail if:

45440 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
45441 resolution of the *path2* argument.

45442 [ENAMETOOLONG] |
45443 As a result of encountering a symbolic link in resolution of the *path2*

argument, the length of the substituted path name string exceeded {PATH_MAX} bytes (including the terminating null byte), or the length of the string pointed to by *path1* exceeded {SYMLINK_MAX}.

EXAMPLES

None.

APPLICATION USAGE

Like a hard link, a symbolic link allows a file to have multiple logical names. The presence of a hard link guarantees the existence of a file, even after the original name has been removed. A symbolic link provides no such assurance; in fact, the file named by the *path1* argument need not exist when the link is created. A symbolic link can cross file system boundaries.

Normal permission checks are made on each component of the symbolic link path name during its resolution.

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 *symlink*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

lchown( ), *link*( ), *lstat*( ), *open*( ), *readlink*( ), *unlink*( ), 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 are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The DESCRIPTION text is updated.
- The [ELOOP] optional error condition is added.

45475 **NAME**

45476 sync — schedule file system updates

45477 **SYNOPSIS**

45478 XSI
```
#include <unistd.h>
```

45479
```
void sync(void);
```

45480

45481 **DESCRIPTION**

45482 The *sync*( ) function shall cause all information in memory that updates file systems to be
45483 scheduled for writing out to all file systems.

45484 The writing, although scheduled, is not necessarily complete upon return from *sync*( ).

45485 **RETURN VALUE**

45486 The *sync*( ) function shall return no value.

45487 **ERRORS**

45488 No errors are defined.

45489 **EXAMPLES**

45490 None.

45491 **APPLICATION USAGE**

45492 None.

45493 **RATIONALE**

45494 None.

45495 **FUTURE DIRECTIONS**

45496 None.

45497 **SEE ALSO**

45498 *fsync*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>**

45499 **CHANGE HISTORY**

45500 First released in Issue 4, Version 2.

45501 **Issue 5**

45502 Moved from X/OPEN UNIX extension to BASE.

45503 **NAME**

45504 sysconf — get configurable system variables

45505 **SYNOPSIS**

45506 `#include <unistd.h>`

45507 `long sysconf(int name);`

45508 **DESCRIPTION**

45509 The *sysconf*( ) function provides a method for the application to determine the current value of a
45510 configurable system limit or option (*variable*). Support for some system variables is dependent
45511 on implementation options (as indicated by the margin codes in the following table). Where an
45512 implementation option is not supported, the variable need not be supported.

45513 The *name* argument represents the system variable to be queried. The following table lists the
45514 minimal set of system variables from **<limits.h>** or **<unistd.h>** that can be returned by *sysconf*( ),
45515 and the symbolic constants, defined in **<unistd.h>** that are the corresponding values used for
45516 *name*. Support for some configuration variables is dependent on implementation options (see
45517 shading and margin codes in the table below). Where an implementation option is not
45518 supported, the variable need not be supported.

45519

45520

	Variable	Value of Name
45521 AIO	{AIO_LISTIO_MAX}	_SC_AIO_LISTIO_MAX
45522	{AIO_MAX}	_SC_AIO_MAX
45523	{AIO_PRIO_DELTA_MAX}	_SC_AIO_PRIO_DELTA_MAX
45524	{ARG_MAX}	_SC_ARG_MAX
45525 XSI	{ATEXIT_MAX}	_SC_ATEXIT_MAX
45526	{BC_BASE_MAX}	_SC_BC_BASE_MAX
45527	{BC_DIM_MAX}	_SC_BC_DIM_MAX
45528	{BC_SCALE_MAX}	_SC_BC_SCALE_MAX
45529	{BC_STRING_MAX}	_SC_BC_STRING_MAX
45530	{CHILD_MAX}	_SC_CHILD_MAX
45531	Clock ticks/second	_SC_CLK_TCK
45532	{COLL_WEIGHTS_MAX}	_SC_COLL_WEIGHTS_MAX
45533 XSI	{DELAYTIMER_MAX}	_SC_DELAYTIMER_MAX
45534	{EXPR_NEST_MAX}	_SC_EXPR_NEST_MAX
45535 XSI	{IOV_MAX}	_SC_IOV_MAX
45536	{LINE_MAX}	_SC_LINE_MAX
45537	{LOGIN_NAME_MAX}	_SC_LOGIN_NAME_MAX
45538	{NGROUPS_MAX}	_SC_NGROUPS_MAX
45539 TSF	Maximum size of *getgrgid_r*( ) and *getgrnam_r*( ) data buffers	_SC_GETGR_R_SIZE_MAX
45541	Maximum size of *getpwuid_r*( ) and *getpwnam_r*( ) data buffers	_SC_GETPW_R_SIZE_MAX
45543 MSG	{MQ_OPEN_MAX}	_SC_MQ_OPEN_MAX
45544	{MQ_PRIO_MAX}	_SC_MQ_PRIO_MAX
45545	{OPEN_MAX}	_SC_OPEN_MAX
45546 ADV	_POSIX_ADVISORY_INFO	_SC_ADVISORY_INFO
45547 BAR	_POSIX_BARRIERS	_SC_BARRIERS
45548 AIO	_POSIX_ASYNCHRONOUS_IO	_SC_ASYNCHRONOUS_IO

	Variable	Value of Name
	_POSIX_BASE	_SC_BASE
	_POSIX_C_LANG_SUPPORT	_SC_C_LANG_SUPPORT
	_POSIX_C_LANG_SUPPORT_R	_SC_C_LANG_SUPPORT_R
CS	_POSIX_CLOCK_SELECTION	_SC_CLOCK_SELECTION
CPT	_POSIX_CPUTIME	_SC_CPUTIME
	_POSIX_DEVICE_IO	_SC_DEVICE_IO
	_POSIX_DEVICE_SPECIFIC	_SC_DEVICE_SPECIFIC
	_POSIX_DEVICE_SPECIFIC_R	_SC_DEVICE_SPECIFIC_R
	_POSIX_FD_MGMT	_SC_FD_MGMT
	_POSIX_FIFO	_SC_FIFO
	_POSIX_FILE_ATTRIBUTES	_SC_FILE_ATTRIBUTES
	_POSIX_FILE_LOCKING	_SC_FILE_LOCKING
	_POSIX_FILE_SYSTEM	_SC_FILE_SYSTEM
FSC	_POSIX_FSYNC	_SC_FSYNC
	_POSIX_JOB_CONTROL	_SC_JOB_CONTROL
MF	_POSIX_MAPPED_FILES	_SC_MAPPED_FILES
ML	_POSIX_MEMLOCK	_SC_MEMLOCK
MLR	_POSIX_MEMLOCK_RANGE	_SC_MEMLOCK_RANGE
MPR	_POSIX_MEMORY_PROTECTION	_SC_MEMORY_PROTECTION
MSG	_POSIX_MESSAGE_PASSING	_SC_MESSAGE_PASSING
MON	_POSIX_MONOTONIC_CLOCK	_SC_MONOTONIC_CLOCK
	_POSIX_MULTIPLE_PROCESS	_SC_MULTIPLE_PROCESS
	_POSIX_NETWORKING	_SC_NETWORKING
	_POSIX_PIPE	_SC_PIPE
PIO	_POSIX_PRIORITIZED_IO	_SC_PRIORITIZED_IO
PS	_POSIX_PRIORITY_SCHEDULING	_SC_PRIORITY_SCHEDULING
THR	_POSIX_READER_WRITER_LOCKS	_SC_READER_WRITER_LOCKS
RTS	_POSIX_REALTIME_SIGNALS	_SC_REALTIME_SIGNALS
	_POSIX_REGEXP	_SC_REGEXP
	_POSIX_SAVED_IDS	_SC_SAVED_IDS
SEM	_POSIX_SEMAPHORES	_SC_SEMAPHORES
SHM	_POSIX_SHARED_MEMORY_OBJECTS	_SC_SHARED_MEMORY_OBJECTS
	_POSIX_SHELL	_SC_SHELL
	_POSIX_SIGNALS	_SC_SIGNALS
	_POSIX_SINGLE_PROCESS	_SC_SINGLE_PROCESS
SPN	_POSIX_SPAWN	_SC_SPAWN
SPI	_POSIX_SPIN_LOCKS	_SC_SPIN_LOCKS
SS	_POSIX_SPORADIC_SERVER	_SC_SPORADIC_SERVER
SIO	_POSIX_SYNCHRONIZED_IO	_SC_SYNCHRONIZED_IO
	_POSIX_SYSTEM_DATABASE	_SC_SYSTEM_DATABASE
	_POSIX_SYSTEM_DATABASE_R	_SC_SYSTEM_DATABASE_R

	Variable	Value of Name
TSA	_POSIX_THREAD_ATTR_STACKADDR	_SC_THREAD_ATTR_STACKADDR
TSS	_POSIX_THREAD_ATTR_STACKSIZE	_SC_THREAD_ATTR_STACKSIZE
TCT	_POSIX_THREAD_CPUTIME	_SC_THREAD_CPUTIME
TPI	_POSIX_THREAD_PRIO_INHERIT	_SC_THREAD_PRIO_INHERIT
TPP	_POSIX_THREAD_PRIO_PROTECT	_SC_THREAD_PRIO_PROTECT
TPS	_POSIX_THREAD_PRIORITY_SCHEDULING	_SC_THREAD_PRIORITY_SCHEDULING
TSH	_POSIX_THREAD_PROCESS_SHARED	_SC_THREAD_PROCESS_SHARED
TSF	_POSIX_THREAD_SAFE_FUNCTIONS	_SC_THREAD_SAFE_FUNCTIONS
TSP	_POSIX_THREAD_SPORADIC_SERVER	_SC_THREAD_SPORADIC_SERVER
THR	_POSIX_THREADS	_SC_THREADS
TMO	_POSIX_TIMEOUTS	_SC_TIMEOUTS
TMR	_POSIX_TIMERS	_SC_TIMERS
TRC	_POSIX_TRACE	_SC_TRACE
TEF	_POSIX_TRACE_EVENT_FILTER	_SC_TRACE_EVENT_FILTER
TRI	_POSIX_TRACE_INHERIT	_SC_TRACE_INHERIT
TRL	_POSIX_TRACE_LOG	_SC_TRACE_LOG
TYM	_POSIX_TYPED_MEMORY_OBJECTS	_SC_TYPED_MEMORY_OBJECTS
	_POSIX_USER_GROUPS	_SC_USER_GROUPS
	_POSIX_USER_GROUPS_R	_SC_USER_GROUPS_R
	_POSIX_VERSION	_SC_VERSION
	_POSIX_V6_ILP32_OFF32	_SC_V6_ILP32_OFF32
	_POSIX_V6_ILP32_OFFBIG	_SC_V6_ILP32_OFFBIG
	_POSIX_V6_LP64_OFF64	_SC_V6_LP64_OFF64
	_POSIX_V6_LPBIG_OFFBIG	_SC_V6_LPBIG_OFFBIG
	_POSIX2_C_BIND	_SC_2_C_BIND
	_POSIX2_C_DEV	_SC_2_C_DEV
	_POSIX2_C_VERSION	_SC_2_C_VERSION
	_POSIX2_CHAR_TERM	_SC_2_CHAR_TERM
	_POSIX2_FORT_DEV	_SC_2_FORT_DEV
	_POSIX2_FORT_RUN	_SC_2_FORT_RUN
	_POSIX2_LOCALEDEF	_SC_2_LOCALEDEF
BE	_POSIX2_PBS	_SC_2_PBS
	_POSIX2_PBS_ACCOUNTING	_SC_2_PBS_ACCOUNTING
	_POSIX2_PBS_LOCATE	_SC_2_PBS_LOCATE
	_POSIX2_PBS_MESSAGE	_SC_2_PBS_MESSAGE
	_POSIX2_PBS_TRACK	_SC_2_PBS_TRACK
	_POSIX2_SW_DEV	_SC_2_SW_DEV
	_POSIX2_UPE	_SC_2_UPE
	_POSIX2_VERSION	_SC_2_VERSION
	_REGEX_VERSION	_SC_REGEX_VERSION
XSI	{PAGE_SIZE}	_SC_PAGE_SIZE
	{PAGESIZE}	_SC_PAGESIZE

	Variable	Value of Name
THR	{PTHREAD_DESTRUCTOR_ITERATIONS}	_SC_THREAD_DESTRUCTOR_ITERATIONS
	{PTHREAD_KEYS_MAX}	_SC_THREAD_KEYS_MAX
	{PTHREAD_STACK_MIN}	_SC_THREAD_STACK_MIN
	{PTHREAD_THREADS_MAX}	_SC_THREAD_THREADS_MAX
	{RE_DUP_MAX}	_SC_RE_DUP_MAX
RTS	{RTSIG_MAX}	_SC_RTSIG_MAX
SEM	{SEM_NSEMS_MAX}	_SC_SEM_NSEMS_MAX
	{SEM_VALUE_MAX}	_SC_SEM_VALUE_MAX
RTS	{SIGQUEUE_MAX}	_SC_SIGQUEUE_MAX
	{STREAM_MAX}	_SC_STREAM_MAX
	{SYMLOOP_MAX}	_SC_SYMLOOP_MAX
TMR	{TIMER_MAX}	_SC_TIMER_MAX
	{TTY_NAME_MAX}	_SC_TTY_NAME_MAX
	{TZNAME_MAX}	_SC_TZNAME_MAX
XSI	_XBS5_ILP32_OFF32 (**LEGACY**)	_SC_XBS5_ILP32_OFF32 (**LEGACY**)
	_XBS5_ILP32_OFFBIG (**LEGACY**)	_SC_XBS5_ILP32_OFFBIG (**LEGACY**)
	_XBS5_LP64_OFF64 (**LEGACY**)	_SC_XBS5_LP64_OFF64 (**LEGACY**)
	_XBS5_LPBIG_OFFBIG (**LEGACY**)	_SC_XBS5_LPBIG_OFFBIG (**LEGACY**)
	_XOPEN_CRYPT	_SC_XOPEN_CRYPT
	_XOPEN_ENH_I18N	_SC_XOPEN_ENH_I18N
	_XOPEN_LEGACY	_SC_XOPEN_LEGACY
	_XOPEN_REALTIME	_SC_XOPEN_REALTIME
	_XOPEN_REALTIME_THREADS	_SC_XOPEN_REALTIME_THREADS
	_XOPEN_SHM	_SC_XOPEN_SHM
	_XOPEN_UNIX	_SC_XOPEN_UNIX
	_XOPEN_VERSION	_SC_XOPEN_VERSION
	_XOPEN_XCU_VERSION	_SC_XOPEN_XCU_VERSION

RETURN VALUE

If *name* is an invalid value, *sysconf*( ) shall return –1 and set *errno* to indicate the error. If the variable corresponding to *name* has no limit, *sysconf*( ) shall return –1 without changing the value of *errno*. Note that indefinite limits do not imply infinite limits; see **<limits.h>**.

Otherwise, *sysconf*( ) shall return the current variable value on the system. The value returned shall not be more restrictive than the corresponding value described to the application when it was compiled with the implementation's **<limits.h>** or **<unistd.h>**. The shall does not change during the lifetime of the calling process.

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.

If the value of *sysconf*(_SC_2_VERSION) is not equal to the value of the _POSIX2_VERSION symbolic constant, the utilities available via *system*( ) or *popen*( ) might not behave as described in the Shell and Utilities volume of IEEE Std. 1003.1-200x. This would mean that the application is

not running in an environment that conforms to the Shell and Utilities volume of IEEE Std. 1003.1-200x. Some applications might be able to deal with this, others might not. However, the functions defined in this volume of IEEE Std. 1003.1-200x continue to operate as specified, even if: *sysconf*(_SC_2_VERSION) reports that the utilities no longer perform as specified.

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 portable 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 50Hz and 60Hz 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.

This runtime facility is not meant to provide ever-changing values that applications have to check multiple times. The values are seen as changing no more frequently than once per system initialization, such as by a system administrator or operator with an automatic configuration program. This volume of IEEE Std. 1003.1-200x specifies that they shall not change within the lifetime of the process.

Some values apply to the system overall and others vary at the file system or directory level. The latter are described in *pathconf*( ).

Note that all values returned must be expressible as integers. String values were considered, but the additional flexibility of this approach was rejected due to its added complexity of

implementation and use.

Some values, such as {PATH_MAX}, are sometimes so large that they must not be used to, say, allocate arrays. The *sysconf*( ) function returns a negative value to show that this symbolic constant is not even defined in this case.

Similar to *pathconf*( ), this permits the implementation not to have a limit. When one resource is infinite, returning an error indicating that some other resource limit has been reached is conforming behavior.

FUTURE DIRECTIONS

None.

SEE ALSO

confstr( ), *pathconf*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<limits.h>**, **<unistd.h>**, the Shell and Utilities volume of IEEE Std. 1003.1-200x, *getconf*

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The type of the function return value is expanded to **long**.

_XOPEN_VERSION is added to the table of configurable system limits; this should have been included in Issue 3.

The following variables are added to the table of configurable system limits in the DESCRIPTION and marked as extensions:

_XOPEN_CRYPT
_XOPEN_ENH_I18N
_XOPEN_SHM
_XOPEN_UNIX

In the RETURN VALUE section the header **<time.h>** is given as an alternative to **<limits.h>** and **<unistd.h>**.

The second paragraph is added to the APPLICATION USAGE section.

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The variables {STREAM_MAX} and {TZNAME_MAX} are added to the table of variables in the DESCRIPTION.

The following change is incorporated for alignment with the ISO POSIX-2 standard:

- The following variables are added to the table of configurable system limits in the DESCRIPTION:

{BC_BASE_MAX}	_POSIX2_C_BIND	_POSIX2_SW_DEV
{BC_DIM_MAX}	_POSIX2_C_DEV	_POSIX2_VERSION
{BC_SCALE_MAX}	_POSIX2_C_VERSION	{RE_DUP_MAX}
{BC_STRING_MAX}	_POSIX2_CHAR_TERM	
{COLL_WEIGHTS_MAX}	_POSIX2_FORT_DEV	
{EXPR_NEST_MAX}	_POSIX2_FORT_RUN	
{LINE_MAX}	_POSIX2_LOCALEDEF	

Issue 4, Version 2

For X/OPEN UNIX conformance, the {ATEXIT_MAX}, {IOV_MAX}, {PAGESIZE}, {PAGE_SIZE}, and _XOPEN_UNIX variables are added to the list of configurable system values that can be determined by calling *sysconf*( ).

Issue 5

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

The _XBS_ variables and name values are added to the table of system variables in the DESCRIPTION. These are all marked EX.

Issue 6

The symbol CLK_TCK is obsolescent and removed. It is replaced with the phrase ‘‘clock ticks per second’’.

The symbol {PASS_MAX} is removed.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Table entries added for the following variables: _SC_REGEXP, _SC_SHELL, _SC_REGEX_VERSION, _SC_SYMLOOP_MAX.

The following *sysconf*( ) variables and their associated names are added for alignment with IEEE Std. 1003.1d-1999:

_POSIX_ADVISORY_INFO
_POSIX_CPUTIME
_POSIX_SPAWN
_POSIX_SPORADIC_SERVER
_POSIX_THREAD_CPUTIME
_POSIX_THREAD_SPORADIC_SERVER
_POSIX_TIMEOUTS

The following changes are made to the DESCRIPTION for alignment with IEEE Std. 1003.1j-2000:

- A statement expressing the dependency of support for some system variables on implementation options is added.
- The following system variables are added:

  _POSIX_BARRIERS
  _POSIX_CLOCK_SELECTION
  _POSIX_MONOTONIC_CLOCK
  _POSIX_READER_WRITER_LOCKS
  _POSIX_SPIN_LOCKS
  _POSIX_TYPED_MEMORY_OBJECTS

The following system variables are added for alignment with IEEE Std. 1003.2d-1994:

_POSIX2_PBS
_POSIX2_PBS_ACCOUNTING
_POSIX2_PBS_LOCATE
_POSIX2_PBS_MESSAGE
_POSIX2_PBS_TRACK

The following *sysconf*( ) variables and their associated names are added for alignment with IEEE Std. 1003.1q-2000:

45817 _POSIX_TRACE
45818 _POSIX_TRACE_EVENT_FILTER
45819 _POSIX_TRACE_INHERIT
45820 _POXIC_TRACE_LOG

45821 The macros associated with the *c89* programming models are marked LEGACY, and new
45822 equivalent macros associated with *c99* are introduced.

45823 **NAME**

45824 syslog — log a message

45825 **SYNOPSIS**

45826 XSI
```
#include <syslog.h>

void syslog(int priority, const char *message, ... /* argument */);
```
45827

45828

45829 **DESCRIPTION**

45830 Refer to *closelog*( ).

45831 **NAME**

45832 system — issue a command

45833 **SYNOPSIS**

45834 `#include <stdlib.h>`

45835 `int system(const char *`*`command`*`);`

45836 **DESCRIPTION**

45837 CX The functionality described on this reference page is aligned with the ISO C standard. Any
45838 conflict between the requirements described here and the ISO C standard is unintentional. This
45839 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

45840 The *system*( ) function passes the string pointed to by *command* to the host environment to be
45841 executed by a command processor in an implementation-defined manner. The environment of
45842 the executed command shall be as if a child process were created using the *fork*( ) function, and
45843 the child process invoked a command interpreter using the *execl*( ) function.

45844 CX If the implementation supports the Shell and Utilities volume of IEEE Std. 1003.1-200x
45845 commands, the environment of the executed command shall be as if a child process were created
45846 using *fork*( ), and the child process invoked the *sh* utility using *execl*( ) as follows:

45847 `execl(<`*`shell path`*`>, "sh", "-c", `*`command`*`, (char *)0);`

45848 where <*shell path*> is an unspecified path name for the *sh* utility.

45849 The *system*( ) function ignores the SIGINT and SIGQUIT signals, and blocks the SIGCHLD
45850 signal, while waiting for the command to terminate. If this might cause the application to miss a
45851 signal that would have killed it, then the application should examine the return value from
45852 *system*( ) and take whatever action is appropriate to the application if the command terminated
45853 due to receipt of a signal.

45854 The *system*( ) function shall not affect the termination status of any child of the calling processes
45855 other than the process or processes it itself creates.

45856 The *system*( ) function shall not return until the child process has terminated.

45857 **RETURN VALUE**

45858 If *command* is a null pointer, *system*( ) shall return non-zero to indicate that a command processor
45859 CX is available, or zero if none is available. If the implementation supports the utilities defined in
45860 the Shell and Utilities volume of IEEE Std. 1003.1-200x, *system*( ) shall always return non-zero
45861 when *command* is NULL.

45862 CX If *command* is not a null pointer, *system*( ) shall return the termination status of the command
45863 language interpreter in the format specified by *waitpid*( ). If the implementation supports the
45864 utilities defined in the Shell and Utilities volume of IEEE Std. 1003.1-200x, the termination status
45865 shall be as defined for the *sh* utility; otherwise, the termination status is unspecified. If some
45866 error prevents the command language interpreter from executing after the child process is
45867 created, the return value from *system*( ) shall be as if the command language interpreter had
45868 terminated using *exit*(127) or *_exit*(127). If a child process cannot be created, or if the
45869 termination status for the command language interpreter cannot be obtained, *system*( ) shall
45870 return –1 and set *errno* to indicate the error.

45871 **ERRORS**

45872 CX The *system*( ) function may set *errno* values as described by *fork*( ).

45873 In addition, *system*( ) may fail if:

45874 CX [ECHILD] The status of the child process created by *system*( ) is no longer available.

EXAMPLES

None.

APPLICATION USAGE

If the return value of *system*( ) is not –1, its value can be decoded through the use of the macros described in **<sys/wait.h>**. For convenience, these macros are also provided in **<stdlib.h>**.

To determine whether or not the environment specified in the Shell and Utilities volume of IEEE Std. 1003.1-200x is present, use *sysconf*(_SC_2_VERSION).

Note that, while *system*( ) must ignore SIGINT and SIGQUIT and block SIGCHLD while waiting for the child to terminate, the handling of signals in the executed command is as specified by *fork*( ) and *exec*. For example, if SIGINT is being caught or is set to SIG_DFL when *system*( ) is called, then the child is started with SIGINT handling set to SIG_DFL.

Ignoring SIGINT and SIGQUIT in the parent process prevents coordination problems (two processes reading from the same terminal, for example) when the executed command ignores or catches one of the signals. It is also usually the correct action when the user has given a command to the application to be executed synchronously (as in the '!' command in many interactive applications). In either case, the signal should be delivered only to the child process, not to the application itself. There is one situation where ignoring the signals might have less than the desired effect. This is when the application uses *system*( ) to perform some task invisible to the user. If the user typed the interrupt character ("^C", for example) while *system*( ) is being used in this way, one would expect the application to be killed, but only the executed command is killed. Applications that use *system*( ) in this way should carefully check the return status from *system*( ) to see if the executed command was successful, and should take appropriate action when the command fails.

Blocking SIGCHLD while waiting for the child to terminate prevents the application from catching the signal and obtaining status from *system*( )'s child process before *system*( ) can get the status itself.

The context in which the utility is ultimately executed may differ from that in which *system*( ) was called. For example, file descriptors that have the FD_CLOEXEC flag set are closed, and the process ID and parent process ID are different. Also, if the executed utility changes its environment variables or its current working directory, that change is not reflected in the caller's context.

There is no defined way for an application to find the specific path for the shell. However, *confstr*( ) can provide a value for *PATH* that is guaranteed to find the *sh* utility.

RATIONALE

The *system*( ) 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.

There are three levels of specification for the *system*( ) function. The ISO C standard gives the most basic. It requires that the function exists, and defines a way for an application to query whether a command language interpreter exists. It says nothing about the command language or the environment in which the command is interpreted.

IEEE Std. 1003.1-200x places additional restrictions on *system*( ). It requires that if there is a command language interpreter, the environment must be as specified by *fork*( ) and *exec*. This ensures, for example, that close-on-*exec* works, that file locks are not inherited, and that the process ID is different. It also specifies the return value from *system*( ) when the command line can be run, thus giving the application some information about the command's completion

status IEEE Std. 1003.1-200x in its base definition still says nothing about the interpretation of the command.

Finally, IEEE Std. 1003.1-200x requires the command to be interpreted as in the shell command language defined in the Shell and Utilities volume of IEEE Std. 1003.1-200x.

Note that, *system*(NULL) is required to return non-zero, indicating that there is a command language interpreter. At first glance, this would seem to conflict with the ISO C standard which allows *system*(NULL) to return zero. There is no conflict, however. A system must have a command language interpreter, and is non-conforming if none is present. It is therefore permissible for the *system*( ) function on such a system to implement the behavior specified by the ISO C standard as long as it is understood that the implementation does not conform to IEEE Std. 1003.1-200x if *system*(NULL) returns zero.

It was explicitly decided that when *command* is NULL, *system*( ) should not be required to check to make sure that the command language interpreter actually exists with the correct mode, that there are enough processes to execute it, and so on. The call *system*(NULL) could, theoretically, check for such problems as too many existing child processes, and return zero. However, it would be inappropriate to return zero due to such a (presumably) transient condition. If some condition exists that is not under the control of this application and that would cause any *system*( ) call to fail, that system has been rendered non-conforming.

Early drafts required, or allowed, *system*( ) to return with *errno* set to [EINTR] if it was interrupted with a signal. This error return was removed, and a requirement that *system*( ) not return until the child has terminated was added. This means that if a *waitpid*( ) call in *system*( ) exits with *errno* set to [EINTR], *system*( ) must re-issue the *waitpid*( ). This change was made for two reasons:

1. There is no way for an application to clean up if *system*( ) returns [EINTR], short of calling *wait*( ), and that could have the undesirable effect of returning the status of children other than the one started by *system*( ).
2. While it might require a change in some historical implementations, those implementations already have to be changed because they use *wait*( ) instead of *waitpid*( ).

Note that if the application is catching SIGCHLD signals, it will receive such a signal before a successful *system*( ) call returns.

To conform to IEEE Std. 1003.1-200x, *system*( ) must use *waitpid*( ), or some similar function, instead of *wait*( ).

The following code sample illustrates how *system*( ) might be implemented on an implementation conforming to IEEE Std. 1003.1-200x.

```
#include <signal.h>
int system(const char *cmd)
{
    int stat;
    pid_t pid;
    struct sigaction sa, savintr, savequit;
    sigset_t saveblock;
    if (cmd == NULL)
        return(1);
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigemptyset(&savintr.sa_mask);
```

```
        sigemptyset(&savequit.sa_mask);
        sigaction(SIGINT, &sa, &savintr);
        sigaction(SIGQUIT, &sa, &savequit);
        sigaddset(&sa.sa_mask, SIGCHLD);
        sigprocmask(SIG_BLOCK, &sa.sa_mask, &saveblock);
        if ((pid = fork()) == 0) {
            sigaction(SIGINT, &savintr, (struct sigaction *)0);
            sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
            sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
            execl("/bin/sh", "sh", "-c", cmd, (char *)0);
            _exit(127);
        }
        if (pid == -1) {
            stat = -1; /* errno comes from fork() */
        } else {
            while (waitpid(pid, &stat, 0) == -1) {
                if (errno != EINTR){
                    stat = -1;
                    break;
                }
            }
        }
        sigaction(SIGINT, &savintr, (struct sigaction *)0);
        sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
        sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
        return(stat);
    }
```

Note that, while a particular implementation of *system*( ) (such as the one above) can assume a particular path for the shell, such a path is not necessarily valid on another system. The above example is not portable, and is not intended to be.

One reviewer suggested that an implementation of *system*( ) might want to use an environment variable such as *SHELL* to determine which command interpreter to use. The supposed implementation would use the default command interpreter if the one specified by the environment variable was not available. This would allow a user, when using an application that prompts for command lines to be processed using *system*( ), to specify a different command interpreter. Such an implementation is discouraged. If the alternate command interpreter did not follow the command line syntax specified in the Shell and Utilities volume of IEEE Std. 1003.1-200x, then changing *SHELL* would render *system*( ) non-conforming. This would affect applications that expected the specified behavior from *system*( ), and since the Shell and Utilities volume of IEEE Std. 1003.1-200x does not mention that *SHELL* affects *system*( ), the application would not know that it needed to unset *SHELL*.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *pipe*( ), *waitpid*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<limits.h>**, **<signal.h>**, **<stdlib.h>**, **<sys/wait.h>**, the Shell and Utilities volume of IEEE Std. 1003.1-200x

46015 **CHANGE HISTORY**

46016 First released in Issue 1. Derived from Issue 1 of the SVID.

46017 **Issue 4**

46018 Extensions beyond the ISO C standard are now marked.

46019 The following changes are incorporated for alignment with the ISO POSIX-2 standard:

46020 • The function is no longer marked as an extension.

46021 • The name of the argument is changed from *string* to *command*, and its type is changed from
46022 **char*** to **const char***.

46023 • The DESCRIPTION and RETURN VALUE sections are completely replaced to bring them in
46024 line with the ISO POSIX-2 standard. They still describe essentially the same functionality,
46025 albeit that the definition is more complete.

46026 • The ERRORS section is changed to indicate that *system*( ) may return error values described
46027 for *fork*( ).

46028 • The APPLICATION USAGE section is added.

46029 The following changes were made to align with the IEEE P1003.1a draft standard:

46030 • The DESCRIPTION is adjusted to reflect the behavior on systems that do not support the
46031 Shell option.

NAME

tan, tanf, tanl — tangent function

SYNOPSIS

```
#include <math.h>

double tan(double x);
float tanf(float x);
long double tanl(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 tangent of its argument *x*, measured in radians.

An application wishing to check for error situations should set *errno* to 0 before calling *tan*(). If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

The *tan*() function may lose accuracy when its argument is far from 0.0.

RETURN VALUE

Upon successful completion, these functions shall return the tangent of *x*.

XSI If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

If *x* is ±Inf, either 0.0 shall be returned and *errno* set to [EDOM], or NaN shall be returned and *errno* may be set to [EDOM].

If the correct value would cause overflow, ±HUGE_VAL shall be returned and *errno* shall be set to [ERANGE].

If the correct value would cause underflow, 0.0 shall be returned and *errno* may be set to [ERANGE].

ERRORS

These functions shall fail if:

[ERANGE] The value to be returned would cause overflow.

These functions may fail if:

XSI [EDOM] The value *x* is NaN or ±Inf.

[ERANGE] The value to be returned would cause underflow.

XSI No other errors shall occur.

EXAMPLES

Taking the Tangent of a 45-Degree Angle

```
#include <math.h>
...
double radians = 45.0 * M_PI / 180;
double result;
...
result = tan (radians);
```

46071 **APPLICATION USAGE**
46072 None.

46073 **RATIONALE**
46074 None.

46075 **FUTURE DIRECTIONS**
46076 None.

46077 **SEE ALSO**
46078 *atan*( ), *isnan*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

46079 **CHANGE HISTORY**
46080 First released in Issue 1. Derived from Issue 1 of the SVID.

46081 **Issue 4**
46082 References to *matherr*( ) are removed.

46083 The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the
46084 ISO C standard and to rationalize error handling in the mathematics functions.

46085 The return value specified for [EDOM] is marked as an extension.

46086 **Issue 5**
46087 The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes
46088 in previous issues.

46089 **Issue 6**
46090 The *tanf*( ) and *tanl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

46091 **NAME**

46092 tanh, tanhf, tanhl — hyperbolic tangent function

46093 **SYNOPSIS**

46094
```
#include <math.h>
```

46095
```
double tanh(double x);
```
46096
```
float tanhf(float x);
```
46097
```
long double tanhl(long double x);
```

46098 **DESCRIPTION**

46099 CX The functionality described on this reference page is aligned with the ISO C standard. Any
46100 conflict between the requirements described here and the ISO C standard is unintentional. This
46101 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

46102 These functions shall ompute the hyperbolic tangent of *x*.

46103 An application wishing to check for error situations should set *errno* to 0 before calling *tanh*( ). If
46104 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

46105 **RETURN VALUE**

46106 Upon successful completion, these functions shall return the hyperbolic tangent of *x*.

46107 XSI If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

46108 If the correct value would cause underflow, 0.0 shall be returned and *errno* may be set to
46109 [ERANGE].

46110 **ERRORS**

46111 These functions may fail if:

46112 XSI [EDOM] The value of *x* is NaN.

46113 [ERANGE] The correct result would cause underflow.

46114 XSI No other errors shall occur.

46115 **EXAMPLES**

46116 None.

46117 **APPLICATION USAGE**

46118 None.

46119 **RATIONALE**

46120 None.

46121 **FUTURE DIRECTIONS**

46122 None.

46123 **SEE ALSO**

46124 *atanh*( ), *isnan*( ), *tan*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

46125 **CHANGE HISTORY**

46126 First released in Issue 1. Derived from Issue 1 of the SVID.

46127 **Issue 4**

46128 References to *matherr*( ) are removed.

46129 The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the
46130 ISO C standard and to rationalize error handling in the mathematics functions.

46131 The return value specified for [EDOM] is marked as an extension.

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 *tanhf*( ) and *tanhl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

46137 **NAME**

46138 tcdrain — wait for transmission of output

46139 **SYNOPSIS**

46140 `#include <termios.h>`

46141 `int tcdrain(int fildes);`

46142 **DESCRIPTION**

46143 The *tcdrain*( ) function shall wait until all output written to the object referred to by *fildes* is
46144 transmitted. The *fildes* argument is an open file descriptor associated with a terminal.

46145 Any attempts to use *tcdrain*( ) from a process which is a member of a background process group
46146 on a *fildes* associated with its controlling terminal, shall cause the process group to be sent a
46147 SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is
46148 allowed to perform the operation, and no signal is sent.

46149 **RETURN VALUE**

46150 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
46151 indicate the error.

46152 **ERRORS**

46153 The *tcdrain*( ) function shall fail if:

46154 [EBADF] The *fildes* argument is not a valid file descriptor. |

46155 [EINTR] A signal interrupted *tcdrain*( ). |

46156 [ENOTTY] The file associated with *fildes* is not a terminal. |

46157 The *tcdrain*( ) function may fail if:

46158 [EIO] The process group of the writing process is orphaned, and the writing process |
46159 is not ignoring or blocking SIGTTOU. |

46160 **EXAMPLES**

46161 None.

46162 **APPLICATION USAGE**

46163 None.

46164 **RATIONALE**

46165 None.

46166 **FUTURE DIRECTIONS**

46167 None.

46168 **SEE ALSO**

46169 *tcflush*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<termios.h>**, **<unistd.h>**, the |
46170 Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 11, General Terminal Interface |

46171 **CHANGE HISTORY**

46172 First released in Issue 3.

46173 Entry included for alignment with the POSIX.1-1988 standard.

46174 **Issue 4**

46175 The [EIO] error is added to the ERRORS section.

46176 The FUTURE DIRECTIONS section is added.

46177 The following change is incorporated for alignment with the FIPS requirements:

- The words ‘‘If _POSIX_JOB_CONTROL is defined’’ are removed from the start of the second paragraph in the DESCRIPTION. This is because job control is defined as mandatory for Issue 4 conforming implementations.

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.

46187 **NAME**

46188 tcflow — suspend or restart the transmission or reception of data

46189 **SYNOPSIS**

46190 `#include <termios.h>`

46191 `int tcflow(int fildes, int action);`

46192 **DESCRIPTION**

46193 The *tcflow*( ) function shall suspend or restart transmission or reception of data on the object
46194 referred to by *fildes*, depending on the value of *action*. The *fildes* argument is an open file
46195 descriptor associated with a terminal.

46196 • If *action* is TCOOFF, output shall be suspended.

46197 • If *action* is TCOON, suspended output shall be restarted.

46198 • If *action* is TCIOFF, the system shall transmit a STOP character, which is intended to cause
46199 the terminal device to stop transmitting data to the system.

46200 • If *action* is TCION, the system shall transmit a START character, which is intended to cause
46201 the terminal device to start transmitting data to the system.

46202 The default on the opening of a terminal file is that neither its input nor its output are
46203 suspended.

46204 Attempts to use *tcflow*( ) from a process which is a member of a background process group on a
46205 *fildes* associated with its controlling terminal, shall cause the process group to be sent a
46206 SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is
46207 allowed to perform the operation, and no signal is sent.

46208 **RETURN VALUE**

46209 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
46210 indicate the error.

46211 **ERRORS**

46212 The *tcflow*( ) function shall fail if:

46213 [EBADF] The *fildes* argument is not a valid file descriptor. |

46214 [EINVAL] The *action* argument is not a supported value. |

46215 [ENOTTY] The file associated with *fildes* is not a terminal. |

46216 The *tcflow*( ) function may fail if:

46217 [EIO] The process group of the writing process is orphaned, and the writing process |
46218 is not ignoring or blocking SIGTTOU. |

46219 **EXAMPLES**

46220 None.

46221 **APPLICATION USAGE**

46222 None.

46223 **RATIONALE**

46224 None.

46225 **FUTURE DIRECTIONS**

46226 None.

SEE ALSO

tcsendbreak( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<termios.h>**, **<unistd.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The descriptions of TCIOFF and TCION are reworded, indicating the intended consequences of transmitting stop and start characters. Issue 3 implied that these consequences were guaranteed.

The [EIO] error is added to the ERRORS section.

The FUTURE DIRECTIONS section is added.

The following change is incorporated for alignment with the FIPS requirements:

- The words ‘‘If _POSIX_JOB_CONTROL is defined’’ are removed from the start of the second paragraph in the DESCRIPTION. This is because job control is defined as mandatory for Issue 4 conforming implementations.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] error is added.

NAME

tcflush — flush non-transmitted output data, non-read input data, or both

SYNOPSIS

```
#include <termios.h>

int tcflush(int fildes, int queue_selector);
```

DESCRIPTION

Upon successful completion, *tcflush*( ) shall discard data written to the object referred to by *fildes* (an open file descriptor associated with a terminal) but not transmitted, or data received but not read, depending on the value of *queue_selector*:

- 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 written but not transmitted.

Attempts to use *tcflush*( ) from a process which is a member of a background process group on a *fildes* associated with its controlling terminal, shall cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is allowed to perform the operation, and no signal is sent.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcflush*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINVAL] The *queue_selector* argument is not a supported value.

[ENOTTY] The file associated with *fildes* is not a terminal.

The *tcflush*( ) function may fail if:

[EIO] The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

tcdrain( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<termios.h>**, **<unistd.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The DESCRIPTION is modified to indicate that the flush operation only results if the call to *tcflush*( ) is successful.

The [EIO] error is added to the ERRORS section.

The FUTURE DIRECTIONS section is added.

The following change is incorporated for alignment with the FIPS requirements:

- The words ‘‘If _POSIX_JOB_CONTROL is defined’’ are removed from the start of the second paragraph in the DESCRIPTION. This is because job control is defined as mandatory for Issue 4 conforming implementations.

Issue 6

The Open Group corrigenda item U035/1 has been applied. In the ERRORS and APPLICATION USAGE sections, references to *tcflow*( ) are replaced with *tcflush*( ).

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.

46305 **NAME**

46306 tcgetattr — get the parameters associated with the terminal

46307 **SYNOPSIS**

46308 `#include <termios.h>`

46309 `int tcgetattr(int` *`fildes`*`, struct termios *`*`termios_p`*`);`

46310 **DESCRIPTION**

46311 The *tcgetattr*( ) function shall get the parameters associated with the terminal referred to by *fildes*
46312 and store them in the **termios** structure referenced by *termios_p*. The *fildes* argument is an open
46313 file descriptor associated with a terminal.

46314 The *termios_p* argument is a pointer to a **termios** structure.

46315 The *tcgetattr*( ) operation is allowed from any process.

46316 If the terminal device supports different input and output baud rates, the baud rates stored in
46317 the **termios** structure returned by *tcgetattr*( ) shall reflect the actual baud rates, even if they are
46318 equal. If differing baud rates are not supported, the rate returned as the output baud rate shall be
46319 the actual baud rate. If the terminal device does not support split baud rates, the input baud rate
46320 stored in the **termios** structure shall be the output rate (as one of the symbolic values).

46321 **RETURN VALUE**

46322 Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* set to
46323 indicate the error.

46324 **ERRORS**

46325 The *tcgetattr*( ) function shall fail if:

46326 [EBADF] The *fildes* argument is not a valid file descriptor. |

46327 [ENOTTY] The file associated with *fildes* is not a terminal. |

46328 **EXAMPLES**

46329 None.

46330 **APPLICATION USAGE**

46331 None.

46332 **RATIONALE**

46333 Care must be taken when changing the terminal attributes. Applications should always do a
46334 *tcgetattr*( ), save the **termios** structure values returned, and then do a *tcsetattr*( ) changing only
46335 the necessary fields. The application should use the values saved from the *tcgetattr*( ) to reset the
46336 terminal state whenever it is done with the terminal. This is necessary because terminal
46337 attributes apply to the underlying port and not to each individual open instance; that is, all
46338 processes that have used the terminal see the latest attribute changes.

46339 A program that uses these functions should be written to catch all signals and take other
46340 appropriate actions to ensure that when the program terminates, whether planned or not, the
46341 terminal device's state is restored to its original state.

46342 Existing practice dealing with error returns when only part of a request can be honored is based
46343 on calls to the *ioctl*( ) function. In historical BSD and System V implementations, the
46344 corresponding *ioctl*( ) returns zero if the requested actions were semantically correct, even if
46345 some of the requested changes could not be made. Many existing applications assume this
46346 behavior and would no longer work correctly if the return value were changed from zero to −1
46347 in this case.

Note that either specification has a problem. When zero is returned, it implies everything succeeded even if some of the changes were not made. When –1 is returned, it implies everything failed even though some of the changes were made.

Applications that need all of the requested changes made to work properly should follow *tcsetattr*( ) with a call to *tcgetattr*( ) and compare the appropriate field values.

FUTURE DIRECTIONS

None.

SEE ALSO

tcsetattr( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<termios.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The FUTURE DIRECTIONS section is added to allow for alignment with the ISO POSIX-1 standard.

Issue 6

In the DESCRIPTION, the rate returned as the input baud rate shall be the output rate. Previously, the number zero was also allowed but was obsolescent.

NAME

tcgetpgrp — get the foreground process group ID

SYNOPSIS

```
#include <unistd.h>

pid_t tcgetpgrp(int fildes);
```

DESCRIPTION

The *tcgetpgrp*( ) function shall return the value of the process group ID of the foreground process group associated with the terminal.

If there is no foreground process group, *tcgetpgrp*( ) returns a value greater than 1 that does not match the process group ID of any existing process group.

The *tcgetpgrp*( ) function is allowed from a process that is a member of a background process group; however, the information may be subsequently changed by a process that is a member of a foreground process group.

RETURN VALUE

Upon successful completion, *tcgetpgrp*( ) shall return the value of the process group ID of the foreground process associated with the terminal. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcgetpgrp*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid file descriptor. |

[ENOTTY] The calling process does not have a controlling terminal, or the file is not the controlling terminal. |

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

setsid( ), *setpgid*( ), *tcsetpgrp*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<unistd.h>** |

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

The **<unistd.h>** header is added to the SYNOPSIS section.

The following change is incorporated for alignment with the FIPS requirements:

- The DESCRIPTION is clarified and the phrase ‘‘If _POSIX_JOB_CONTROL is defined’’ is removed because job control is now mandatory on all XSI-conformant systems.

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, text previously conditional on support for _POSIX_JOB_CONTROL is now mandatory. This is a FIPS requirement.

NAME

tcgetsid — get process group ID for session leader for controlling terminal

SYNOPSIS

XSI
```
#include <termios.h>

pid_t tcgetsid(int fildes);
```

DESCRIPTION

The *tcgetsid*( ) function shall obtain the process group ID of the session for which the terminal specified by *fildes* is the controlling terminal.

RETURN VALUE

Upon successful completion, *tcgetsid*( ) shall return the process group ID associated with the terminal. Otherwise, a value of (**pid_t**)–1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcgetsid*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid file descriptor.

[ENOTTY] The calling process does not have a controlling terminal, or the file is not the controlling terminal.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std. 1003.1-200x, **<termios.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The [EACCES] error has been removed from the list of mandatory errors, and the description of [ENOTTY] has been reworded.

NAME

tcsendbreak — send a ‘‘break’’ for a specific duration

SYNOPSIS

```
#include <termios.h>

int tcsendbreak(int fildes, int duration);
```

DESCRIPTION

The *fildes* argument is an open file descriptor associated with a terminal.

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.

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 *fildes* associated with its controlling terminal, shall cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is allowed to perform the operation, and no signal is sent.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcsendbreak*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid file descriptor.

[ENOTTY] The file associated with *fildes* is not a terminal.

The *tcsendbreak*( ) function may fail if:

[EIO] The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std. 1003.1-200x, **<termios.h>**, **<unistd.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 11, General Terminal Interface

46492 **CHANGE HISTORY**

46493 First released in Issue 3.

46494 Entry included for alignment with the POSIX.1-1988 standard.

46495 **Issue 4**

46496 The [EIO] error is added to the ERRORS section.

46497 The following change is incorporated for alignment with the FIPS requirements:

46498 • In the DESCRIPTION the phrase ‘‘If _POSIX_JOB_CONTROL is defined’’ is removed
46499 because job control is now mandatory on all XSI-conformant systems.

46500 **Issue 6**

46501 The following new requirements on POSIX implementations derive from alignment with the
46502 Single UNIX Specification:

46503 • In the DESCRIPTION, text previously conditional on _POSIX_JOB_CONTROL is now
46504 mandated. This is a FIPS requirement.

46505 • The [EIO] error is added.

NAME

tcsetattr — set the parameters associated with the terminal

Notes to Reviewers

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

See the FUTURE DIRECTIONS section.

SYNOPSIS

```
#include <termios.h>

int tcsetattr(int fildes, int optional_actions,
    const struct termios *termios_p);
```

DESCRIPTION

The *tcsetattr*( ) function shall set the parameters associated with the terminal referred to by the open file descriptor *fildes* (an open file descriptor associated with a terminal) from the **termios** structure referenced by *termios_p* as follows:

- If *optional_actions* is TCSANOW, the change shall occur immediately.
- If *optional_actions* is TCSADRAIN, the change shall occur after all output written to *fildes* is 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 *fildes* is transmitted, and all input so far received but not read shall be discarded before the change is made.

If the output baud rate stored in the **termios** structure pointed to by *termios_p* is the zero baud rate, B0, the modem control lines shall no longer be asserted. Normally, this shall disconnect the line.

If the input baud rate stored in the **termios** structure pointed to by *termios_p* is 0, the input baud rate given to the hardware is the same as the output baud rate stored in the **termios** structure.

The *tcsetattr*( ) function shall return successfully if it was able to perform any of the requested actions, even if some of the requested actions could not be performed. It shall set all the attributes that the implementation supports as requested and leaves all the attributes not supported by the implementation unchanged. If no part of the request can be honored, it shall return −1 and set *errno* to [EINVAL]. If the input and output baud rates differ and are a combination that is not supported, neither baud rate is changed. A subsequent call to *tcgetattr*( ) shall return the actual state of the terminal device (reflecting both the changes made and not made in the previous *tcsetattr*( ) call). The *tcsetattr*( ) function shall not change the values found in the **termios** structure under any circumstances.

The effect of *tcsetattr*( ) is undefined if the value of the **termios** structure pointed to by *termios_p* was not derived from the result of a call to *tcgetattr*( ) on *fildes*; an application should modify only fields and flags defined by this volume of IEEE Std. 1003.1-200x between the call to *tcgetattr*( ) and *tcsetattr*( ), leaving all other fields and flags unmodified.

No actions defined by this volume of IEEE Std. 1003.1-200x, other than a call to *tcsetattr*( ) or a close of the last file descriptor in the system associated with this terminal device, shall cause any of the terminal attributes defined by this volume of IEEE Std. 1003.1-200x to change.

If *tcsetattr*( ) is called from a process which is a member of a background process group on a *fildes* associated with its controlling terminal:

- If the calling process is blocking or ignoring SIGTTOU signals, the operation completes normally and no signal is sent.

- Otherwise, a SIGTTOU signal shall be sent to the process group.

RETURN VALUE

Upon successful completion, 0 shall be shall be. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcsetattr*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINTR] A signal interrupted *tcsetattr*( ).

[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.

[ENOTTY] The file associated with *fildes* is not a terminal.

The *tcsetattr*( ) function may fail if:

[EIO] The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

EXAMPLES

None.

APPLICATION USAGE

If trying to change baud rates, applications should call *tcsetattr*( ) then call *tcgetattr*( ) in order to determine what baud rates were actually selected.

RATIONALE

The *tcsetattr*( ) function can be interrupted in the following situations:

- It is interrupted while waiting for output to drain.
- It is called from a process in a background process group and SIGTTOU is caught.

See also the RATIONALE section in *tcgetattr*( ).

FUTURE DIRECTIONS

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.

SEE ALSO

cfgetispeed( ), *tcgetattr*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<termios.h>**, **<unistd.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The words ‘‘and stores them in’’ are changed to ‘‘from’’ in the first paragraph of the DESCRIPTION.

The [EINTR] and [EIO] errors are added to the ERRORS section.

The FUTURE DIRECTIONS section is added to allow for alignment with the ISO POSIX-1 standard.

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The argument *termios_p* is changed from type **struct termios*** to **const struct termios***.

The following change is incorporated for alignment with the FIPS requirements:

- In the DESCRIPTION the phrase ‘‘If _POSIX_JOB_CONTROL is defined’’ is removed because job control is now mandatory on all XSI-conformant systems.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, text previously conditional on _POSIX_JOB_CONTROL is now mandated. This is a FIPS requirement.
- The [EIO] error is added.

In the DESCRIPTION, the text describing use of *tcsetattr*( ) from a process which is a member of a background process group is clarified.

NAME

tcsetpgrp — set the foreground process group ID

SYNOPSIS

```
#include <unistd.h>

int tcsetpgrp(int fildes, pid_t pgid_id);
```

DESCRIPTION

If the process has a controlling terminal, *tcsetpgrp*( ) shall set the foreground process group ID associated with the terminal to *pgid_id*. The application shall ensure that the file associated with *fildes* is the controlling terminal of the calling process and the controlling terminal is currently associated with the session of the calling process. The application shall ensure that the value of *pgid_id* matches a process group ID of a process in the same session as the calling process.

Attempts to use *tcsetpgrp*( ) from a process which is a member of a background process group on a *fildes* associated with its controlling terminal will cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is allowed to perform the operation, and no signal is sent.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcsetpgrp*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINVAL] This implementation does not support the value in the *pgid_id* argument.

[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.

[EPERM] The value of *pgid_id* 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.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

tcgetpgrp( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

The **<unistd.h>** header is added to the SYNOPSIS section.

The [ENOSYS] error is removed from the ERRORS section.

The following change is incorporated for alignment with the FIPS requirements:

- In the DESCRIPTION the phrase ‘‘If _POSIX_JOB_CONTROL is defined’’ is removed because job control is now mandatory on all XSI-conformant systems.

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 DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements. |

The Open Group corrigenda item U047/4 has been applied. |

NAME

tdelete, tfind, tsearch, twalk — manage a binary search tree

SYNOPSIS

XSI

```
#include <search.h>

void *tdelete(const void *restrict key, void **restrict rootp,
    int(*compar)(const void *, const void *));
void *tfind(const void *key, void *const *rootp,
    int(*compar)(const void *, const void *));
void *tsearch(const void *key, void **rootp,
    int (*compar)(const void *, const void *));
void twalk(const void *root,
    void (*action)(const void *, VISIT, int));
```

DESCRIPTION

The *tdelete*(), *tfind*(), *tsearch*(), and *twalk*() functions manipulate binary search trees. Comparisons are made with a user-supplied routine, the address of which is passed as the *compar* argument. This routine is called with two arguments, the pointers to the elements being compared. The application shall ensure that the user-supplied routine returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to, or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The *tsearch*() function is used to build and access the tree. The *key* argument is a pointer to an element to be accessed or stored. If there is a node in the tree whose element is equal to the value pointed to by *key*, a pointer to this found node is returned. Otherwise, the value pointed to by *key* is inserted (that is, a new node is created and the value of *key* is copied to this node), and a pointer to this node returned. Only pointers are copied, so the application shall ensure that the calling routine stores the data. The *rootp* argument points to a variable that points to the root node of the tree. A null pointer value for the variable pointed to by *rootp* denotes an empty tree; in this case, the variable shall be set to point to the node which shall be at the root of the new tree.

Like *tsearch*(), *tfind*() shall search for a node in the tree, returning a pointer to it if found. However, if it is not found, *tfind*() shall return a null pointer. The arguments for *tfind*() are the same as for *tsearch*().

The *tdelete*() function deletes a node from a binary search tree. The arguments are the same as for *tsearch*(). The variable pointed to by *rootp* shall be changed if the deleted node was the root of the tree. The *tdelete*() function returns a pointer to the parent of the deleted node, or a null pointer if the node is not found.

The *twalk*() function traverses a binary search tree. The *root* argument is a pointer to the root node of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) The argument *action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The structure pointed to by this argument is unspecified and shall not be modified by the application, but it is guaranteed that a pointer-to-node can be converted to pointer-to-pointer-to-element to access the element stored in the node. The second argument is a value from an enumeration data type:

```
typedef enum { preorder, postorder, endorder, leaf } VISIT;
```

(defined in **<search.h>**), depending on whether this is the first, second, or third time that the node is visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level 0.

If the calling function alters the pointer to the root, the result is undefined.

RETURN VALUE

If the node is found, both *tsearch*( ) and *tfind*( ) shall return a pointer to it. If not, *tfind*( ) shall return a null pointer, and *tsearch*( ) shall return a pointer to the inserted item.

A null pointer shall be returned by *tsearch*( ) if there is not enough space available to create a new node.

A null pointer shall be returned by *tdelete*( ), *tfind*( ), and *tsearch*( ) if *rootp* is a null pointer on entry.

The *tdelete*( ) function shall return a pointer to the parent of the deleted node, or a null pointer if the node is not found.

The *twalk*( ) function shall return no value.

ERRORS

No errors are defined.

EXAMPLES

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

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

#define STRSZ    10000
#define NODSZ    500

struct node {      /* Pointers to these are stored in the tree. */
    char    *string;
    int     length;
};

char   string_space[STRSZ];  /* Space to store strings. */
struct node nodes[NODSZ];    /* Nodes to store. */
void  *root = NULL;          /* This points to the root. */

int main(int argc, char *argv[])
{
    char   *strptr = string_space;
    struct node    *nodeptr = nodes;
    void   print_node(const void *, VISIT, int);
    int    i = 0, node_compare(const void *, const void *);

    while (gets(strptr) != NULL && i++ < NODSZ)  {
        /* Set node. */
        nodeptr->string = strptr;
        nodeptr->length = strlen(strptr);
        /* Put node into the tree. */
        (void) tsearch((void *)nodeptr, (void **)&root,
            node_compare);
```

```
            /* Adjust pointers, so we do not overwrite tree. */
            strptr += nodeptr->length + 1;
            nodeptr++;
        }
        twalk(root, print_node);
        return 0;
    }

    /*
     *  This routine compares two nodes, based on an
     *  alphabetical ordering of the string field.
     */
    int
    node_compare(const void *node1, const void *node2)
    {
        return strcmp(((const struct node *) node1)->string,
            ((const struct node *) node2)->string);
    }

    /*
     *  This routine prints out a node, the second time
     *  twalk encounters it or if it is a leaf.
     */
    void
    print_node(const void *ptr, VISIT order, int level)
    {
        const struct node *p = *(const struct node **) ptr;

        if (order == postorder || order == leaf)  {
            (void) printf("string = %s,  length = %d\n",
                p->string, p->length);
        }
    }
```

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 4

The type of argument *key* is changed from **char*** to **const void***.

The function return value is changed from **char*** to **void***.

Arguments to *compar* are formally defined.

The type of argument *rootp* is changed from **char**** to **void**** for the *tsearch*( ) function.

The type of argument *rootp* is changed from **char**** to **void*const*** for the *tfind*( ) function.

The type of argument *root* is changed from **char*** to **const void***, and the argument list to *action* is formally defined for the *twalk*( ) function.

Various minor wording changes are made in the DESCRIPTION to improve clarity and accuracy. In particular, additional notes are added about constraints on the first argument to *twalk*( ).

The sample code in the EXAMPLES section is updated to use ISO C standard syntax. Also the definition of the *root* and *argv* items is changed.

The paragraph in the APPLICATION USAGE section about casts is removed.

Issue 5

The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *tdelete*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

46823 **NAME**

46824 telldir — current location of a named directory stream

46825 **SYNOPSIS**

46826 XSI
```
#include <dirent.h>

long telldir(DIR *dirp);
```

46827 |

46828 |

46829 **DESCRIPTION**

46830 The *telldir*( ) function obtains the current location associated with the directory stream specified
46831 by *dirp*.

46832 If the most recent operation on the directory stream was a *seekdir*( ), the directory position
46833 returned from the *telldir*( ) shall be the same as that supplied as a *loc* argument for *seekdir*( ).

46834 **RETURN VALUE**

46835 Upon successful completion, *telldir*( ) shall return the current location of the specified directory
46836 stream.

46837 **ERRORS**

46838 No errors are defined.

46839 **EXAMPLES**

46840 None.

46841 **APPLICATION USAGE**

46842 None.

46843 **RATIONALE**

46844 None.

46845 **FUTURE DIRECTIONS**

46846 None.

46847 **SEE ALSO**

46848 *opendir*( ), *readdir*( ), *seekdir*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<dirent.h>** |

46849 **CHANGE HISTORY**

46850 First released in Issue 2.

46851 **Issue 4**

46852 The **<sys/types.h>** header is removed from the SYNOPSIS section.

46853 The function return value is expanded to **long**. |

46854 **Issue 4, Version 2**

46855 The DESCRIPTION is updated for X/OPEN UNIX conformance to indicate that a call to *telldir*( )
46856 immediately following a call to *seekdir*( ), returns the *loc* value passed to the *seekdir*( ) call.

NAME

tempnam — create a name for a temporary file

SYNOPSIS

XSI
```
#include <stdio.h>

char *tempnam(const char *dir, const char *pfx);
```

DESCRIPTION

The *tempnam*( ) function generates a path name that may be used for a temporary file.

The *tempnam*( ) function allows the user to control the choice of a directory. The *dir* argument points to the name of the directory in which the file is to be created. If *dir* is a null pointer or points to a string which is not a name for an appropriate directory, the path prefix defined as P_tmpdir in the **<stdio.h>** header is used. If that directory is not accessible, an implementation-defined directory may be used.

Many applications prefer their temporary files to have certain initial letter sequences in their names. The *pfx* argument should be used for this. This argument may be a null pointer or point to a string of up to five bytes to be used as the beginning of the file name.

Some implementations of *tempnam*( ) may use *tmpnam*( ) internally. On such implementations, if called more than {TMP_MAX} times in a single process, the behavior is implementation-defined.

RETURN VALUE

Upon successful completion, *tempnam*( ) shall allocate space for a string, put the generated path name in that space, and return a pointer to it. The pointer shall be suitable for use in a subsequent call to *free*( ). Otherwise, it shall return a null pointer and set *errno* to indicate the error.

ERRORS

The *tempnam*( ) function shall fail if:

[ENOMEM] Insufficient storage space is available.

EXAMPLES

Generating a Path Name

The following example generates a path name for a temporary file in directory **/tmp**, with the prefix *file*. After the file name has been created, the call to *free*( ) deallocates the space used to store the file name.

```
#include <stdio.h>
#include <stdlib.h>
...
char *directory = "/tmp";
char *fileprefix = "file";
char *file;

file = tempnam(directory, fileprefix);
free(file);
```

APPLICATION USAGE

This function only creates path names. It is the application's responsibility to create and remove the files. Between the time a path name is created and the file is opened, it is possible for some other process to create a file with the same name. Applications may find *tmpfile*( ) more useful.

46900 **RATIONALE**

46901 None.

46902 **FUTURE DIRECTIONS**

46903 None.

46904 **SEE ALSO**

46905 *fopen*( ), *free*( ), *open*( ), *tmpfile*( ), *tmpnam*( ), *unlink*( ), the Base Definitions volume of |
46906 IEEE Std. 1003.1-200x, **<stdio.h>** |

46907 **CHANGE HISTORY**

46908 First released in Issue 1. Derived from Issue 1 of the SVID. |

46909 **Issue 4**

46910 The type of arguments *dir* and *pfx* is changed from **char*** to **const char***.

46911 The DESCRIPTION is changed to indicate that *pfx* is treated as a string of bytes and not as a
46912 string of (possibly multi-byte) characters.

46913 The second paragraph of the APPLICATION USAGE section is expanded.

46914 **Issue 5**

46915 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
46916 previous issues.

NAME

tfind — search binary search tree

SYNOPSIS

XSI

```
#include <search.h>

void *tfind(const void *key, void *const *rootp,
    int (*compar)(const void *, const void *));
```

DESCRIPTION

Refer to *tdelete*( ).

46926 **NAME**
46927 tgamma, tgammaf, tgammal — compute gamma( ) function

46928 **SYNOPSIS**
46929 `#include <math.h>`

46930 `double tgamma(double x);`
46931 `float tgammaf(float x);`
46932 `long double tgammal(long double x);`

46933 **DESCRIPTION**
46934 CX The functionality described on this reference page is aligned with the ISO C standard. Any
46935 conflict between the requirements described here and the ISO C standard is unintentional. This
46936 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

46937 These functions shall compute the *gamma*( ) function of *x*.

46938 An application wishing to check for error situations should set *errno* to 0 before calling these
46939 functions. If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

46940 **RETURN VALUE**
46941 Upon successful completion, these functions shall return *Gamma*(*x*).

46942 If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

46943 If *x* is a negative integer, or if the result cannot be represented when *x* is 0, either HUGE_VAL or
46944 NaN shall be returned and *errno* shall be set to [EDOM].

46945 If the magnitude of *x* is too large or too small, ±HUGE_VAL shall be returned and *errno* shall be
46946 set to [ERANGE].

46947 **ERRORS**
46948 These functions shall fail if:

46949 [EDOM] The value of *x* is negative or the result cannot be represented when *x* is zero.

46950 [ERANGE] The magnitude of *x* is too large or too small.

46951 These functions may fail if:

46952 [EDOM] The value of *x* is NaN.

46953 **EXAMPLES**
46954 None.

46955 **APPLICATION USAGE**
46956 None.

46957 **RATIONALE**
46958 This function is named *tgamma*( ) in order to avoid conflicts with the historical *gamma*( ) and
46959 *lgamma*( ) functions.

46960 **FUTURE DIRECTIONS**
46961 None.

46962 **SEE ALSO**
46963 *lgamma*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

46964 **CHANGE HISTORY**
46965 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

time — get time

SYNOPSIS

```
#include <time.h>

time_t time(time_t *tloc);
```

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.

CX The *time*( ) function returns the value of time in seconds since the Epoch.

The *tloc* argument points to an area where the return value is also stored. If *tloc* is a null pointer, no value is stored.

RETURN VALUE

Upon successful completion, *time*( ) shall return the value of time. Otherwise, (**time_t**)–1 shall be returned.

ERRORS

No errors are defined.

EXAMPLES

Getting the Current Time

The following example uses the *time*( ) function to calculate the time elapsed, in seconds, since January 1, 1970 0:00 UTC, *localtime*( ) to convert that value to a broken-down time, and *asctime*( ) to convert the broken-down time values into a printable string.

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

main()
{
time_t result;

    result = time(NULL);
    printf("%s%ld secs since the Epoch\n",
        asctime(localtime(&result)),
            (long)result);
    return(0);
}
```

This example writes the current time to *stdout* in a form like this:

```
Wed Jun 26 10:32:15 1996
835810335 secs since the Epoch
```

47002 **Timing an Event**

47003 The following example gets the current time, prints it out in the user's format, and prints the
47004 number of minutes to an event being timed.

```
47005 #include <time.h>
47006 #include <stdio.h>
47007 ...
47008 time_t now;
47009 int minutes_to_event;
47010 ...
47011 time(&now);
47012 printf("The time is ");
47013 puts(asctime(localtime(&now)));
47014 printf("There are %d minutes to the event.\n",
47015     minutes_to_event);
47016 ...
```

47017 **APPLICATION USAGE**

47018 None.

47019 **RATIONALE**

47020 The *time*( ) function returns a value in seconds (type **time_t**) while *times*( ) returns a set of values
47021 in clock ticks (type **clock_t**). Some historical implementations, such as 4.3 BSD, have
47022 mechanisms capable of returning more precise times (see below). A generalized timing scheme
47023 to unify these various timing mechanisms has been proposed but not adopted.

47024 Implementations in which **time_t** is a 32-bit signed integer (many historical implementations)
47025 fail in the year 2038. This issue of this volume of IEEE Std. 1003.1-200x does not address this
47026 problem. However, the use of the **time_t** type is mandated in order to ease the eventual fix.

47027 The use of the **<time.h>**, header instead of **<sys/types.h>**, allows compatibility with the ISO C
47028 standard.

47029 Many historical implementations (including Version 7) and the 1984 /usr/group standard use
47030 **long** instead of **time_t**. This volume of IEEE Std. 1003.1-200x uses the latter type in order to
47031 agree with the ISO C standard.

47032 4.3 BSD includes *time*( ) only as an alternate function to the more flexible *gettimeofday*( ) function. |

47033 **FUTURE DIRECTIONS**

47034 In a future version of this volume of IEEE Std. 1003.1-200x, **time_t** is likely to be required to be
47035 capable of representing times far in the future. Whether this will be mandated as a 64-bit type or
47036 a requirement that a specific date in the future be representable (for example, 10000 AD) is not
47037 yet determined. Systems purchased after the approval of this volume of IEEE Std. 1003.1-200x
47038 should be evaluated to determine whether their lifetime will extend past 2038.

47039 **SEE ALSO**

47040 *asctime*( ), *clock*( ), *ctime*( ), *difftime*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *utime*( ), |
47041 the Base Definitions volume of IEEE Std. 1003.1-200x, **<time.h>** |

47042 **CHANGE HISTORY**

47043 First released in Issue 1. Derived from Issue 1 of the SVID. |

47044 **Issue 4**

47045 The following change is incorporated for alignment with the ISO POSIX-1 standard:

47046 • The RETURN VALUE section is updated to indicate that (**time_t**)–1 is returned on error.

47047 **Issue 6**

47048 Extensions beyond the ISO C standard are now marked.

47049 **NAME**

47050 timer_create — create a per-process timer (**REALTIME**)

47051 **SYNOPSIS**

47052 TMR
```
#include <signal.h>
#include <time.h>

int timer_create(clockid_t clockid, struct sigevent *restrict evp,
    timer_t *restrict timerid);
```

47053

47054

47055

47056

47057 **DESCRIPTION**

47058 The *timer_create*( ) function shall create a per-process timer using the specified clock, *clock_id*, as
47059 the timing base. The *timer_create*( ) function returns, in the location referenced by *timerid*, a timer
47060 ID of type **timer_t** used to identify the timer in timer requests. This timer ID shall be unique
47061 within the calling process until the timer is deleted. The particular clock, *clock_id*, is defined in
47062 **<time.h>**. The timer whose ID is returned shall be in a disarmed state upon return from
47063 *timer_create*( ).

47064 The *evp* argument, if non-NULL, points to a **sigevent** structure. This structure, allocated by the
47065 application, defines the asynchronous notification to occur as specified in Section 2.4.1 (on page
47066 528) when the timer expires. If the *evp* argument is NULL, the effect is as if the *evp* argument
47067 pointed to a **sigevent** structure with the *sigev_notify* member having the value SIGEV_SIGNAL,
47068 the *sigev_signo* having a default signal number, and the *sigev_value* member having the value of
47069 the timer ID.

47070 Each implementation shall define a set of clocks that can be used as timing bases for per-process
47071 MON timers. All implementations shall support a *clock_id* of CLOCK_REALTIME. If the Monotonic
47072 Clock option is supported, implementations shall support a *clock_id* of CLOCK_MONOTONIC.

47073 Per-process timers shall not be inherited by a child process across a *fork*( ) and shall be disarmed
47074 and deleted by an *exec*.

47075 CPT If _POSIX_CPUTIME is defined, implementations shall support *clock_id* values representing the
47076 CPU-time clock of the calling process.

47077 TCT If _POSIX_THREAD_CPUTIME is defined, implementations shall support *clock_id* values
47078 representing the CPU-time clock of the calling thread.

47079 CPT|TCT It is implementation-defined whether a *timer_create*( ) function will succeed if the value defined
47080 by *clock_id* corresponds to the CPU-time clock of a process or thread different from the process
47081 or thread invoking the function.

47082 **RETURN VALUE**

47083 If the call succeeds, *timer_create*( ) shall return zero and update the location referenced by *timerid*
47084 to a **timer_t**, which can be passed to the per-process timer calls. If an error occurs, the function
47085 shall return a value of –1 and set *errno* to indicate the error. The value of *timerid* is undefined if
47086 an error occurs.

47087 **ERRORS**

47088 The *timer_create*( ) function shall fail if:

47089 [EAGAIN] The system lacks sufficient signal queuing resources to honor the request.

47090 [EAGAIN] The calling process has already created all of the timers it is allowed by this
47091 implementation.

47092 [EINVAL] The specified clock ID is not defined.

47093 CPT|TCT [ENOTSUP] The implementation does not support the creation of a timer attached to the
47094 CPU-time clock that is specified by *clock_id* and associated with a process or
47095 thread different from the process or thread invoking *timer_create*( ).

47096 **EXAMPLES**

47097 None.

47098 **APPLICATION USAGE**

47099 None.

47100 **RATIONALE**

47101 **Periodic Timer Overrun and Resource Allocation**

47102 The specified timer facilities may deliver realtime signals (that is, queued signals) on
47103 implementations that support this option. Because realtime applications cannot afford to lose
47104 notifications of asynchronous events, like timer expirations or asynchronous I/O completions, it
47105 must be possible to ensure that sufficient resources exist to deliver the signal when the event
47106 occurs. In general, this is not a difficulty because there is a one-to-one correspondence between a
47107 request and a subsequent signal generation. If the request cannot allocate the signal delivery
47108 resources, it can fail the call with an [EAGAIN] error.

47109 Periodic timers are a special case. A single request can generate an indeterminate number of
47110 signals. This is not a problem if the requesting process can service the signals as fast as they are
47111 generated, thus making the signal delivery resources available for delivery of subsequent
47112 periodic timer expiration signals. But, in general, this cannot be assured—processing of periodic
47113 timer signals may ‘‘overrun’’; that is, subsequent periodic timer expirations may occur before the
47114 currently pending signal has been delivered.

47115 Also, for signals, according to the POSIX.1-1990 standard, if subsequent occurrences of a
47116 pending signal are generated, it is implementation-defined whether a signal is delivered for each
47117 occurrence. This is not adequate for some realtime applications. So a mechanism is required to
47118 allow applications to detect how many timer expirations were delayed without requiring an
47119 indefinite amount of system resources to store the delayed expirations.

47120 The specified facilities provide for an overrun count. The overrun count is defined as the number
47121 of extra timer expirations that occurred between the time a timer expiration signal is generated
47122 and the time the signal is delivered. The signal-catching function, if it is concerned with
47123 overruns, can retrieve this count on entry. With this method, a periodic timer only needs one
47124 ‘‘signal queuing resource’’ that can be allocated at the time of the *timer_create*( ) function call.

47125 A function is defined to retrieve the overrun count so that an application need not allocate static
47126 storage to contain the count, and an implementation need not update this storage
47127 asynchronously on timer expirations. But, for some high-frequency periodic applications, the
47128 overhead of an additional system call on each timer expiration may be prohibitive. The
47129 functions, as defined, permit an implementation to maintain the overrun count in user space,
47130 associated with the *timerid*. The *timer_getoverrun*( ) function can then be implemented as a macro
47131 that uses the *timerid* argument (which may just be a pointer to a user space structure containing
47132 the counter) to locate the overrun count with no system call overhead. Other implementations,
47133 less concerned with this class of applications, can avoid the asynchronous update of user space
47134 by maintaining the count in a system structure at the cost of the extra system call to obtain it.

47135 **Timer Expiration Signal Parameters**

47136 The Realtime Signals Extension option supports an application-specific datum that is delivered
47137 to the extended signal handler. This value is explicitly specified by the application, along with
47138 the signal number to be delivered, in a **sigevent** structure. The type of the application-defined
47139 value can be either an integer constant or a pointer. This explicit specification of the value, as
47140 opposed to always sending the timer ID, was selected based on existing practice.

47141 It is common practice for realtime applications (on non-POSIX systems or realtime extended
47142 POSIX systems) to use the parameters of event handlers as the case label of a switch statement
47143 or as a pointer to an application-defined data structure. Because *timer_id*s are dynamically
47144 allocated by the *timer_create*( ) function, they can be used for neither of these functions without
47145 additional application overhead in the signal handler; for example, to search an array of saved
47146 timer IDs to associate the ID with a constant or application data structure.

47147 **FUTURE DIRECTIONS**

47148 None.

47149 **SEE ALSO**

47150 *clock_getres*( ), *timer_delete*( ), *timer_getoverrun*( ), the Base Definitions volume of |
47151 IEEE Std. 1003.1-200x, **<time.h>** |

47152 **CHANGE HISTORY**

47153 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

47154 **Issue 6**

47155 The *timer_create*( ) function is marked as part of the Timers option. |

47156 The [ENOSYS] error condition has been removed as stubs need not be provided if an
47157 implementation does not support the Timers option. |

47158 CPU-time clocks are added for alignment with IEEE Std. 1003.1d-1999.

47159 The DESCRIPTION is updated for alignment with IEEE Std. 1003.1j-2000 by adding the
47160 requirement for the CLOCK_MONOTONIC clock under the Monotonic Clock option. |

47161 The **restrict** keyword is added to the *timer_create*( ) prototype for alignment with the |
47162 ISO/IEC 9899:1999 standard. |

47163 **NAME**

47164 timer_delete — delete a per-process timer (**REALTIME**)

47165 **SYNOPSIS**

47166 TMR
```
#include <time.h>
```

47167
```
int timer_delete(timer_t timerid);
```

47168

47169 **DESCRIPTION**

47170 The *timer_delete*( ) function deletes the specified timer, *timerid*, previously created by the
47171 *timer_create*( ) function. If the timer is armed when *timer_delete*( ) is called, the behavior shall be
47172 as if the timer is automatically disarmed before removal. The disposition of pending signals for
47173 the deleted timer is unspecified.

47174 **RETURN VALUE**

47175 If successful, the *timer_delete*( ) function shall return a value of zero. Otherwise, the function shall
47176 return a value of –1 and set *errno* to indicate the error.

47177 **ERRORS**

47178 The *timer_delete*( ) function shall fail if:

47179 [EINVAL] The timer ID specified by *timerid* is not a valid timer ID. |

47180 **EXAMPLES**

47181 None.

47182 **APPLICATION USAGE**

47183 None.

47184 **RATIONALE**

47185 None.

47186 **FUTURE DIRECTIONS**

47187 None.

47188 **SEE ALSO**

47189 *timer_create*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<time.h>** |

47190 **CHANGE HISTORY**

47191 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

47192 **Issue 6**

47193 The *timer_delete*( ) function is marked as part of the Timers option. |

47194 The [ENOSYS] error condition has been removed as stubs need not be provided if an
47195 implementation does not support the Timers option. |

47196 **NAME**

47197 timer_getoverrun, timer_gettime, timer_settime — per-process timers (**REALTIME**)

47198 **SYNOPSIS**

47199 TMR
```
#include <time.h>

int timer_getoverrun(timer_t timerid);
int timer_gettime(timer_t timerid, struct itimerspec *value);
int timer_settime(timer_t timerid, int flags,
    const struct itimerspec *restrict value,
    struct itimerspec *restrict ovalue);
```

47206 **DESCRIPTION**

47207 Only a single signal shall be queued to the process for a given timer at any point in time. When a
47208 timer for which a signal is still pending expires, no signal shall be queued, and a timer overrun
47209 RTS shall occur. When a timer expiration signal is delivered to or accepted by a process, if the
47210 implementation supports the Realtime Signals Extension, the *timer_getoverrun*( ) function returns
47211 the timer expiration overrun count for the specified timer. The overrun count returned contains
47212 the number of extra timer expirations that occurred between the time the signal was generated
47213 (queued) and when it was delivered or accepted, up to but not including an implementation-
47214 defined maximum of {DELAYTIMER_MAX}. If the number of such extra expirations is greater
47215 than or equal to {DELAYTIMER_MAX}, then the overrun count is set to {DELAYTIMER_MAX}.
47216 The value returned by *timer_getoverrun*( ) applies to the most recent expiration signal delivery or
47217 acceptance for the timer. If no expiration signal has been delivered for the timer, or if the
47218 Realtime Signals Extension is not supported, the return value of *timer_getoverrun*( ) is
47219 unspecified.

47220 The *timer_gettime*( ) function shall store the amount of time until the specified timer, *timerid*,
47221 expires and the reload value of the timer into the space pointed to by the *value* argument. The
47222 *it_value* member of this structure shall contain the amount of time before the timer expires, or
47223 zero if the timer is disarmed. This value is returned as the interval until timer expiration, even if
47224 the timer was armed with absolute time. The *it_interval* member of *value* shall contain the reload
47225 value last set by *timer_settime*( ).

47226 The *timer_settime*( ) function shal set the time until the next expiration of the timer specified by
47227 *timerid* from the *it_value* member of the *value* argument and arms the timer if the *it_value*
47228 member of *value* is non-zero. If the specified timer was already armed when *timer_settime*( ) is
47229 called, this call shall reset the time until next expiration to the *value* specified. If the *it_value*
47230 member of *value* is zero, the timer shall be disarmed. The effect of disarming or resetting a timer
47231 on pending expiration notifications is unspecified.

47232 If the flag TIMER_ABSTIME is not set in the argument *flags*, *timer_settime*( ) behaves as if the
47233 time until next expiration is set to be equal to the interval specified by the *it_value* member of
47234 *value*. That is, the timer shall expire in *it_value* nanoseconds from when the call is made. If the
47235 flag TIMER_ABSTIME is set in the argument *flags*, *timer_settime*( ) behaves as if the time until
47236 next expiration is set to be equal to the difference between the absolute time specified by the
47237 *it_value* member of *value* and the current value of the clock associated with *timerid*. That is, the
47238 timer shal expire when the clock reaches the value specified by the *it_value* member of *value*. If
47239 the specified time has already passed, the function shall succeed and the expiration notification
47240 shall be made.

47241 The reload value of the timer is set to the value specified by the *it_interval* member of *value*.
47242 When a timer is armed with a non-zero *it_interval*, a periodic (or repetitive) timer is specified.

Time values that are between two consecutive non-negative integer multiples of the resolution of the specified timer shall be rounded up to the larger multiple of the resolution. Quantization error shall not cause the timer to expire earlier than the rounded time value.

If the argument *ovalue* is not NULL, the function *timer_settime*( ) shall store, in the location referenced by *ovalue*, a value representing the previous amount of time before the timer would have expired, or zero if the timer was disarmed, together with the previous timer reload value.

Notes to Reviewers

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

D1, XSH, ERN 388 suggests rewording ‘‘are subject to the resolution of the timer’’ to ‘‘may have been rounded to the resolution of the timer’’.

The members of *ovalue* are subject to the resolution of the timer, and they are the same values that would be returned by a *timer_gettime*( ) call at that point in time.

RETURN VALUE

If the *timer_getoverrun*( ) function succeeds, it shall return the timer expiration overrun count as explained above.

If the *timer_gettime*( ) or *timer_settime*( ) functions succeed, a value of 0 shall be returned.

If an error occurs for any of these functions, the value –1 shall be returned, and *errno* set to indicate the error.

ERRORS

The *timer_getoverrun*( ), *timer_gettime*( ), and *timer_settime*( ) functions shall if:

[EINVAL] The *timerid* argument does not correspond to an ID returned by *timer_create*( ) but not yet deleted by *timer_delete*( ).

The *timer_settime*( ) function shall fail if:

[EINVAL] A *value* structure specified a nanosecond value less than zero or greater than or equal to 1,000 million, and the *it_value* member of that structure did not specify zero seconds and nanoseconds.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *clock_settime*( ), *timer_settime*( ), and *nanosleep*( ) functions are defined to truncate specified time values down to the resolution supported by the implementation. Values are truncated when set because this appears to be existing practice, and it does not seem reasonable to require an error in this case. Note that this is symmetric with the truncation that occurs when reading the time via *clock_gettime*( ) or *timer_gettime*( ) at a time that is not an integral multiple of the clock or timer resolution.

This volume of IEEE Std. 1003.1-200x defines functions that allow an application to determine the implementation-supported resolution for the clocks and requires an implementation to document the resolution supported for timers and *nanosleep*( ) if they differ from the supported clock resolution. This is more of a procurement issue than a runtime application issue.

47284 **FUTURE DIRECTIONS**

47285 None.

47286 **SEE ALSO**

47287 *clock_getres*( ), *timer_create*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<time.h>**

47288 **CHANGE HISTORY**

47289 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

47290 **Issue 6**

47291 The *timer_getoverrun*( ), *timer_gettime*( ), and *timer_settime*( ) functions are marked as part of the
47292 Timers option.

47293 The [ENOSYS] error condition has been removed as stubs need not be provided if an
47294 implementation does not support the Timers option.

47295 The [EINVAL] error condition is updated to include the following: ''and the *it_value* member of
47296 that structure did not specify zero seconds and nanoseconds.'' This change is for IEEE PASC
47297 Interpretation 1003.1 #89.

47298 The DESCRIPTION for *timer_getoverrun*( ) is updated to clarify that ''If no expiration signal has
47299 been delivered for the timer, or if the Realtime Signals Extension is not supported, the return
47300 value of *timer_getoverrun*( ) is unspecified''.

47301 The **restrict** keyword is added to the *timer_settime*( ) prototype for alignment with the
47302 ISO/IEC 9899:1999 standard.

NAME

times — get process and waited-for child process times

SYNOPSIS

```
#include <sys/times.h>

clock_t times(struct tms *buffer);
```

DESCRIPTION

The *times*( ) function shall fill the **tms** structure pointed to by *buffer* with time-accounting information. The structure **tms** is defined in **<sys/times.h>**.

All times are measured in terms of the number of clock ticks used.

The times of a terminated child process are included in the *tms_cutime* and *tms_cstime* elements of the parent when *wait*( ) or *waitpid*( ) returns the process ID of this terminated child. If a child process has not waited for its children, their times shall not be included in its times.

- The *tms_utime* structure member is the CPU time charged for the execution of user instructions of the calling process.
- The *tms_stime* structure member is the CPU time charged for execution by the system on behalf of the calling process.
- The *tms_cutime* structure member is the sum of the *tms_utime* and *tms_cutime* times of the child processes.
- The *tms_cstime* structure member is the sum of the *tms_stime* and *tms_cstime* times of the child processes.

RETURN VALUE

Upon successful completion, *times*( ) shall return the elapsed real time, in clock ticks, since an arbitrary point in the past (for example, system start-up time). This point does not change from one invocation of *times*( ) within the process to another. The return value may overflow the possible range of type **clock_t**. If *times*( ) fails, (**clock_t**)–1 shall be returned and *errno* set to indicate the error.

ERRORS

No errors are defined.

EXAMPLES

Timing a Database Lookup

The following example defines two functions, *start_clock*( ) and *end_clock*( ), that are used to time a lookup. It also defines variables of type **clock_t** and **tms** to measure the duration of transactions. The *start_clock*( ) function saves the beginning times given by the *times*( ) function. The *end_clock*( ) function gets the ending times and prints the difference between the two times.

```
#include <sys/times.h>
#include <stdio.h>
...
void start_clock(void);
void end_clock(char *msg);
...
static clock_t st_time;
static clock_t en_time;
static struct tms st_cpu;
static struct tms en_cpu;
```

```
...
void
start_clock()
{
    st_time = times(&st_cpu);
}

void
end_clock(char *msg)
{
    en_time = times(&en_cpu);

    printf(msg);
    printf("Real Time: %ld, User Time %ld, System Time %ld\n",
        en_time - st_time,
        en_cpu.tms_utime - st_cpu.tms_utime,
        en_cpu.tms_stime - st_cpu.tms_stime);
}
```

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 multiprocessor 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

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.

Issue 4

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- All references to the constant CLK_TCK are removed.

47392 • The RETURN VALUE section is updated to indicate that (**clock_t**)−1 is returned on error.

47393 **NAME**

47394 timezone — difference from UTC and local standard time

47395 **SYNOPSIS**

47396 `#include <time.h>`

47397 XSI `extern long timezone;` |

47398 |

47399 **DESCRIPTION**

47400 Refer to *tzset*( ).

NAME

tmpfile — create a temporary file

SYNOPSIS

```
#include <stdio.h>

FILE *tmpfile(void);
```

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 *tmpfile*( ) function shall create a temporary file and open a corresponding stream. The file is automatically deleted when all references to the file are closed. The file is opened as in *fopen*( ) for update (*w+*).

CX The largest value that can be represented correctly in an object of type **off_t** is established as the offset maximum in the open file description.

In some implementations, a permanent file may be left behind if the process calling *tmpfile*( ) is killed while it is processing a call to *tmpfile*( ).

An error message may be written to standard error if the stream cannot be opened.

RETURN VALUE

Upon successful completion, *tmpfile*( ) shall return a pointer to the stream of the file that is created. Otherwise, it shall return a null pointer CX and set *errno* to indicate the error.

ERRORS

The *tmpfile*( ) function shall fail if:

CX [EINTR] A signal was caught during *tmpfile*( ). |

CX [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process. |

CX [ENFILE] The maximum allowable number of files is currently open in the system. |

CX [ENOSPC] The directory or file system which would contain the new file cannot be expanded. |

CX [EOVERFLOW] The file is a regular file and the size of the file cannot be represented correctly in an object of type **off_t**. |

The *tmpfile*( ) function may fail if:

CX [EMFILE] {FOPEN_MAX} streams are currently open in the calling process. |

CX [ENOMEM] Insufficient storage space is available. |

EXAMPLES

Creating a Temporary File

The following example creates a temporary file for update, and returns a pointer to a stream for the created file in the *fp* variable.

```
#include <stdio.h>
...
FILE *fp;

fp = tmpfile ();
```

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( ), *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 4

The argument list is explicitly defined as **void**.

The [EINTR] error is moved to the ''fails'' part of the ERRORS section; [EMFILE], [ENFILE], and [ENOSPC] are no longer marked as extensions; [EACCES], [ENOTDIR], and [EROFS] are removed; and the [EMFILE] error in the ''may fail'' part is marked as an extension.

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 now 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 [EMFILE] optional error condition is added.

The APPLICATION USAGE section is added for alignment with the ISO/IEC 9899:1999 standard.

NAME

tmpnam — create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>

char *tmpnam(char *s);
```

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 *tmpnam*( ) function shall generate a string that is a valid file name and that is not the same as the name of an existing file. The function is potentially capable of generating {TMP_MAX} different strings, but any or all of them may already be in use by existing files and thus not be suitable return values.

The *tmpnam*( ) function generates a different string each time it is called from the same process, up to {TMP_MAX} times. If it is called more than {TMP_MAX} times, the behavior is implementation-defined.

The implementation shall behave as if no function defined in this volume of IEEE Std. 1003.1-200x calls *tmpnam*( ).

CX If the application uses any of the functions guaranteed to be available if either _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS is defined, the application shall ensure that the *tmpnam*( ) function is called with a non-NULL parameter.

RETURN VALUE

Upon successful completion, *tmpnam*( ) shall return a pointer to a string. If no suitable string can be generated, the *tmpnam*( ) function shall return a null pointer.

If the argument *s* is a null pointer, *tmpnam*( ) shall leave its result in an internal static object and return a pointer to that object. Subsequent calls to *tmpnam*( ) may modify the same object. If the argument *s* is not a null pointer, it is presumed to point to an array of at least {L_tmpnam} **char**s; *tmpnam*( ) writes its result in that array and returns the argument as its value.

ERRORS

No errors are defined.

EXAMPLES

Generating a File Name

The following example generates a unique file name and stores it in the array pointed to by *ptr*.

```
#include <stdio.h>
...
char filename[L_tmpnam+1];
char *ptr;

ptr = tmpnam(filename);
```

APPLICATION USAGE

This function only creates file names. It is the application's responsibility to create and remove the files.

Between the time a path name is created and the file is opened, it is possible for some other process to create a file with the same name. Applications may find *tmpfile*( ) more useful.

47517 **RATIONALE**

47518 None.

47519 **FUTURE DIRECTIONS**

47520 None.

47521 **SEE ALSO**

47522 *fopen*( ), *open*( ), *tempnam*( ), *tmpfile*( ), *unlink*( ), the Base Definitions volume of |
47523 IEEE Std. 1003.1-200x, **<stdio.h>** |

47524 **CHANGE HISTORY**

47525 First released in Issue 1. Derived from Issue 1 of the SVID. |

47526 **Issue 5**

47527 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

47528 **Issue 6**

47529 Extensions beyond the ISO C standard are now marked.

47530 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements. |

47531 The DESCRIPTION is expanded for alignment with the ISO / IEC 9899: 1999 standard. |

47532 **NAME**

47533 toascii — translate integer to a 7-bit ASCII character

47534 **SYNOPSIS**

47535 XSI
```
#include <ctype.h>
```
47536
```
int toascii(int c);
```
47537

47538 **DESCRIPTION**

47539 The *toascii*( ) function shall convert its argument into a 7-bit ASCII character.

47540 **RETURN VALUE**

47541 The *toascii*( ) function shall return the value (*c* &0x7f).

47542 **ERRORS**

47543 No errors are returned.

47544 **EXAMPLES**

47545 None.

47546 **APPLICATION USAGE**

47547 None.

47548 **RATIONALE**

47549 None.

47550 **FUTURE DIRECTIONS**

47551 None.

47552 **SEE ALSO**

47553 *isascii*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>** |

47554 **CHANGE HISTORY**

47555 First released in Issue 1. Derived from Issue 1 of the SVID. |

47556 **NAME**

47557 tolower — transliterate uppercase characters to lowercase

47558 **SYNOPSIS**

47559 `#include <ctype.h>`

47560 `int tolower(int c);`

47561 **DESCRIPTION**

47562 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47563 conflict between the requirements described here and the ISO C standard is unintentional. This
47564 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

47565 The *tolower*( ) function has as a domain a type **int**, the value of which is representable as an
47566 **unsigned char** or the value of EOF. If the argument has any other value, the behavior is
47567 undefined. If the argument of *tolower*( ) represents an uppercase letter, and there exists a
47568 CX corresponding lowercase letter (as defined by character type information in the program locale
47569 category *LC_CTYPE*), the result is the corresponding lowercase letter. All other arguments in the
47570 domain are returned unchanged.

47571 **RETURN VALUE**

47572 Upon successful completion, *tolower*( ) shall return the lowercase letter corresponding to the
47573 argument passed; otherwise, it shall return the argument unchanged.

47574 **ERRORS**

47575 No errors are defined.

47576 **EXAMPLES**

47577 None.

47578 **APPLICATION USAGE**

47579 None.

47580 **RATIONALE**

47581 None.

47582 **FUTURE DIRECTIONS**

47583 None.

47584 **SEE ALSO**

47585 *setlocale*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>**, the Base Definitions |
47586 volume of IEEE Std. 1003.1-200x, Chapter 7, Locale |

47587 **CHANGE HISTORY**

47588 First released in Issue 1. Derived from Issue 1 of the SVID. |

47589 **Issue 4**

47590 Reference to ‘‘shift information’’ is replaced by ‘‘character type information’’.

47591 The RETURN VALUE section is added.

47592 **Issue 6**

47593 Extensions beyond the ISO C standard are now marked.

47594 **NAME**

47595 toupper — transliterate lowercase characters to uppercase

47596 **SYNOPSIS**

47597 `#include <ctype.h>`

47598 `int toupper(int c);`

47599 **DESCRIPTION**

47600 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47601 conflict between the requirements described here and the ISO C standard is unintentional. This
47602 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

47603 The *toupper*( ) function has as a domain a type **int**, the value of which is representable as an
47604 **unsigned char** or the value of EOF. If the argument has any other value, the behavior is
47605 undefined. If the argument of *toupper*( ) represents a lowercase letter, and there exists a
47606 CX corresponding uppercase letter (as defined by character type information in the program locale
47607 category *LC_CTYPE*), the result is the corresponding uppercase letter. All other arguments in the
47608 domain are returned unchanged.

47609 **RETURN VALUE**

47610 Upon successful completion, *toupper*( ) shall return the uppercase letter corresponding to the
47611 argument passed.

47612 **ERRORS**

47613 No errors are defined.

47614 **EXAMPLES**

47615 None.

47616 **APPLICATION USAGE**

47617 None.

47618 **RATIONALE**

47619 None.

47620 **FUTURE DIRECTIONS**

47621 None.

47622 **SEE ALSO**

47623 *setlocale*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ctype.h>**, the Base Definitions |
47624 volume of IEEE Std. 1003.1-200x, Chapter 7, Locale |

47625 **CHANGE HISTORY**

47626 First released in Issue 1. Derived from Issue 1 of the SVID. |

47627 **Issue 4**

47628 Reference to ‘‘shift information’’ is replaced by ‘‘character type information’’.

47629 The RETURN VALUE section is added.

47630 **Issue 6**

47631 Extensions beyond the ISO C standard are now marked.

47632 **NAME**

47633 towctrans — character transliteration

47634 **SYNOPSIS**

47635 ```
#include <wctype.h>
```

47636 ```
wint_t towctrans(wint_t wc, wctrans_t desc);
```

47637 **DESCRIPTION**

47638 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47639 conflict between the requirements described here and the ISO C standard is unintentional. This
47640 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

47641 The *towctrans*( ) function transliterates the wide-character code *wc* using the mapping described
47642 by *desc*. The current setting of the *LC_CTYPE* category should be the same as during the call to
47643 CX *wctrans*( ) that returned the value *desc*. If the value of *desc* is invalid (that is, not obtained by a
47644 call to *wctrans*( ) or *desc* is invalidated by a subsequent call to *setlocale*( ) that has affected
47645 category *LC_CTYPE*), the result is unspecified.

47646 An application wishing to check for error situations should set *errno* to 0 before calling
47647 *towctrans*( ). If *errno* is non-zero on return, an error has occurred.

47648 **RETURN VALUE**

47649 If successful, the *towctrans*( ) function shall return the mapped value of *wc* using the mapping
47650 described by *desc*. Otherwise, it shall return *wc* unchanged.

47651 **ERRORS**

47652 The *towctrans*( ) function may fail if:

47653 CX [EINVAL] *desc* contains an invalid transliteration descriptor. |

47654 **EXAMPLES**

47655 None.

47656 **APPLICATION USAGE**

47657 The strings `"tolower"` and `"toupper"` are reserved for the standard mapping names. In the
47658 table below, the functions in the left column are equivalent to the functions in the right column.

47659 ```
towlower(wc)   towctrans(wc, wctrans("tolower"))
```
47660 ```
towupper(wc)    towctrans(wc, wctrans("toupper"))
```

47661 **RATIONALE**

47662 None.

47663 **FUTURE DIRECTIONS**

47664 None.

47665 **SEE ALSO**

47666 *towlower*( ), *towupper*( ), *wctrans*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
47667 **<wctype.h>**

47668 **CHANGE HISTORY**

47669 First released in Issue 5. Derived from ISO / IEC 9899: 1990 / Amendment 1: 1995 (E). |

47670 **Issue 6**

47671 Extensions beyond the ISO C standard are now marked.

NAME

towlower — transliterate uppercase wide-character code to lowercase

SYNOPSIS

```
#include <wctype.h>

wint_t towlower(wint_t wc);
```

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 *towlower*( ) function has as a domain a type **wint_t**, the value of which the application shall ensure is a character representable as a **wchar_t**, and a wide-character code corresponding to a valid character in the current locale or the value of WEOF. If the argument has any other value, the behavior is undefined. If the argument of *towlower*( ) represents an uppercase wide-character code, and there exists a corresponding lowercase wide-character code (as defined by character type information in the program locale category *LC_CTYPE*), the result is the corresponding lowercase wide-character code. All other arguments in the domain are returned unchanged.

RETURN VALUE

Upon successful completion, *towlower*( ) shall return the lowercase letter corresponding to the argument passed; otherwise, it shall return the argument unchanged.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

setlocale( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

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 now made visible by inclusion of the header **<wctype.h>** rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

47713 **NAME**

47714 towupper — transliterate lowercase wide-character code to uppercase

47715 **SYNOPSIS**

47716 `#include <wctype.h>`

47717 `wint_t towupper(wint_t wc);`

47718 **DESCRIPTION**

47719 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47720 conflict between the requirements described here and the ISO C standard is unintentional. This
47721 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

47722 The *towupper*( ) function has as a domain a type **wint_t**, the value of which the application shall
47723 ensure is a character representable as a **wchar_t**, and a wide-character code corresponding to a
47724 valid character in the current locale or the value of WEOF. If the argument has any other value,
47725 the behavior is undefined. If the argument of *towupper*( ) represents a lowercase wide-character
47726 code, and there exists a corresponding uppercase wide-character code (as defined by character
47727 type information in the program locale category *LC_CTYPE*), the result is the corresponding
47728 uppercase wide-character code. All other arguments in the domain are returned unchanged.

47729 **RETURN VALUE**

47730 Upon successful completion, *towupper*( ) shall return the uppercase letter corresponding to the
47731 argument passed. Otherwise, it shall return the argument unchanged.

47732 **ERRORS**

47733 No errors are defined.

47734 **EXAMPLES**

47735 None.

47736 **APPLICATION USAGE**

47737 None.

47738 **RATIONALE**

47739 None.

47740 **FUTURE DIRECTIONS**

47741 None.

47742 **SEE ALSO**

47743 *setlocale*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the |
47744 Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale |

47745 **CHANGE HISTORY**

47746 First released in Issue 4.

47747 **Issue 5**

47748 The following change has been made in this issue for alignment with
47749 ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

47750 • The SYNOPSIS has been changed to indicate that this function and associated data types are
47751 now made visible by inclusion of the header **<wctype.h>** rather than **<wchar.h>**.

47752 **Issue 6** |

47753 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

47754 **NAME**

47755 trunc, truncf, truncl — round to truncated integer value

47756 **SYNOPSIS**

47757
```
#include <math.h>

double trunc(double x);
float truncf(float x);
long double truncl(long double x);
```

47761 **DESCRIPTION**

47762 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.

47765 These functions shall round their argument to the integer value, in floating format, nearest to but no larger in magnitude than the argument.

47767 An application wishing to check for error situations should set *errno* to 0 before calling these functions. If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

47769 **RETURN VALUE**

47770 Upon successful completion, these functions shall return the truncated integer value.

47771 If *x* is ±Inf, these functions shall return *x*.

47772 If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

47773 **ERRORS**

47774 These functions may fail if:

47775 [EDOM] The value of *x* is NaN.

47776 **EXAMPLES**

47777 None.

47778 **APPLICATION USAGE**

47779 None.

47780 **RATIONALE**

47781 None.

47782 **FUTURE DIRECTIONS**

47783 None.

47784 **SEE ALSO**

47785 The Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>**

47786 **CHANGE HISTORY**

47787 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

47788 **NAME**

47789 truncate — truncate a file to a specified length

47790 **SYNOPSIS**

47791 XSI
```
#include <unistd.h>

int truncate(const char *path, off_t length);
```
47792

47793

47794 **DESCRIPTION**

47795 The *truncate*( ) function shall cause the regular file named by *path* to have a size which shall be
47796 equal to *length* bytes.

47797 If the file previously was larger than *length*, the extra data is discarded. If the file was previously
47798 shorter than *length*, its size is increased, and the extended area appears as if it were zero-filled.

47799 The application shall ensure that the process has write permission for the file.

47800 If the request would cause the file size to exceed the soft file size limit for the process, the
47801 request shall fail and the implementation shall generate the SIGXFSZ signal for the process.

47802 This function shall not modify the file offset for any open file descriptions associated with the
47803 file. Upon successful completion, if the file size is changed, this function shall mark for update
47804 the *st_ctime* and *st_mtime* fields of the file, and the S_ISUID and S_ISGID bits of the file mode
47805 may be cleared.

47806 **RETURN VALUE**

47807 Upon successful completion, *truncate*( ) shall return 0. Otherwise, –1 shall be returned, and *errno*
47808 set to indicate the error.

47809 **ERRORS**

47810 The *truncate*( ) function shall fail if:

47811 [EINTR] A signal was caught during execution.

47812 [EINVAL] The *length* argument was less than 0.

47813 [EFBIG] or [EINVAL]
47814 The *length* argument was greater than the maximum file size.

47815 [EIO] An I/O error occurred while reading from or writing to a file system.

47816 [EACCES] A component of the path prefix denies search permission, or write permission
47817 is denied on the file.

47818 [EISDIR] The named file is a directory.

47819 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
47820 argument.

47821 [ENAMETOOLONG]
47822 The length of the *path* argument exceeds {PATH_MAX} or a path name
47823 component is longer than {NAME_MAX}.

47824 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

47825 [ENOTDIR] A component of the path prefix of *path* is not a directory.

47826 [EROFS] The named file resides on a read-only file system.

47827 The *truncate*( ) function may fail if:

47828 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during |
47829 resolution of the *path* argument. |

47830 [ENAMETOOLONG] |
47831 Path name resolution of a symbolic link produced an intermediate result
47832 whose length exceeds {PATH_MAX}.

47833 **EXAMPLES**

47834 None.

47835 **APPLICATION USAGE**

47836 None.

47837 **RATIONALE**

47838 None.

47839 **FUTURE DIRECTIONS**

47840 None.

47841 **SEE ALSO**

47842 *open*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>** |

47843 **CHANGE HISTORY**

47844 First released in Issue 4, Version 2.

47845 **Issue 5**

47846 Moved from X/OPEN UNIX extension to BASE.

47847 Large File Summit extensions are added.

47848 **Issue 6**

47849 This reference page is split out from the *ftruncate*( ) reference page.

47850 The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This
47851 is since behavior may vary from one file system to another.

47852 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements. |

47853 The wording of the mandatory [ELOOP] error condition is updated, and a second optional |
47854 [ELOOP] error condition is added. |

47855 **NAME**

47856 tsearch — search a binary search tree

47857 **SYNOPSIS**

47858 XSI
```
#include <search.h>

void *tsearch(const void *key, void **rootp,
    int (*compar)(const void *, const void *));
```

47859

47860

47861

47862 **DESCRIPTION**

47863 Refer to *tdelete*( ).

NAME

ttyname, ttyname_r — find path name of a terminal

SYNOPSIS

```
#include <unistd.h>

char *ttyname(int fildes);
int ttyname_r(int fildes, char *name, size_t namesize);
```

DESCRIPTION

The *ttyname*( ) function shall return a pointer to a string containing a null-terminated path name of the terminal associated with file descriptor *fildes*. The return value may point to static data whose content is overwritten by each call.

The *ttyname*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *ttyname_r*( ) function stores the null-terminated path name of the terminal associated with the file descriptor *fildes* in the character array referenced by *name*. The array is *namesize* characters long and should have space for the name and the terminating null character. The maximum length of the terminal name is {TTY_NAME_MAX}.

RETURN VALUE

Upon successful completion, *ttyname*( ) shall return a pointer to a string. Otherwise, a null pointer shall be returned and *errno* set to indicate the error.

TSF If successful, the *ttyname_r*( ) function shall return zero. Otherwise, an error number shall be returned to indicate the error.

ERRORS

The *ttyname*( ) function may fail if:

[EBADF] The *fildes* argument is not a valid file descriptor.

[ENOTTY] The *fildes* argument does not refer to a terminal.

The *ttyname_r*( ) function may fail if:

TSF [EBADF] The *fildes* argument is not a valid file descriptor.

TSF [ENOTTY] The *fildes* argument does not refer to a terminal.

Notes to Reviewers

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

D1, XSH, ERN 397 points out an inconsistency for the [ERANGE] error condition below and suggests this be changed to [E2BIG].

TSF [ERANGE] The value of *namesize* is smaller than the length of the string to be returned including the terminating null character.

47899 **EXAMPLES**

47900 None.

47901 **APPLICATION USAGE**

47902 None.

47903 **RATIONALE**

47904 The term *terminal* is used instead of the historical term *terminal device* in order to avoid a
47905 reference to an undefined term.

47906 The thread-safe version places the terminal name in a user-supplied buffer and returns a non-
47907 zero value if it fails. The non-thread-safe version may return the name in a static data area that
47908 may be overwritten by each call.

47909 **FUTURE DIRECTIONS**

47910 None.

47911 **SEE ALSO**

47912 The Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>** |

47913 **CHANGE HISTORY**

47914 First released in Issue 1. Derived from Issue 1 of the SVID. |

47915 **Issue 4**

47916 The **<unistd.h>** header is added to the SYNOPSIS.

47917 The statement indicating that *errno* is set on error in the RETURN VALUE section, and the errors
47918 [EBADF] and [ENOTTY], are marked as extensions.

47919 **Issue 5**

47920 The *ttyname_r*( ) function is included for alignment with the POSIX Threads Extension.

47921 A note indicating that the *ttyname*( ) function need not be reentrant is added to the
47922 DESCRIPTION.

47923 **Issue 6**

47924 The *ttyname_r*( ) function is marked as part of the Thread-Safe Functions option. |

47925 The following new requirements on POSIX implementations derive from alignment with the
47926 Single UNIX Specification:

47927 • The statement that *errno* is set on error is added.

47928 • The [EBADF] and [ENOTTY] optional error conditions are added.

47929 **NAME**

47930 twalk — traverse a binary search tree

47931 **SYNOPSIS**

47932 XSI
```
#include <search.h>

void twalk(const void *root,
    void (*action)(const void *, VISIT, int ));
```
47935

47936 **DESCRIPTION**

47937 Refer to *tdelete*( ).

47938 **NAME**

47939 tzname — timezone strings

47940 **SYNOPSIS**

47941 `#include <time.h>`

47942 `extern char *tzname[2];` |

47943 **DESCRIPTION** |

47944 Refer to *tzset*( ).

NAME

daylight, timezone, tzname, tzset — set timezone conversion information

SYNOPSIS

```
#include <time.h>

extern int daylight;
extern long timezone;
extern char *tzname[2];
void tzset(void);
```

DESCRIPTION

The *tzset*( ) function uses the value of the environment variable *TZ* to set time conversion information used by *ctime*( ), *localtime*( ), *mktime*( ), and *strftime*( ). If *TZ* is absent from the environment, implementation-defined default timezone information is used.

The *tzset*( ) function shall set the external variable *tzname* as follows:

```
tzname[0] = "std";
tzname[1] = "dst";
```

where *std* and *dst* are as described in the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 8, Environment Variables.

The *tzset*( ) function also shall set the external variable *daylight* to 0 if Daylight Savings Time conversions should never be applied for the timezone in use; otherwise, non-zero. The external variable *timezone* shall be set to the difference, in seconds, between Coordinated Universal Time (UTC) and local standard time.

RETURN VALUE

The *tzset*( ) function shall return no value.

ERRORS

No errors are defined.

EXAMPLES

Example TZ variables and their timezone differences are given in the table below:

TZ	*timezone*
EST	5*60*60
GMT	0*60*60
JST	–9*60*60
MET	–1*60*60
MST	7*60*60
PST	8*60*60

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

47986 **SEE ALSO**

47987 *ctime*( ), *localtime*( ), *mktime*( ), *strftime*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, |
47988 **<time.h>**

47989 **CHANGE HISTORY**

47990 First released in Issue 1. Derived from Issue 1 of the SVID. |

47991 **Issue 4**

47992 The reference to *timezone* in the SYNOPSIS section is marked as an extension.

47993 The type of *timezone* is expanded to **extern long**. |

47994 The **<time.h>** header is added to the SYNOPSIS section.

47995 The following change is incorporated for alignment with the ISO POSIX-1 standard:

47996 • The argument list is explicitly defined as **void**.

NAME

ualarm — set the interval timer

SYNOPSIS

XSI
```
#include <unistd.h>

useconds_t ualarm(useconds_t useconds, useconds_t interval);
```

DESCRIPTION

The *ualarm*( ) function shall cause the SIGALRM signal to be generated for the calling process after the number of realtime microseconds specified by the *useconds* argument has elapsed. When the *interval* argument is non-zero, repeated timeout notification occurs with a period in microseconds specified by the *interval* argument. If the notification signal, SIGALRM, is not caught or ignored, the calling process is terminated.

Implementations may place limitations on the granularity of timer values. For each interval timer, if the requested timer value requires a finer granularity than the implementation supports, the actual timer value shall be rounded up to the next supported value.

Interactions between *ualarm*( ) and any of the following are unspecified:

alarm( )
nanosleep( )
setitimer( )
timer_create( )
timer_delete( )
timer_getoverrun( )
timer_gettime( )
timer_settime( )
sleep( )

RETURN VALUE

The *ualarm*( ) function shall return the number of microseconds remaining from the previous *ualarm*( ) call. If no timeouts are pending or if *ualarm*( ) has not previously been called, *ualarm*( ) shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Applications are recommended to use *nanosleep*( ) if the Timers option is supported, or *setitimer*( ), *timer_create*( ), *timer_delete*( ), *timer_getoverrun*( ), *timer_gettime*( ), or *timer_settime*( ) instead of this function.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *nanosleep*( ), *setitimer*( ), *sleep*( ), *timer_create*( ), *timer_delete*( ), *timer_getoverrun*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>**

48041 **CHANGE HISTORY**

48042 First released in Issue 4, Version 2.

48043 **Issue 5**

48044 Moved from X/OPEN UNIX extension to BASE.

NAME

ulimit — get and set process limits

SYNOPSIS

XSI
```
#include <ulimit.h>

long ulimit(int cmd, ...);
```

DESCRIPTION

The *ulimit*( ) function provides for control over process limits. The process limits that can be controlled by this function include the maximum size of a single file that can be written (this is equivalent to using *setrlimit*( ) with RLIMIT_FSIZE). The *cmd* values, defined in **<ulimit.h>** include:

UL_GETFSIZE Return the file size limit (RLIMIT_FSIZE) of the process. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read. The return value shall be the integer part of the soft file size limit divided by 512. If the result cannot be represented as a **long**, the result is unspecified.

UL_SETFSIZE Set the file size limit for output operations of the process to the value of the second argument, taken as a **long**, multiplied by 512. If the result would overflow an **rlim_t**, the actual value set is unspecified. Any process may decrease its own limit, but only a process with appropriate privileges may increase the limit. The return value shall be the integer part of the new file size limit divided by 512.

The *ulimit*( ) function shall not change the setting of *errno* if successful.

As all return values are permissible in a successful situation, an application wishing to check for error situations should set *errno* to 0, then call *ulimit*( ), and, if it returns –1, check to see if *errno* is non-zero.

RETURN VALUE

Upon successful completion, *ulimit*( ) shall return the value of the requested limit. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *ulimit*( ) function shall fail and the limit shall be unchanged if:

[EINVAL] The *cmd* argument is not valid.

[EPERM] A process not having appropriate privileges attempts to increase its file size limit.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

48087 **SEE ALSO**

48088 *getrlimit*( ), *setrlimit*( ), *write*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<ulimit.h>**

48089 **CHANGE HISTORY**

48090 First released in Issue 1. Derived from Issue 1 of the SVID.

48091 **Issue 4**

48092 The use of **long** is replaced by **long** in the SYNOPSIS and the DESCRIPTION sections.

48093 **Issue 4, Version 2**

48094 In the DESCRIPTION, the discussion of UL_GETFSIZE and UL_SETFSIZE is revised generally to
48095 distinguish between the soft and the hard file size limit of the process. For UL_GETFSIZE, the
48096 return value is defined more precisely. For UL_SETFSIZE, the effect on both file size limits is
48097 specified, as is the effect if the result would overflow an **rlim_t**.

48098 **Issue 5**

48099 In the description of UL_SETFSIZE, the text is corrected to refer to **rlim_t** rather than the
48100 spurious **rlimit_t**.

48101 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

NAME

umask — set and get file mode creation mask

SYNOPSIS

```
#include <sys/stat.h>

mode_t umask(mode_t cmask);
```

DESCRIPTION

The *umask*( ) function shall set the process' file mode creation mask to *cmask* and return the previous value of the mask. Only the file permission bits of *cmask* (see **<sys/stat.h>**) are used; the meaning of the other bits is implementation-defined.

The process' file mode creation mask is used during *open*( ), *creat*( ), *mkdir*( ), and *mkfifo*( ) to turn off permission bits in the *mode* argument supplied. Bit positions that are set in *cmask* are cleared in the mode of the created file.

RETURN VALUE

The file permission bits in the value returned by *umask*( ) shall be the previous value of the file mode creation mask. The state of any other bits in that value is unspecified, except that a subsequent call to *umask*( ) with the returned value as *cmask* shall leave the state of the mask the same as its state before the first call, including any unspecified use of those bits.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Unsigned argument and return types for *umask*( ) were proposed. The return type and the argument were both changed to **mode_t**.

Historical implementations have made use of additional bits in *cmask* for their implementation-defined purposes. The addition of the text that the meaning of other bits of the field is implementation-defined permits these implementations to conform to this volume of IEEE Std. 1003.1-200x.

FUTURE DIRECTIONS

None.

SEE ALSO

creat( ), *mkdir*( ), *mkfifo*( ), *open*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

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

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

The RETURN VALUE section is expanded, in line with the ISO POSIX-1 standard, to describe the situation with regard to additional bits in the file mode creation mask.

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.

NAME

uname — get name of current system

SYNOPSIS

```
#include <sys/utsname.h>

int uname(struct utsname *name);
```

DESCRIPTION

The *uname*( ) function shall store information identifying the current system in the structure pointed to by *name*.

The *uname*( ) function uses the **utsname** structure defined in **<sys/utsname.h>**.

The *uname*( ) function returns a string naming the current system in the character array *sysname*. Similarly, *nodename* contains the name that the system is known by on a communications network. The arrays *release* and *version* further identify the operating system. The array *machine* contains a name that identifies the hardware that the system is running on.

The format of each member is implementation-defined.

RETURN VALUE

Upon successful completion, a non-negative value shall be returned. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The inclusion of the *nodename* member in this structure does not imply that it is sufficient information for interfacing to communications networks.

RATIONALE

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>**.

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.

The *uname*( ) function is specific to 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.

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 length of the host name is limited to 31 characters in most implementations and the host ID is a 32-bit integer.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/utsname.h>**

CHANGE HISTORY

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

Issue 4

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The DESCRIPTION is changed to indicate that the format of members in the **utsname** structure is implementation-defined.
- The RETURN VALUE section is updated to indicate that –1 is returned and *errno* set to indicate an error.

NAME

ungetc — push byte back into input stream

SYNOPSIS

```
#include <stdio.h>

int ungetc(int c, 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 *ungetc*( ) function shall push the byte specified by *c* (converted to an **unsigned char**) back onto the input stream pointed to by *stream*. The pushed-back bytes shall be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by *stream*) to a file-positioning function (*fseek*( ), *fsetpos*( ), or *rewind*( )) discards any pushed-back bytes for the stream. The external storage corresponding to the stream is unchanged.

One byte of push-back is guaranteed. If *ungetc*( ) is called too many times on the same stream without an intervening read or file-positioning operation on that stream, the operation may fail.

If the value of *c* equals that of the macro EOF, the operation fails and the input stream is unchanged.

A successful call to *ungetc*( ) shall clear the end-of-file indicator for the stream. The value of the file-position indicator for the stream after reading or discarding all pushed-back bytes shall be the same as it was before the bytes were pushed back. The file-position indicator is decremented by each successful call to *ungetc*( ); if its value was 0 before a call, its value is indeterminate after the call.

RETURN VALUE

Upon successful completion, *ungetc*( ) shall return the byte pushed back after conversion. Otherwise, it shall return EOF.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fseek( ), *getc*( ), *fsetpos*( ), *read*( ), *rewind*( ), *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 4

The DESCRIPTION is changed to make it clear that *ungetc*( ) manipulates bytes rather than (possibly multi-byte) characters.

The APPLICATION USAGE section is removed.

The following changes are incorporated for alignment with the ISO C standard:

- The *fsetpos*( ) function is added to the list of file-positioning functions in the DESCRIPTION.
- Also, this issue states that the file-position indicator is decremented by each successful call to *ungetc*( ), although note that XSI-conformant systems do not distinguish between text and binary streams. Previous issues state that the disposition of this indicator is unspecified.

NAME

ungetwc — push wide-character code back into input stream

SYNOPSIS

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

wint_t ungetwc(wint_t wc, 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 *ungetwc*( ) function shall push the character corresponding to the wide-character code specified by *wc* back onto the input stream pointed to by *stream*. The pushed-back characters shall be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by *stream*) to a file-positioning function (*fseek*( ), *fsetpos*( ), or *rewind*( )) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

One character of push-back is guaranteed. If *ungetwc*( ) is called too many times on the same stream without an intervening read or file-positioning operation on that stream, the operation may fail.

If the value of *wc* equals that of the macro WEOF, the operation fails and the input stream is unchanged.

A successful call to *ungetwc*( ) shall clear the end-of-file indicator for the stream. The value of the file-position indicator for the stream after reading or discarding all pushed-back characters shall be the same as it was before the characters were pushed back. The file-position indicator is decremented (by one or more) by each successful call to *ungetwc*( ); if its value was 0 before a call, its value is indeterminate after the call.

RETURN VALUE

Upon successful completion, *ungetwc*( ) shall return the wide-character code corresponding to the pushed-back character. Otherwise, it shall return WEOF.

ERRORS

The *ungetwc*( ) function may fail if:

[EILSEQ] An invalid character sequence is detected, or a wide-character code does not correspond to a valid character.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

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>**.

48301 **NAME**

48302 unlink — remove a directory entry

48303 **SYNOPSIS**

48304 `#include <unistd.h>`

48305 `int unlink(const char *path);`

48306 **DESCRIPTION**

48307 The *unlink*( ) function shall remove a link to a file. If *path* names a symbolic link, *unlink*( ) |
48308 removes the symbolic link named by *path* and does not affect any file or directory named by the
48309 contents of the symbolic link. Otherwise, *unlink*( ) removes the link named by the path name |
48310 pointed to by *path* and decrements the link count of the file referenced by the link.

48311 When the file's link count becomes 0 and no process has the file open, the space occupied by the
48312 file shall be freed and the file shall no longer be accessible. If one or more processes have the file
48313 open when the last link is removed, the link shall be removed before *unlink*( ) returns, but the
48314 removal of the file contents shall be postponed until all references to the file are closed.

48315 The application shall ensure that the *path* argument does not name a directory unless the process
48316 has appropriate privileges and the implementation supports using *unlink*( ) on directories.

48317 Upon successful completion, *unlink*( ) shall mark for update the *st_ctime* and *st_mtime* fields of
48318 the parent directory. Also, if the file's link count is not 0, the *st_ctime* field of the file shall be
48319 marked for update.

48320 **RETURN VALUE**

48321 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
48322 indicate the error. If –1 is returned, the named file shall not be changed.

48323 **ERRORS**

48324 The *unlink*( ) function shall fail and shall not unlink the file if:

48325 [EACCES] Search permission is denied for a component of the path prefix, or write |
48326 permission is denied on the directory containing the directory entry to be
48327 removed.

48328 [EBUSY] The file named by the *path* argument cannot be unlinked because it is being |
48329 used by the system or another process and the implementation considers this
48330 an error. |

48331 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* |
48332 argument. |

48333 [ENAMETOOLONG] |
48334 The length of the *path* argument exceeds {PATH_MAX} or a path name
48335 component is longer than {NAME_MAX}. |

48336 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string. |

48337 [ENOTDIR] A component of the path prefix is not a directory. |

48338 [EPERM] The file named by *path* is a directory, and either the calling process does not |
48339 have appropriate privileges, or the implementation prohibits using *unlink*( )
48340 on directories.

48341 XSI [EPERM] or [EACCES] |
48342 The S_ISVTX flag is set on the directory containing the file referred to by the
48343 *path* argument and the caller is not the file owner, nor is the caller the
48344 directory owner, nor does the caller have appropriate privileges.

[EROFS] The directory entry to be unlinked is part of a read-only file system.

The *unlink*( ) function may fail and not unlink the file if:

XSI [EBUSY] The file named by *path* is a named STREAM.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted path name string exceeded {PATH_MAX}.

[ETXTBSY] The entry to be unlinked is the last directory entry to a pure procedure (shared text) file that is being executed.

EXAMPLES

Removing a Link to a File

The following example shows how to remove a link to a file named **/home/cnd/mod1** by removing the entry named **/modules/pass1**.

```
#include <unistd.h>

char *path = "/modules/pass1";
int   status;
...
status = unlink(path);
```

Checking for an Error

The following example fragment creates a temporary password lock file named **LOCKFILE**, which is defined as **/etc/ptmp**, and gets a file descriptor for it. If the file cannot be opened for writing, *unlink*( ) is used to remove the link between the file descriptor and **LOCKFILE**.

```
#include <sys/types.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/stat.h>

#define LOCKFILE "/etc/ptmp"

int pfd;  /* Integer for file descriptor returned by open call. */
FILE *fpfd;  /* File pointer for use in putpwent(). */
...
/* Open password Lock file. If it exists, this is an error. */
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);
}

/* Lock file created, proceed with fdopen of lock file so that
   putpwent() can be used.
 */
if ((fpfd = fdopen(pfd, "w")) == NULL) {
```

```
        close(pfd);
        unlink(LOCKFILE);
        exit(1);
    }
```

Replacing Files

The following example fragment uses *unlink*( ) to discard links to files, so that they can be replaced with new versions of the files. The first call remove the link to **LOCKFILE** if an error occurs. Successive calls remove the links to **SAVEFILE** and **PASSWDFILE** so that new links can be created, then removes the link to **LOCKFILE** when it is no longer needed.

```
#include <sys/types.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/stat.h>

#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"
#define SAVEFILE "/etc/opasswd"
...
/* If no change was made, assume error and leave passwd unchanged. */
if (!valid_change) {
    fprintf(stderr, "Could not change password for user %s\n", user);
    unlink(LOCKFILE);
    exit(1);
}

/* Change permissions on new password file. */
chmod(LOCKFILE, S_IRUSR | S_IRGRP | S_IROTH);

/* Remove saved password file. */
unlink(SAVEFILE);

/* Save current password file. */
link(PASSWDFILE, SAVEFILE);

/* Remove current password file. */
unlink(PASSWDFILE);

/* Save new password file as current password file. */
link(LOCKFILE,PASSWDFILE);

/* Remove lock file. */
unlink(LOCKFILE);

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.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *link*( ), *remove*( ), *rmdir*( ), 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 4

The **<unistd.h>** header is added to the SYNOPSIS section.

The error [ETXTBSY] is marked as an extension.

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument *path* is changed from **char*** to **const char***.

The following change is incorporated for alignment with the FIPS requirements:

- In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path name component is larger that {NAME_MAX} is now defined as mandatory and marked as an extension.

Issue 4, Version 2

The entry is updated for X/OPEN UNIX conformance as follows:

- In the DESCRIPTION, the effect is specified if *path* specifies a symbolic link.
- In the ERRORS section, [ELOOP] is added to indicate that too many symbolic links were encountered during path name resolution
- In the ERRORS section, [EPERM] or [EACCES] are added to indicate a permission check failure when operating on directories with S_ISVTX set.
- In the ERRORS section, a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of path name resolution of a symbolic link.

Issue 5

The [EBUSY] error is added to the ''may fail'' part of the ERRORS section.

Issue 6

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

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 DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

unlockpt — unlock a pseudo-terminal master/slave pair

SYNOPSIS

XSI
```
#include <stdlib.h>

int unlockpt(int fildes);
```

DESCRIPTION

The *unlockpt*( ) function shall unlock the slave pseudo-terminal device associated with the master to which *fildes* refers.

Portable applications shall ensure that they call *unlockpt*( ) before opening the slave side of a pseudo-terminal device.

RETURN VALUE

Upon successful completion, *unlockpt*( ) shall return 0. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *unlockpt*( ) function may fail if:

[EBADF] The *fildes* argument is not a file descriptor open for writing. |

[EINVAL] The *fildes* argument is not associated with a master pseudo-terminal device. |

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

grantpt( ), *open*( ), *ptsname*( ), 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 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

48511 **NAME**

48512 unsetenv — remove environment variable

48513 **SYNOPSIS**

48514 `#include <stdlib.h>`

48515 `int unsetenv(const char *`*name*`);`

48516 **DESCRIPTION**

48517 The *unsetenv*( ) function removes an environment variable from the environment of the calling
48518 process. The *name* argument points to a string, which is the name of the variable to be removed.
48519 The named argument shall not contain an `'='` character. If the named variable does not exist in
48520 the current environment, the environment is unchanged and the function is considered to have
48521 completed successfully.

48522 If the application modifies *environ* or the pointers to which it points, the behavior of *unsetenv*( ) is
48523 undefined. The *unsetenv*( ) function shall update the list of pointers to which *environ* points.

48524 The *unsetenv*( ) function need not be reentrant. A function that is not required to be reentrant is
48525 not required to be thread-safe.

48526 **RETURN VALUE**

48527 Upon successful completion, zero shall be returned. Otherwise, –1 shall be returned, *errno* set to
48528 indicate the error, and the environment shall be unchanged.

48529 **ERRORS**

48530 The *unsetenv*( ) function shall fail if:

48531 [EINVAL] The *name* argument is a null pointer, points to an empty string, or points to a
48532 string containing an `'='` character.

48533 **EXAMPLES**

48534 None.

48535 **APPLICATION USAGE**

48536 None.

48537 **RATIONALE**

48538 Refer to the RATIONALE section in *setenv*( ).

48539 **FUTURE DIRECTIONS**

48540 None.

48541 **SEE ALSO**

48542 *getenv*( ), *setenv*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>**, |
48543 **<sys/types.h>**, **<unistd.h>**

48544 **CHANGE HISTORY**

48545 First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

48546 **NAME**

48547 usleep — suspend execution for an interval

48548 **SYNOPSIS**

48549 XSI
```
#include <unistd.h>

int usleep(useconds_t useconds);
```
48550

48551

48552 **DESCRIPTION**

48553 The *usleep*( ) function shall cause the calling thread to be suspended from execution until either
48554 the number of realtime microseconds specified by the argument *useconds* has elapsed or a signal
48555 is delivered to the calling thread and its action is to invoke a signal-catching function or to
48556 terminate the process. The suspension time may be longer than requested due to the scheduling
48557 of other activity by the system.

48558 The application shall ensure that the *useconds* argument is less than 1,000,000. If the value of
48559 *useconds* is 0, then the call has no effect.

48560 If a SIGALRM signal is generated for the calling process during execution of *usleep*( ) and if the
48561 SIGALRM signal is being ignored or blocked from delivery, it is unspecified whether *usleep*( )
48562 returns when the SIGALRM signal is scheduled. If the signal is being blocked, it is also
48563 unspecified whether it remains pending after *usleep*( ) returns or it is discarded.

48564 If a SIGALRM signal is generated for the calling process during execution of *usleep*( ), except as a
48565 result of a prior call to *alarm*( ), and if the SIGALRM signal is not being ignored or blocked from
48566 delivery, it is unspecified whether that signal has any effect other than causing *usleep*( ) to return.

48567 If a signal-catching function interrupts *usleep*( ) and examines or changes either the time a
48568 SIGALRM is scheduled to be generated, the action associated with the SIGALRM signal, or
48569 whether the SIGALRM signal is blocked from delivery, the results are unspecified.

48570 If a signal-catching function interrupts *usleep*( ) and calls *siglongjmp*( ) or *longjmp*( ) to restore an
48571 environment saved prior to the *usleep*( ) call, the action associated with the SIGALRM signal and
48572 the time at which a SIGALRM signal is scheduled to be generated are unspecified. It is also
48573 unspecified whether the SIGALRM signal is blocked, unless the process' signal mask is restored
48574 as part of the environment.

48575 Implementations may place limitations on the granularity of timer values. For each interval
48576 timer, if the requested timer value requires a finer granularity than the implementation supports,
48577 the actual timer value shall be rounded up to the next supported value.

48578 Interactions between *usleep*( ) and any of the following are unspecified:

48579 *nanosleep*( )
48580 *setitimer*( )
48581 *timer_create*( )
48582 *timer_delete*( )
48583 *timer_getoverrun*( )
48584 *timer_gettime*( )
48585 *timer_settime*( )
48586 *ualarm*( )
48587 *sleep*( )

48588 **RETURN VALUE**

48589 Upon successful completion, *usleep*( ) shall return 0; otherwise, it shall return –1 and set *errno* to
48590 indicate the error.

48591 **ERRORS**

48592 The *usleep*( ) function may fail if:

48593 [EINVAL] The time interval specified 1,000,000 or more microseconds. |

48594 **EXAMPLES**

48595 None.

48596 **APPLICATION USAGE**

48597 Applications are recommended to use *nanosleep*( ) if the Timers option is supported, or |
48598 *setitimer*( ), *timer_create*( ), *timer_delete*( ), *timer_getoverrun*( ), *timer_gettime*( ), or *timer_settime*( )
48599 instead of this function.

48600 **RATIONALE**

48601 None.

48602 **FUTURE DIRECTIONS**

48603 None.

48604 **SEE ALSO**

48605 *alarm*( ), *getitimer*( ), *nanosleep*( ), *sigaction*( ), *sleep*( ), *timer_create*( ), *timer_delete*( ),
48606 *timer_getoverrun*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>** |

48607 **CHANGE HISTORY**

48608 First released in Issue 4, Version 2.

48609 **Issue 5**

48610 Moved from X/OPEN UNIX extension to BASE.

48611 The DESCRIPTION is changed to indicate that timers are now thread-based rather than
48612 process-based.

48613 **Issue 6**

48614 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

48615 **NAME**

48616 utime — set file access and modification times

48617 **SYNOPSIS**

48618 `#include <utime.h>`

48619 `int utime(const char *`*`path`*`, const struct utimbuf *`*`times`*`);`

48620 **DESCRIPTION**

48621 The *utime*( ) function shall set the access and modification times of the file named by the *path*
48622 argument.

48623 If *times* is a null pointer, the access and modification times of the file are set to the current time.
48624 The application shall ensure that the effective user ID of the process matches the owner of the
48625 file, or the process has write permission to the file or has appropriate privileges, to use *utime*( ) in
48626 this manner.

48627 If *times* is not a null pointer, *times* is interpreted as a pointer to a **utimbuf** structure and the
48628 access and modification times are set to the values contained in the designated structure. Only a
48629 process with effective user ID equal to the user ID of the file or a process with appropriate
48630 privileges may use *utime*( ) this way.

48631 The **utimbuf** structure is defined by the header **<utime.h>**. The times in the structure **utimbuf**
48632 are measured in seconds since the Epoch.

48633 Upon successful completion, *utime*( ) shall mark the time of the last file status change, *st_ctime*,
48634 to be updated; see **<sys/stat.h>**.

48635 **RETURN VALUE**

48636 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* shall
48637 be set to indicate the error, and the file times shall not be affected.

48638 **ERRORS**

48639 The *utime*( ) function shall fail if:

48640 [EACCES] Search permission is denied by a component of the path prefix; or the *times* |
48641 argument is a null pointer and the effective user ID of the process does not
48642 match the owner of the file, the process does not have write permission for the
48643 file, and the process does not have appropriate privileges. |

48644 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* |
48645 argument. |

48646 [ENAMETOOLONG] |
48647 The length of the *path* argument exceeds {PATH_MAX} or a path name
48648 component is longer than {NAME_MAX}. |

48649 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string. |

48650 [ENOTDIR] A component of the path prefix is not a directory. |

48651 [EPERM] The *times* argument is not a null pointer and the calling process' effective user |
48652 ID does not match the owner of the file and the calling process does not have
48653 the appropriate privileges.

48654 [EROFS] The file system containing the file is read-only. |

48655 The *utime*( ) function may fail if:

48656 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
48657 resolution of the *path* argument. |

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted path name string exceeded {PATH_MAX}.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *actime* structure member must be present so that an application may set it, even though an implementation may ignore it and not change the access time on the file. If an application intends to leave one of the times of a file unchanged while changing the other, it should use *stat*( ) to retrieve the file's *st_atime* and *st_mtime* parameters, set *actime* and *modtime* in the buffer, and change one of them before making the *utime*( ) call.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<utime.h>**

CHANGE HISTORY

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

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument *path* is changed from **char*** to **const char***, and *times* is changed from **struct utimbuf*** to **const struct utimbuf***.

The following change is incorporated for alignment with the FIPS requirements:

- In the ERRORS section, the condition whereby [ENAMETOOLONG] is returned if a path name component is larger that {NAME_MAX} is now defined as mandatory and marked as an extension.

Issue 4, Version 2

The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- It states that [ELOOP] is returned if too many symbolic links are encountered during path name resolution.
- A second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of path name resolution of a symbolic link.

Issue 6

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

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 [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.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

utimes — set file access and modification times (**LEGACY**)

SYNOPSIS

XSI
```
#include <sys/time.h>

int utimes(const char *path, const struct timeval times[2]);
```

DESCRIPTION

The *utimes*( ) function shall set the access and modification times of the file pointed to by the *path* argument to the value of the *times* argument. The *utimes*( ) function allows time specifications accurate to the microsecond.

For *utimes*( ), the *times* argument is an array of **timeval** structures. The first array member represents the date and time of last access, and the second member represents the date and time of last modification. The times in the **timeval** structure are measured in seconds and microseconds since the Epoch, although rounding toward the nearest second may occur.

If the *times* argument is a null pointer, the access and modification times of the file shall be set to the current time. The application shall ensure that the effective user ID of the process is the same as the owner of the file, or has write access to the file or appropriate privileges to use this call in this manner. Upon completion, *utimes*( ) shall mark the time of the last file status change, *st_ctime*, for update.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* shall be set to indicate the error, and the file times shall not be affected.

ERRORS

The *utimes*( ) function shall fail if:

[EACCES] Search permission is denied by a component of the path prefix; or the *times* argument is a null pointer and the effective user ID of the process does not match the owner of the file and write access is denied.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

[ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a path name component is longer than {NAME_MAX}.

[ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The *times* argument is not a null pointer and the calling process' effective user ID has write access to the file but does not match the owner of the file and the calling process does not have the appropriate privileges.

[EROFS] The file system containing the file is read-only.

The *utimes*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG] Path name resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

EXAMPLES

None.

APPLICATION USAGE

For applications portability, the *utime*() function should be used to set file access and modification times instead of *utimes*().

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

The Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/time.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

This function is marked LEGACY.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENAMETOOLONG] error is restored as an error dependent on _POSIX_NO_TRUNC. This is since behavior may vary from one file system to another.

The DESCRIPTION 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.

48775 **NAME**

48776 va_arg, va_copy, va_end, va_start — handle variable argument list

48777 **SYNOPSIS**

48778 `#include <stdarg.h>`

48779 `type va_arg(va_list ap, type);`
48780 `void va_copy(va_list dest, va_list src);`
48781 `void va_end(va_list ap);`
48782 `void va_start(va_list ap, argN);`

48783 **DESCRIPTION**

48784 Refer to the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdarg.h>**.

48785 **NAME**

48786 vfork — create new process; share virtual memory

48787 **SYNOPSIS**

48788 XSI
```
#include <unistd.h>

pid_t vfork(void);
```
48789

48790

48791 **DESCRIPTION**

48792 The *vfork*( ) function has the same effect as *fork*( ), except that the behavior is undefined if the
48793 process created by *vfork*( ) either modifies any data other than a variable of type **pid_t** used to
48794 store the return value from *vfork*( ), or returns from the function in which *vfork*( ) was called, or
48795 calls any other function before successfully calling *_exit*( ) or one of the *exec* family of functions.

48796 **RETURN VALUE**

48797 Upon successful completion, *vfork*( ) shall return 0 to the child process and return the process ID
48798 of the child process to the parent process. Otherwise, –1 shall be returned to the parent, no child
48799 process shall be created, and *errno* shall be set to indicate the error.

48800 **ERRORS**

48801 The *vfork*( ) function shall fail if:

48802 [EAGAIN] The system-wide limit on the total number of processes under execution |
48803 would be exceeded, or the system-imposed limit on the total number of
48804 processes under execution by a single user would be exceeded.

48805 [ENOMEM] There is insufficient swap space for the new process. |

48806 **EXAMPLES**

48807 None.

48808 **APPLICATION USAGE**

48809 On some systems, *vfork*( ) is the same as *fork*( ).

48810 The *vfork*( ) function differs from *fork*( ) only in that the child process can share code and data
48811 with the calling process (parent process). This speeds cloning activity significantly at a risk to
48812 the integrity of the parent process if *vfork*( ) is misused.

48813 The use of *vfork*( ) for any purpose except as a prelude to an immediate call to a function from
48814 the *exec* family, or to *_exit*( ), is not advised.

48815 The *vfork*( ) function can be used to create new processes without fully copying the address
48816 space of the old process. If a forked process is simply going to call *exec*, the data space copied
48817 from the parent to the child by *fork*( ) is not used. This is particularly inefficient in a paged
48818 environment, making *vfork*( ) particularly useful. Depending upon the size of the parent's data
48819 space, *vfork*( ) can give a significant performance improvement over *fork*( ).

48820 The *vfork*( ) function can normally be used just like *fork*( ). It does not work, however, to return
48821 while running in the child's context from the caller of *vfork*( ) since the eventual return from
48822 *vfork*( ) would then return to a no longer existent stack frame. Care should be taken, also, to call
48823 *_exit*( ) rather than *exit*( ) if *exec* cannot be used, since *exit*( ) flushes and closes standard I/O
48824 channels, thereby damaging the parent process' standard I/O data structures. (Even with *fork*( ),
48825 it is wrong to call *exit*( ), since buffered data would then be flushed twice.)

48826 If signal handlers are invoked in the child process after *vfork*( ), they must follow the same rules
48827 as other code in the child process.

48828 **RATIONALE**

48829 None.

48830 **FUTURE DIRECTIONS**

48831 None.

48832 **SEE ALSO**

48833 *exec*, *exit*( ), *fork*( ), *wait*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<unistd.h>**

48834 **CHANGE HISTORY**

48835 First released in Issue 4, Version 2.

48836 **Issue 5**

48837 Moved from X/OPEN UNIX extension to BASE.

NAME

vfprintf, vprintf, vsnprintf, vsprintf — format output of a stdarg argument list

SYNOPSIS

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

int vfprintf(FILE *restrict stream, const char *restrict format,
    va_list ap);
int vprintf(const char *restrict format, va_list ap);
int vsnprintf(char *restrict s, size_t n, const char *restrict format,
    va_list ap);
int vsprintf(char *restrict s, const char *restrict format, va_list ap);
```

DESCRIPTION

CX For *vfprintf*( ), *vprintf*( ), and *vsprintf*( ): 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.

XSI The *vprintf*( ), *vfprintf*( ), *vsnprintf*( ), and *vsprintf*( ) functions shall be the same as *printf*( ), XSI *fprintf*( ), *snprintf*( ), and *sprintf*( ) respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by **<stdarg.h>**.

These functions do not invoke the *va_end* macro. As these functions invoke the *va_arg* macro, the value of *ap* after the return is indeterminate.

RETURN VALUE

Refer to *fprintf*( ).

ERRORS

Refer to *fprintf*( ).

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions should call *va_end*(*ap*) afterwards to clean up.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fprintf( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdarg.h>**, **<stdio.h>**

CHANGE HISTORY

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

Issue 4

The APPLICATION USAGE section is added.

The FUTURE DIRECTIONS section is removed.

The following changes are incorporated for alignment with the ISO C standard:

- These functions are no longer marked as extensions.

- The type of argument *format* is changed from **char*** to **const char***.
- Reference to the **<varargs.h>** header in the DESCRIPTION is replaced by **<stdarg.h>**. The last paragraph has also been added to indicate interactions with the *va_arg* and *va_end* macros.

Issue 5

The *vsnprintf*( ) function is added.

Issue 6

The *vfprintf*( ), *vprintf*( ), *vsnprintf*( ), and *vsprintf*( ) functions are updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

vfscanf, vscanf, vsscanf — format input of a stdarg list

SYNOPSIS

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

int vfscanf(FILE *restrict stream, const char *restrict format,
    va_list arg);
int vscanf(const char *restrict format, va_list arg);
int vsscanf(const char *restrict s, const char *restrict format,
    va_list arg);
```

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 be equivalent to the *scanf*( ), *fscanf*( ), and *sscanf*( ) functions, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by the **<stdarg.h>** header. These functions do not invoke the *va_end* macro. As these functions invoke the *va_arg* macro, the value of *ap* after the return is indeterminate.

RETURN VALUE

Refer to *fscanf*( ).

ERRORS

Refer to *fscanf*( ).

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions should call *va_end*(*ap*) afterwards to clean up.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fscanf( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdarg.h>**, **<stdio.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

vfwprintf, vswprintf, vwprintf — wide-character formatted output of a stdarg argument list

SYNOPSIS

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

int vfwprintf(FILE *restrict stream, const wchar_t *restrict format,
    va_list arg);
int vswprintf(wchar_t *restrict ws, size_t n, const wchar_t *restrict format,
    va_list arg);
int vwprintf(const wchar_t *restrict format, va_list arg);
```

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 *vfwprintf*( ), *vswprintf*( ), and *vwprintf*( ) functions shall be the same as *fwprintf*( ), *swprintf*( ), and *wprintf*( ) respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by **<stdarg.h>**.

These functions do not invoke the *va_end* macro. However, as these functions do invoke the *va_arg* macro, the value of *ap* after the return is indeterminate.

RETURN VALUE

Refer to *fwprintf*( ).

ERRORS

Refer to *fwprintf*( ).

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions should call *va_end*(*ap*) afterwards to clean up.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fwprintf( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdarg.h>**, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The *vfwprintf*( ), *vswprintf*( ), and *vwprintf*( ) prototypes are updated for alignment with the ISO/IEC 9899:1999 standard. ( )

NAME

vfwscanf, vswscanf, vwscanf — wide-character formattted input of a stdarg list

SYNOPSIS

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

int vfwscanf(FILE *restrict stream, const wchar_t *restrict format,
    va_list arg);
int vswscanf(const wchar_t *restrict ws, const wchar_t *restrict format,
    va_list arg);
int vwscanf(const wchar_t *restrict format, va_list arg);
```

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 be equivalent to the *fwscanf*( ), *swscanf*( ), and *wscanf*( ) functions, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by the **<stdarg.h>** header. These functions do not invoke the *va_end* macro. As these functions invoke the *va_arg* macro, the value of *ap* after the return is indeterminate.

RETURN VALUE

Refer to *fwscanf*( ).

ERRORS

Refer to *fwscanf*( ).

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions should call *va_end*(*ap*) afterwards to clean up.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fwscanf( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdarg.h>**, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

49002 **NAME**

49003 vprintf, vsnprintf, vsprintf — format output of a stdarg argument list

49004 **SYNOPSIS**

49005 `#include <stdarg.h>`
49006 `#include <stdio.h>`

49007 `int vprintf(const char *format, va_list ap);`
49008 XSI `int vsnprintf(char *s, size_t n, const char *format, va_list ap);`
49009 `int vsprintf(char *s, const char *format, va_list ap);`

49010 **DESCRIPTION**

49011 Refer to *vfprintf*( ).

49012 **NAME**
49013 vscanf, vsscanf — format input of a stdarg list

49014 **SYNOPSIS**
49015 `#include <stdarg.h>`
49016 `#include <stdio.h>`

49017 `int vscanf(const char *restrict format, va_list arg);`
49018 `int vsscanf(const char *restrict s, const char *restrict format,`
49019 `va_list arg);`

49020 **DESCRIPTION**
49021 Refer to *vfscanf*( ).

49022 **NAME**

49023 vswprintf, vwprintf — wide-character formatted output of a stdarg argument list

49024 **SYNOPSIS**

49025 ```
#include <stdarg.h>
```
49026 ```
#include <stdio.h>
```
49027 ```
#include <wchar.h>
```

49028 ```
int vswprintf(wchar_t *ws, size_t n, const wchar_t *format,
```
49029 ```
    va_list arg);
```
49030 ```
int vwprintf(const wchar_t *format, va_list arg);
```

49031 **DESCRIPTION**

49032 Refer to *vfwprintf*( ).

49033 **NAME**

49034 vswscanf, vwscanf — wide-character formattted input of a stdarg list

49035 **SYNOPSIS**

49036 `#include <stdarg.h>`

49037 `#include <stdio.h>`

49038 `#include <wchar.h>`

49039 `int vswscanf(const wchar_t *restrict` *`ws`*`, const wchar_t *restrict` *`format`*`,`

49040 `va_list` *`arg`*`);`

49041 `int vwscanf(const wchar_t *restrict` *`format`*`, va_list` *`arg`*`);`

49042 **DESCRIPTION**

49043 Refer to *vfwscanf*( ).

NAME

wait, waitpid — wait for a child process to stop or terminate

SYNOPSIS

```
#include <sys/wait.h>

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

DESCRIPTION

The *wait*( ) and *waitpid*( ) functions allow the calling process to obtain status information pertaining to one of its child processes. Various options permit status information to be obtained for child processes that have terminated or stopped. If status information is available for two or more child processes, the order in which their status is reported is unspecified.

The *wait*( ) function shall suspend execution of the calling thread until status information for one of the terminated child processes of the calling process is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. If more than one thread is suspended in *wait*( ) or *waitpid*( ) awaiting termination of the same process, exactly one thread shall return the process status at the time of the target process termination. If status information is available prior to the call to *wait*( ), return shall be immediate.

The *waitpid*( ) function shall behave identically to *wait*( ) if the *pid* argument is (**pid_t**)–1 and the *options* argument is 0. Otherwise, its behavior shall be modified by the values of the *pid* and *options* arguments.

The *pid* argument specifies a set of child processes for which *status* is requested. The *waitpid*( ) function shall only return the status of a child process from this set:

- If *pid* is equal to (**pid_t**)–1, *status* is requested for any child process. In this respect, *waitpid*( ) is then equivalent to *wait*( ).
- If *pid* is greater than 0, it specifies the process ID of a single child process for which *status* is requested.
- If *pid* is 0, *status* is requested for any child process whose process group ID is equal to that of the calling process.
- If *pid* is less than (**pid_t**)–1, *status* is requested for any child process whose process group ID is equal to the absolute value of *pid*.

The *options* argument is constructed from the bitwise-inclusive OR of zero or more of the following flags, defined in the header **<sys/wait.h>**:

XSI

WCONTINUED The *waitpid*( ) function shall report the status of any continued child process specified by *pid* whose status has not been reported since it continued from a job control stop.

WNOHANG The *waitpid*( ) function shall not suspend execution of the calling thread if *status* is not immediately available for one of the child processes specified by *pid*.

WUNTRACED The status of any child processes specified by *pid* that are stopped, and whose status has not yet been reported since they stopped, shall also be reported to the requesting process.

XSI

If the calling process has SA_NOCLDWAIT set or has SIGCHLD set to SIG_IGN, and the process has no unwaited-for children that were transformed into zombie processes, the calling thread shall block until all of the children of the process containing the calling thread terminate, and *wait*( ) and *waitpid*( ) shall fail and set *errno* to [ECHILD].

If *wait*( ) or *waitpid*( ) 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 *stat_loc* is not a null pointer, information shall be stored in the location pointed to by *stat_loc*. The value stored at the location pointed to by *stat_loc* shall be 0 if and only if the status returned is from a terminated child process that terminated by one of the following means:

1. The process returned 0 from *main*( ).
2. The process called *_exit*( ) or *exit*( ) with a *status* argument of 0.
3. The process was terminated because the last thread in the process terminated.

Regardless of its value, this information may be interpreted using the following macros, which are defined in **<sys/wait.h>** and evaluate to integral expressions; the *stat_val* argument is the integer value pointed to by *stat_loc*.

WIFEXITED(*stat_val*)
: Evaluates to a non-zero value if *status* was returned for a child process that terminated normally.

WEXITSTATUS(*stat_val*)
: If the value of WIFEXITED(*stat_val*) is non-zero, this macro evaluates to the low-order 8 bits of the *status* argument that the child process passed to *_exit*( ) or *exit*( ), or the value the child process returned from *main*( ).

WIFSIGNALED(*stat_val*)
: Evaluates to non-zero value if status was returned for a child process that terminated due to the receipt of a signal that was not caught (see **<signal.h>**).

WTERMSIG(*stat_val*)
: If the value of WIFSIGNALED(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFSTOPPED(*stat_val*)
: Evaluates to a non-zero value if status was returned for a child process that is currently stopped.

WSTOPSIG(*stat_val*)
: If the value of WIFSTOPPED(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

XSI WIFCONTINUED(*stat_val*)
: Evaluates to a non-zero value if status was returned for a child process that has continued from a job control stop.

SPN It is unspecified whether the *status* value returned by calls to *wait*( ) or *waitpid*( ) for processes created by *posix_spawn*( ) or *posix_spawnp*( ) may indicate a WIFSTOPPED(*stat_val*) before subsequent calls to *wait*( ) or *waitpid*( ) indicate WIFEXITED(*stat_val*) as the result of an error detected before the new process image starts executing.

It is unspecified whether the *status* value returned by calls to *wait*( ) or *waitpid*( ) for processes created by *posix_spawn*( ) or *posix_spawnp*( ) may indicate a WIFSIGNALED(*stat_val*) if a signal is sent to the parent's process group after *posix_spawn*( ) or *posix_spawnp*( ) is called.

If the information pointed to by *stat_loc* was stored by a call to *waitpid*( ) that specified the XSI WUNTRACED flag and did not specify the WCONTINUED flag, exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), and WIFSTOPPED(**stat_loc*) shall evaluate to a non-zero value.

If the information pointed to by *stat_loc* was stored by a call to *waitpid*() that specified the WUNTRACED and WCONTINUED flags, exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), WIFSTOPPED(**stat_loc*), and WIFCONTINUED(**stat_loc*) shall evaluate to a non-zero value.

If the information pointed to by *stat_loc* was stored by a call to *waitpid*() that did not specify the WUNTRACED or WCONTINUED flags, or by a call to the *wait*() function, exactly one of the macros WIFEXITED(**stat_loc*) and WIFSIGNALED(**stat_loc*) shall evaluate to a non-zero value.

If the information pointed to by *stat_loc* was stored by a call to *waitpid*() that did not specify the WUNTRACED flag and specified the WCONTINUED flag, or by a call to the *wait*() function, exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), and WIFCONTINUED(**stat_loc*) shall evaluate to a non-zero value.

If _POSIX_REALTIME_SIGNALS is defined, and the implementation queues the SIGCHLD signal, then if *wait*() or *waitpid*() returns because the status of a child process is available, any pending SIGCHLD signal associated with the process ID of the child process shall be discarded. Any other pending SIGCHLD signals shall remain pending.

Otherwise, if SIGCHLD is blocked, if *wait*() or *waitpid*() return because the status of a child process is available, any pending SIGCHLD signal shall be cleared unless the status of another child process is available.

For all other conditions, it is unspecified whether child *status* will be available when a SIGCHLD signal is delivered.

There may be additional implementation-defined circumstances under which *wait*() or *waitpid*() report *status*. This shall not occur unless the calling process or one of its child processes explicitly makes use of a non-standard extension. In these cases the interpretation of the reported *status* is implementation-defined.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes shall be assigned a new parent process ID corresponding to an implementation-defined system process.

RETURN VALUE

If *wait*() or *waitpid*() returns because the status of a child process is available, these functions shall return a value equal to the process ID of the child process for which *status* is reported. If *wait*() or *waitpid*() returns due to the delivery of a signal to the calling process, –1 shall be returned and *errno* set to [EINTR]. If *waitpid*() was invoked with WNOHANG set in *options*, it has at least one child process specified by *pid* for which *status* is not available, and *status* is not available for any process specified by *pid*, 0 is returned. Otherwise, (**pid_t**)–1 shall be returned, and *errno* set to indicate the error.

ERRORS

The *wait*() function shall fail if:

[ECHILD] The calling process has no existing unwaited-for child processes.

[EINTR] The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

The *waitpid*() function shall fail if:

[ECHILD] The process specified by *pid* does not exist or is not a child of the calling process, or the process group specified by *pid* does not exist or does not have any member process that is a child of the calling process.

[EINTR] The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

[EINVAL] The *options* argument is not valid.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

A call to the *wait*( ) or *waitpid*( ) function only returns *status* on an immediate child process of the calling process; that is, a child that was produced by a single *fork*( ) call (perhaps followed by an *exec* or other function calls) from the parent. If a child produces grandchildren by further use of *fork*( ), none of those grandchildren nor any of their descendants affect the behavior of a *wait*( ) from the original parent process. Nothing in this volume of IEEE Std. 1003.1-200x prevents an implementation from providing extensions that permit a process to get *status* from a grandchild or any other process, but a process that does not use such extensions must be guaranteed to see *status* from only its direct children.

The *waitpid*( ) function is provided for three reasons:

1. To support job control
2. To permit a non-blocking version of the *wait*( ) function
3. To permit a library routine, such as *system*( ) or *pclose*( ), to wait for its children without interfering with other terminated children for which the process has not waited

The first two of these facilities are based on the *wait3*( ) function provided by 4.3 BSD. The function uses the *options* argument, which is identical to an argument to *wait3*( ). The WUNTRACED flag is used only in conjunction with job control on systems supporting job control. Its name comes from 4.3 BSD and refers to the fact that there are two types of stopped processes in that implementation: processes being traced via the *ptrace*( ) debugging facility and (untraced) processes stopped by job control signals. Since *ptrace*( ) is not part of this volume of IEEE Std. 1003.1-200x, only the second type is relevant. The name WUNTRACED was retained because its usage is the same, even though the name is not intuitively meaningful in this context.

The third reason for the *waitpid*( ) function is to permit independent sections of a process to spawn and wait for children without interfering with each other. For example, the following problem occurs in developing a portable shell, or command interpreter:

```
stream = popen("/bin/true");
(void) system("sleep 100");
(void) pclose(stream);
```

On all historical implementations, the final *pclose*( ) fails to reap the *wait*( ) *status* of the *popen*( ).

The status values are retrieved by macros, rather than given as specific bit encodings as they are in most historical implementations (and thus expected by existing programs). This was necessary to eliminate a limitation on the number of signals an implementation can support that was inherent in the traditional encodings. This volume of IEEE Std. 1003.1-200x does require that a *status* value of zero corresponds to a process calling *_exit*(0), as this is the most common encoding expected by existing programs. Some of the macro names were adopted from 4.3 BSD.

These macros syntactically operate on an arbitrary integer value. The behavior is undefined unless that value is one stored by a successful call to *wait*( ) or *waitpid*( ) in the location pointed to by the *stat_loc* argument. An early proposal attempted to make this clearer by specifying each

argument as **stat_loc* rather than *stat_val*. However, that did not follow the conventions of other specifications in this volume of IEEE Std. 1003.1-200x or traditional usage. It also could have implied that the argument to the macro must literally be **stat_loc*; in fact, that value can be stored or passed as an argument to other functions before being interpreted by these macros.

The extension that affects *wait*( ) and *waitpid*( ) and is common in historical implementations is the *ptrace*( ) function. It is called by a child process and causes that child to stop and return a *status* that appears identical to the *status* indicated by WIFSTOPPED. The *status* of *ptrace*( ) children is traditionally returned regardless of the WUNTRACED flag (or by the *wait*( ) function). Most applications do not need to concern themselves with such extensions because they have control over what extensions they or their children use. However, applications, such as command interpreters, that invoke arbitrary processes may see this behavior when those arbitrary processes misuse such extensions.

Implementations that support *core* file creation or other implementation-defined actions on termination of some processes traditionally provide a bit in the *status* returned by *wait*( ) to indicate that such actions have occurred.

Allowing the *wait*( ) family of functions to discard a pending SIGCHLD signal that is associated with a successfully waited-for child process puts them into the *sigwait*( ) and *sigwaitinfo*( ) category with respect to SIGCHLD.

This definition allows implementations to treat a pending SIGCHLD signal as accepted by the process in *wait*( ), with the same meaning of ‘‘accepted’’ as when that word is applied to the *sigwait*( ) family of functions.

Allowing the *wait*( ) family of functions to behave this way permits an implementation to be able to deal precisely with SIGCHLD signals.

In particular, an implementation that does accept (discard) the SIGCHLD signal can make the following guarantees regardless of the queuing depth of signals in general (the list of waitable children can hold the SIGCHLD queue):

1. If a SIGCHLD signal handler is established via *sigaction*( ) without the SA_RESETHAND flag, SIGCHLD signals can be accurately counted; that is, exactly one SIGCHLD signal will be delivered to or accepted by the process for every child process that terminates.
2. A single *wait*( ) issued from a SIGCHLD signal handler can be guaranteed to return immediately with status information for a child process.
3. When SA_SIGINFO is requested, the SIGCHLD signal handler can be guaranteed to receive a non-NULL pointer to a **siginfo_t** structure that describes a child process for which a wait via *waitpid*( ) or *waitid*( ) will not block or fail.
4. The *system*( ) function will not cause a processs SIGCHLD handler to be called as a result of the *fork*( )/*exec* executed within *system*( ) because *system*( ) will accept the SIGCHLD signal when it performs a *waitpid*( ) for its child process. This is a desirable behavior of *system*( ) so that it can be used in a library without causing side effects to the application linked with the library.

An implementation that does not permit the *wait*( ) family of functions to accept (discard) a pending SIGCHLD signal associated with a successfully waited-for child, cannot make the guarantees described above for the following reasons:

Guarantee #1

Although it might be assumed that reliable queuing of all SIGCHLD signals generated by the system can make this guarantee, the counter example is the case of a process that blocks SIGCHLD and performs an indefinite loop of *fork*( )/*wait*( ) operations. If the

implementation supports queued signals, then eventually the system will run out of memory for the queue. The guarantee cannot be made because there must be some limit to the depth of queuing.

Guarantees #2 and #3

These cannot be guaranteed unless the *wait*( ) family of functions accepts the SIGCHLD signal. Otherwise, a *fork*( )/*wait*( ) executed while SIGCHLD is blocked (as in the *system*( ) function) will result in an invocation of the handler when SIGCHLD is unblocked, after the process has disappeared.

Guarantee #4

Although possible to make this guarantee, *system*( ) would have to set the SIGCHLD handler to SIG_DFL so that the SIGCHLD signal generated by its *fork*( ) would be discarded (the SIGCHLD default action is to be ignored), then restore it to its previous setting. This would have the undesirable side effect of discarding all SIGCHLD signals pending to the process.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *exit*( ), *fork*( ), *waitid*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/types.h>**, **<sys/wait.h>**

CHANGE HISTORY

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

Issue 4

The **<sys/types.h>** header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

Error return values throughout the DESCRIPTION and RETURN VALUE sections are changed to show the proper casting (that is, (**pid_t**)–1).

The words ‘‘If the implementation supports job control’’ are removed from the description of WUNTRACED. This is because job control is defined as mandatory for Issue 4 conforming implementations.

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- Text describing conditions under which 0 is returned when WNOHANG is set in *options* is added to the RETURN VALUE section.

Issue 4, Version 2

The *waitpid*( ) function is added.

The following changes are incorporated in the DESCRIPTION for X/OPEN UNIX conformance:

- The WCONTINUED *options* flag and the WIFCONTINUED(*stat_val*) macro are added.
- Text following the list of *options* flags explains the implications of setting the SA_NOCLDWAIT signal flag, or setting SIGCHLD to SIG_IGN.
- Text following the list of macros, which explains what macros return non-zero values in certain cases, is expanded and the value of the WCONTINUED flag on the previous call to *waitpid*( ) is taken into account.

Issue 5

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

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.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The processing of the SIGCHLD signal and the [ECHILD] error is clarified.

The semantics of WIFSTOPPED(*stat_val*), WIFEXITED(*stat_val*), and WIFSIGNALED(*stat_val*) are defined with respect to *posix_spawn*() or *posix_spawnp*() for alignment with IEEE Std. 1003.1d-1999.

The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

waitid — wait for a child process to change state

SYNOPSIS

XSI

```
#include <sys/wait.h>

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

DESCRIPTION

The *waitid*( ) function shall suspend the calling thread until one child of the process containing the calling thread changes state. It records the current state of a child in the structure pointed to by *infop*. If a child process changed state prior to the call to *waitid*( ), *waitid*( ) returns immediately. If more than one thread is suspended in *wait*( ) or *waitpid*( ) waiting termination of the same process, exactly one thread returns the process status at the time of the target process termination

The *idtype* and *id* arguments are used to specify which children *waitid*( ) waits for.

If *idtype* is P_PID, *waitid*( ) shall wait for the child with a process ID equal to (**pid_t**)*id*.

If *idtype* is P_PGID, *waitid*( ) shall wait for any child with a process group ID equal to (**pid_t**)*id*.

If *idtype* is P_ALL, *waitid*( ) shall wait for any children and *id* is ignored.

The *options* argument is used to specify which state changes *waitid*( ) shall wait for. It is formed by OR'ing together one or more of the following flags:

WEXITED	Wait for processes that have exited.
WSTOPPED	Status shall be returned for any child that has stopped upon receipt of a signal.
WCONTINUED	Status shall be returned for any child that was stopped and has been continued.
WNOHANG	Return immediately if there are no children to wait for.
WNOWAIT	Keep the process whose status is returned in *infop* in a waitable state. This shall not affect the state of the process; the process may be waited for again after this call completes.

The application shall ensure that the *infop* argument points to a **siginfo_t** structure. If *waitid*( ) returns because a child process was found that satisfied the conditions indicated by the arguments *idtype* and *options*, then the structure pointed to by *infop* shall be filled in by the system with the status of the process. The *si_signo* member shall always be equal to SIGCHLD.

RETURN VALUE

Notes to Reviewers

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

D1, XSH, ERN 416 points out an omission. The following text is proposed: ''If WNOHANG was specified and there are no children to wait for, 0 shall be returned.''

If *waitid*( ) returns due to the change of state of one of its children, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *waitid*( ) function shall fail if:

[ECHILD]	The calling process has no existing unwaited-for child processes.

[EINTR] The *waitid*( ) function was interrupted by a signal. |

[EINVAL] An invalid value was specified for *options*, or *idtype* and *id* specify an invalid set of processes. |

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *exit*( ), *wait*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<sys/wait.h>** |

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

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

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

49385 **NAME**

49386 waitpid — wait for a child process to stop or terminate

49387 **SYNOPSIS**

49388 `#include <sys/wait.h>`

49389 `pid_t waitpid(pid_t` *`pid`*`, int *`*`stat_loc`*`, int` *`options`*`);`

49390 **DESCRIPTION**

49391 Refer to *wait*( ).

NAME

wcrtomb — convert a wide-character code to a character (restartable)

SYNOPSIS

```
#include <stdio.h>

size_t wcrtomb(char *restrict s, wchar_t wc, mbstate_t *restrict ps);
```

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.

If *s* is a null pointer, the *wcrtomb*( ) function shall be equivalent to the call:

```
wcrtomb(buf, L'\0', ps)
```

where *buf* is an internal buffer.

If *s* is not a null pointer, the *wcrtomb*( ) function shall determine the number of bytes needed to represent the character that corresponds to the wide character given by *wc* (including any shift sequences), and stores the resulting bytes in the array whose first element is pointed to by *s*. At most {MB_CUR_MAX} bytes are stored. If *wc* is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state. The resulting state described is the initial conversion state.

If *ps* is a null pointer, the *wcrtomb*( ) function uses its own internal **mbstate_t** object, which is initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by *ps* is used to completely describe the current conversion state of the associated character sequence. The implementation shall behave as if no function defined in this volume of IEEE Std. 1003.1-200x calls *wcrtomb*( ).

CX If the application uses any of the _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS functions, the application shall ensure that the *wcrtomb*( ) function is called with a non-NULL *ps* argument.

XSI The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

RETURN VALUE

The *wcrtomb*( ) function shall return the number of bytes stored in the array object (including any shift sequences). When *wc* is not a valid wide character, an encoding error shall occur. In this case, the function shall store the value of the macros [EILSEQ] in *errno* and shall return (**size_t**)–1; the conversion state shall be undefined.

ERRORS

The *wcrtomb*( ) function may fail if:

CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

[EILSEQ] Invalid wide-character code is detected.

49428 **EXAMPLES**

49429 None.

49430 **APPLICATION USAGE**

49431 None.

49432 **RATIONALE**

49433 None.

49434 **FUTURE DIRECTIONS**

49435 None.

49436 **SEE ALSO**

49437 *mbsinit*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>** |

49438 **CHANGE HISTORY**

49439 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
49440 (E).

49441 **Issue 6**

49442 In the DESCRIPTION, a note on using this function in a threaded application is added.

49443 Extensions beyond the ISO C standard are now marked.

49444 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements. |

49445 The *wcrtomb*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard. |

NAME

wcscat — concatenate two wide-character strings

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcscat(wchar_t *restrict ws1, const wchar_t *restrict ws2);
```

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 *wcscat*( ) function shall append a copy of the wide-character string pointed to by *ws2* (including the terminating null wide-character code) to the end of the wide-character string pointed to by *ws1*. The initial wide-character code of *ws2* overwrites the null wide-character code at the end of *ws1*. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *wcscat*( ) function shall return *ws1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcsncat( ), 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 6

The Open Group corrigenda item U040/2 has been applied. In the RETURN VALUE section, *s1* is changed to *ws1*.

The *wcscat*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wcschr — wide-character string scanning operation

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcschr(const wchar_t *ws, wchar_t wc);
```

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 *wcschr*( ) function shall locate the first occurrence of *wc* in the wide-character string pointed to by *ws*. The application shall ensure that the value of *wc* is a character representable as a type **wchar_t** and a wide-character code corresponding to a valid character in the current locale. The terminating null wide-character code is considered to be part of the wide-character string.

RETURN VALUE

Upon completion, *wcschr*( ) shall return a pointer to the wide-character code, or a null pointer if the wide-character code is not found.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcsrchr( ), 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 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

wcscmp — compare two wide-character strings

SYNOPSIS

```
#include <wchar.h>

int wcscmp(const wchar_t *ws1, const wchar_t *ws2);
```

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 *wcscmp*( ) function shall compare the wide-character string pointed to by *ws1* to the wide-character string pointed to by *ws2*.

The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of wide-character codes that differ in the objects being compared.

RETURN VALUE

Upon completion, *wcscmp*( ) shall return an integer greater than, equal to, or less than 0, if the wide-character string pointed to by *ws1* is greater than, equal to, or less than the wide-character string pointed to by *ws2*, respectively.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcsncmp( ), 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.

NAME

wcscoll — wide-character string comparison using collating information

SYNOPSIS

```
#include <wchar.h>

int wcscoll(const wchar_t *ws1, const wchar_t *ws2);
```

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 *wcscoll*( ) function shall compare the wide-character string pointed to by *ws1* to the wide-character string pointed to by *ws2*, both interpreted as appropriate to the *LC_COLLATE* category of the current locale.

CX The *wcscoll*( ) 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 *wcscoll*( ). If *errno* is non-zero on return, an error has occurred.

RETURN VALUE

Upon successful completion, *wcscoll*( ) shall return an integer greater than, equal to, or less than 0, according to whether the wide-character string pointed to by *ws1* is greater than, equal to, or less than the wide-character string pointed to by *ws2*, when both are interpreted as appropriate CX to the current locale. On error, *wcscoll*( ) may set *errno*, but no return value is reserved to indicate an error.

ERRORS

The *wcscoll*( ) function may fail if:

[EINVAL] The *ws1* or *ws2* arguments contain wide-character codes outside the domain of the collating sequence. |

EXAMPLES

None.

APPLICATION USAGE

The *wcsxfrm*( ) and *wcscmp*( ) functions should be used for sorting large lists.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcscmp( ), *wcsxfrm*( ), 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.

NAME

wcscpy — copy a wide-character string

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcscpy(wchar_t *restrict ws1, const wchar_t *restrict ws2);
```

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 *wcscpy*( ) function shall copy the wide-character string pointed to by *ws2* (including the terminating null wide-character code) into the array pointed to by *ws1*. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *wcscpy*( ) function shall return *ws1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcsncpy( ), 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 6

The *wcscpy*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

wcscspn — get length of a complementary wide substring

SYNOPSIS

```
#include <wchar.h>

size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);
```

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 *wcscspn*( ) function shall compute the length of the maximum initial segment of the wide-character string pointed to by *ws1* which consists entirely of wide-character codes *not* from the wide-character string pointed to by *ws2*.

RETURN VALUE

The *wcscspn*( ) function shall return the length of the initial substring of *ws1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcsspn( ), 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

The RETURN VALUE section is updated to indicate that *wcscspn*( ) returns the length of *ws1*, rather than *ws1* itself.

NAME

wcsftime — convert date and time to a wide-character string

SYNOPSIS

```
#include <wchar.h>

size_t wcsftime(wchar_t *restrict wcs, size_t maxsize,
    const wchar_t *restrict format, const struct tm *restrict timptr);
```

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 *wcsftime*( ) function shall be equivalent to the *strftime*( ) function, except that:

- The argument *wcs* points to the initial element of an array of wide characters into which the generated output is to be placed.
- The argument *maxsize* indicates the maximum number of wide characters to be placed in the output array.
- The argument *format* is a wide-character string and the conversion specifications are replaced by corresponding sequences of wide characters.
- The return value indicates the number of wide characters placed in the output array.

If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

If the total number of resulting wide-character codes including the terminating null wide-character code is no more than *maxsize*, *wcsftime*( ) shall return the number of wide-character codes placed into the array pointed to by *wcs*, not including the terminating null wide-character code.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strftime( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 5

Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of the *format* argument is changed from **const char*** to **const wchar_t***.

Issue 6

The *wcsftime*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

wcslen — get wide-character string length

SYNOPSIS

```
#include <wchar.h>

size_t wcslen(const wchar_t *ws);
```

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 *wcslen*( ) function shall compute the number of wide-character codes in the wide-character string to which *ws* points, not including the terminating null wide-character code.

RETURN VALUE

The *wcslen*( ) function shall return the length of *ws*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

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. Derived from the MSE working draft.

NAME

wcsncat — concatenate a wide-character string with part of another

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcsncat(wchar_t *restrict ws1, const wchar_t *restrict 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 *wcsncat*( ) function shall append not more than *n* wide-character codes (a null wide-character code and wide-character codes that follow it are not appended) from the array pointed to by *ws2* to the end of the wide-character string pointed to by *ws1*. The initial wide-character code of *ws2* overwrites the null wide-character code at the end of *ws1*. A terminating null wide-character code is always appended to the result. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *wcsncat*( ) function shall return *ws1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcscat( ), 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 6

The *wcsncat*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

wcsncmp — compare part of two wide-character strings

SYNOPSIS

```
#include <wchar.h>

int wcsncmp(const 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 *wcsncmp*( ) function shall compare not more than *n* wide-character codes (wide-character codes that follow a null wide-character code are not compared) from the array pointed to by *ws1* to the array pointed to by *ws2*.

The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of wide-character codes that differ in the objects being compared.

RETURN VALUE

Upon successful completion, *wcsncmp*( ) shall return an integer greater than, equal to, or less than 0, if the possibly null-terminated array pointed to by *ws1* is greater than, equal to, or less than the possibly null-terminated array pointed to by *ws2*, respectively.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcscmp( ), 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.

49783 **NAME**

49784 wcsncpy — copy part of a wide-character string

49785 **SYNOPSIS**

49786 `#include <wchar.h>`

49787 `wchar_t *wcsncpy(wchar_t *restrict ws1, const wchar_t *restrict ws2,`
49788 `size_t n);`

49789 **DESCRIPTION**

49790 CX The functionality described on this reference page is aligned with the ISO C standard. Any
49791 conflict between the requirements described here and the ISO C standard is unintentional. This
49792 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

49793 The *wcsncpy*( ) function shall copy not more than *n* wide-character codes (wide-character codes
49794 that follow a null wide-character code are not copied) from the array pointed to by *ws2* to the
49795 array pointed to by *ws1*. If copying takes place between objects that overlap, the behavior is
49796 undefined.

49797 If the array pointed to by *ws2* is a wide-character string that is shorter than *n* wide-character
49798 codes, null wide-character codes are appended to the copy in the array pointed to by *ws1*, until *n*
49799 wide-character codes in all are written.

49800 **RETURN VALUE**

49801 The *wcsncpy*( ) function shall return *ws1*; no return value is reserved to indicate an error.

49802 **ERRORS**

49803 No errors are defined.

49804 **EXAMPLES**

49805 None.

49806 **APPLICATION USAGE**

49807 If there is no null wide-character code in the first *n* wide-character codes of the array pointed to
49808 by *ws2*, the result is not null-terminated.

49809 **RATIONALE**

49810 None.

49811 **FUTURE DIRECTIONS**

49812 None.

49813 **SEE ALSO**

49814 *wcscpy*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>**

49815 **CHANGE HISTORY**

49816 First released in Issue 4. Derived from the MSE working draft.

49817 **Issue 6**

49818 The *wcsncpy*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

wcspbrk — scan wide-character string for a wide-character code

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);
```

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 *wcspbrk*( ) function shall locate the first occurrence in the wide-character string pointed to by *ws1* of any wide-character code from the wide-character string pointed to by *ws2*.

RETURN VALUE

Upon successful completion, *wcspbrk*( ) shall return a pointer to the wide-character code or a null pointer if no wide-character code from *ws2* occurs in *ws1*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcschr( ), *wcsrchr*( ), 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.

NAME

wcsrchr — wide-character string scanning operation

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcsrchr(const wchar_t *ws, wchar_t wc);
```

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 *wcsrchr*( ) function shall locate the last occurrence of *wc* in the wide-character string pointed to by *ws*. The application shall ensure that the value of *wc* is a character representable as a type **wchar_t** and a wide-character code corresponding to a valid character in the current locale. The terminating null wide-character code is considered to be part of the wide-character string.

RETURN VALUE

Upon successful completion, *wcsrchr*( ) shall return a pointer to the wide-character code or a null pointer if *wc* does not occur in the wide-character string.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcschr( ), 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 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

wcsrtombs — convert a wide-character string to a character string (restartable)

SYNOPSIS

```
#include <wchar.h>

size_t wcsrtombs(char *restrict dst, const wchar_t **restrict src,
    size_t len, mbstate_t *restrict ps);
```

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 *wcsrtombs*( ) function shall convert a sequence of wide characters from the array indirectly pointed to by *src* into a sequence of corresponding characters, beginning in the conversion state described by the object pointed to by *ps*. If *dst* is not a null pointer, the converted characters are then stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null wide character, which is also stored. Conversion stops earlier in the following cases:

- When a code is reached that does not correspond to a valid character
- When the next character would exceed the limit of *len* total bytes to be stored in the array pointed to by *dst* (and *dst* is not a null pointer)

Each conversion takes place as if by a call to the *wcrtomb*( ) function.

If *dst* is not a null pointer, the pointer object pointed to by *src* is assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the last wide character converted (if any). If conversion stopped due to reaching a terminating null wide character, the resulting state described is the initial conversion state.

If *ps* is a null pointer, the *wcsrtombs*( ) function uses its own internal **mbstate_t** object, which is initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by *ps* is used to completely describe the current conversion state of the associated character sequence. The implementation shall behave as if no function defined in this volume of IEEE Std. 1003.1-200x calls *wcsrtombs*( ).

CX If the application uses any of the _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS functions, the application shall ensure that the *wcsrtombs*( ) function is called with a non-NULL *ps* argument.

XSI The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

RETURN VALUE

If conversion stops because a code is reached that does not correspond to a valid character, an encoding error occurs. In this case, the *wcsrtombs*( ) function shall store the value of the macro [EILSEQ] in *errno* and return (**size_t**)–1; the conversion state is undefined. Otherwise, it shall return the number of bytes in the resulting character sequence, not including the terminating null (if any).

ERRORS

The *wcsrtombs*( ) function may fail if:

CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

[EILSEQ] A wide-character code does not correspond to a valid character.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbsinit( ), *wcrtomb*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

In the DESCRIPTION, a note on using this function in a threaded application is added.

Extensions beyond the ISO C standard are now marked.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The *wcsrtombs*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wcsspn — get length of a wide substring

SYNOPSIS

```
#include <wchar.h>

size_t wcsspn(const wchar_t *ws1, const wchar_t *ws2);
```

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 *wcsspn*( ) function shall compute the length of the maximum initial segment of the wide-character string pointed to by *ws1* which consists entirely of wide-character codes from the wide-character string pointed to by *ws2*.

RETURN VALUE

The *wcsspn*( ) function shall return the length of the initial substring of *ws1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcscspn( ), 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

The RETURN VALUE section is updated to indicate that *wcsspn*( ) returns the length of *ws1* rather that *ws1* itself.

NAME

wcsstr — find a wide-character substring

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcsstr(const wchar_t *restrict ws1, const wchar_t *restrict ws2);
```

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 *wcsstr*( ) function shall locate the first occurrence in the wide-character string pointed to by *ws1* of the sequence of wide characters (excluding the terminating null wide character) in the wide-character string pointed to by *ws2*.

RETURN VALUE

Upon successful completion, *wcsstr*( ) shall return a pointer to the located wide-character string, or a null pointer if the wide-character string is not found.

If *ws2* points to a wide-character string with zero length, the function shall return *ws1*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcschr( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The *wcsstr*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wcstod, wcstof, wcstold — convert a wide-character string to a double-precision number

SYNOPSIS

```
#include <wchar.h>

double wcstod(const wchar_t *restrict nptr, wchar_t **restrict endptr);
float wcstof(const wchar_t *restrict nptr, wchar_t **restrict endptr);
long double wcstold(const wchar_t *restrict nptr,
    wchar_t **restrict endptr);
```

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 convert the initial portion of the wide-character string pointed to by *nptr* to **double**, **float**, and **long double** representation, respectively. First, they decompose the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by *iswspace*( ))
2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN
3. A final wide-character string of one or more unrecognized wide-character codes, including the terminating null wide-character code of the input wide-character string

Then it attempts to convert the subject sequence to a floating-point number, and returns the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character, then an optional exponent part
- A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix character, then an optional binary exponent part
- One of INF or INFINITY, or any other wide string equivalent except for case
- One of NAN or NAN(*n-wchar-sequence*$_{opt}$), or any other wide string ignoring case in the NAN part, where:

```
n-wchar-sequence:
    digit
    nondigit
    n-wchar-sequence digit
    n-wchar-sequence nondigit
```

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of wide characters starting with the first digit or the radix character (whichever occurs first) is interpreted as a floating constant according to the rules of the C language, except that the radix character is used in place of a period, and that if neither an exponent part nor a radix character appears in a decimal floating-point number, or if a binary exponent part does not appear in a

hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated. A wide-character sequence INF or INFINITY is interpreted as an infinity, if representable in the return type, else like a floating constant that is too large for the range of the return type. A wide-character sequence NAN or NAN(*n-wchar-sequence*$_{opt}$) is interpreted as a quiet NaN, if supported in the return type, else like a subject sequence part that does not have the expected form; the meaning of the *n*-wchar sequences is implementation-defined. A pointer to the final wide string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence has the hexadecimal form and FLT_RADIX is a power of 2, the value resulting from the conversion is correctly rounded.

CX The radix character is defined in the program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period ('.').

CX In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

The *wcstod*( ) function shall not change the setting of *errno* if successful.

Because 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set *errno* to 0, then call *wcstod*( ), then check *errno*.

RETURN VALUE

Upon successful completion, these functions shall return the converted value. If no conversion could be performed, 0 shall be returned and *errno* may be set to [EINVAL].

If the correct value is outside the range of representable values, HUGE_VAL, HUGE_VALF, or HUGE_VALL shall be returned (according to the sign of the value), and *errno* shall be set to [ERANGE].

If the correct value would cause underflow, a value whose magnitude is no greater than the smallest normalized positive number in the return type shall be returned and *errno* set to [ERANGE].

ERRORS

The *wcstod*( ) function shall fail if:

[ERANGE] The value to be returned would cause overflow or underflow.

The *wcstod*( ) function may fail if:

CX [EINVAL] No conversion could be performed.

EXAMPLES

None.

APPLICATION USAGE

If the subject sequence has the hexadecimal form and FLT_RADIX is not a power of 2, 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.

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 <= D <= 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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswspace( ), *localeconv*( ), *scanf*( ), *setlocale*( ), *wcstol*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<float.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

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 now 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 *wcstod*( ) prototype is updated.
- The *wcstof*( ) and *wcstold*( ) functions are added.
- The DESCRIPTION, RETURN VALUE, and APPLICATION USAGE sections are extensively updated.

50123 **NAME**
50124 wcstoimax, wcstoumax — convert wide-character string to integer type

50125 **SYNOPSIS**
50126 ```
#include <stddef.h>
```
50127 ```
#include <inttypes.h>
```
50128 ```
intmax_t wcstoimax(const wchar_t *restrict nptr,
```
50129 ```
    wchar_t **restrict endptr, int base);
```
50130 ```
uintmax_t wcstoumax(const wchar_t *restrict nptr,
```
50131 ```
    wchar_t **restrict endptr, int base);
```

50132 **DESCRIPTION**
50133 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50134 conflict between the requirements described here and the ISO C standard is unintentional. This
50135 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

50136 These functions shall be equivalent to the *wcstol*( ), *wcstoll*( ), *wcstoul*( ), and *wcstoull*( ) functions,
50137 respectively, except that the initial portion of the wide string shall be converted to **intmax_t** and
50138 **uintmax_t** representation, respectively.

50139 **RETURN VALUE**
50140 These functions shall return the converted value, if any.

50141 If no conversion could be performed, zero shall be returned. If the correct value is outside the
50142 range of representable values, {INTMAX_MAX}, {INTMAX_MIN}, or {UINTMAX_MAX} shall
50143 be returned (according to the return type and sign of the value, if any), and *errno* shall be set to
50144 [ERANGE].

50145 **ERRORS**
50146 These functions shall fail if:

50147 [EINVAL] The value of *base* is not supported.

50148 [ERANGE] The value to be returned is not representable.

50149 These functions may fail if:

50150 [EINVAL] No conversion could be performed.

50151 **EXAMPLES**
50152 None.

50153 **APPLICATION USAGE**
50154 None.

50155 **RATIONALE**
50156 None.

50157 **FUTURE DIRECTIONS**
50158 None.

50159 **SEE ALSO**
50160 *wcstol*( ), *wcstoul*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<inttypes.h>**,
50161 **<stddef.h>**

50162 **CHANGE HISTORY**
50163 First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

wcstok — split wide-character string into tokens

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcstok(wchar_t *restrict ws1, const wchar_t *restrict ws2,
    wchar_t **restrict ptr);
```

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.

A sequence of calls to *wcstok*( ) breaks the wide-character string pointed to by *ws1* into a sequence of tokens, each of which is delimited by a wide-character code from the wide-character string pointed to by *ws2*. The third argument points to a caller-provided **wchar_t** pointer into which the *wcstok*( ) function stores information necessary for it to continue scanning the same wide-character string.

The first call in the sequence has *ws1* as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by *ws2* may be different from call to call.

The first call in the sequence searches the wide-character string pointed to by *ws1* for the first wide-character code that is *not* contained in the current separator string pointed to by *ws2*. If no such wide-character code is found, then there are no tokens in the wide-character string pointed to by *ws1* and *wcstok*( ) returns a null pointer. If such a wide-character code is found, it is the start of the first token.

The *wcstok*( ) function then searches from there for a wide-character code that *is* contained in the current separator string. If no such wide-character code is found, the current token extends to the end of the wide-character string pointed to by *ws1*, and subsequent searches for a token shall return a null pointer. If such a wide-character code is found, it is overwritten by a null wide-character, which terminates the current token. The *wcstok*( ) function saves a pointer to the following wide-character code, from which the next search for a token shall start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The implementation shall behave as if no function calls *wcstok*( ).

RETURN VALUE

Upon successful completion, the *wcstok*( ) function shall return a pointer to the first wide-character code of a token. Otherwise, if there is no token, *wcstok*( ) shall return a null pointer.

ERRORS

No errors are defined.

50201 **EXAMPLES**

50202 None.

50203 **APPLICATION USAGE**

50204 None.

50205 **RATIONALE**

50206 None.

50207 **FUTURE DIRECTIONS**

50208 None.

50209 **SEE ALSO**

50210 The Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>** |

50211 **CHANGE HISTORY**

50212 First released in Issue 4.

50213 **Issue 5**

50214 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, a third argument is
50215 added to the definition of this function in the SYNOPSIS. |

50216 **Issue 6** |

50217 The *wcstok*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard. |

NAME

wcstol, wcstoll — convert a wide-character string to a long integer

SYNOPSIS

```
#include <wchar.h>

long wcstol(const wchar_t *restrict nptr, wchar_t **restrict endptr,
    int base);
long long wcstoll(const wchar_t *restrict nptr,
    wchar_t **restrict endptr, int base);
```

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 convert the initial portion of the wide-character string pointed to by *nptr* to **long**, **long long**, **unsigned long**, and **unsigned long long** representation, respectively. First, they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by *iswspace*( ))
2. A subject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. A final wide-character string of one or more unrecognized wide-character codes, including the terminating null wide-character code of the input wide-character string

Then it attempts to convert the subject sequence to an integer, and returns the result.

If *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant, any of which may be preceded by a '+' or '−' sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7' only. A hexadecimal constant consists of the prefix "0x" or "0X" followed by a sequence of the decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a '+' or '−' sign, but not including an integer suffix. The letters from 'a' (or 'A') to 'z' (or 'Z') inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the wide-character code representations of "0x" or "0X" may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first non-white-space wide-character code that is of the expected form. The subject sequence contains no wide-character codes if the input wide-character string is empty or consists entirely of white-space wide-character code, or if the first non-white-space wide-character code is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and *base* is 0, the sequence of wide-character codes starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final wide-character string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

50264 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
50265 accepted.

50266 If the subject sequence is empty or does not have the expected form, no conversion is performed;
50267 the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null
50268 pointer.

50269 The *wcstol*( ) function shall not change the setting of *errno* if successful.

50270 Because 0, {LONG_MIN} or {LLONG_MIN} and {LONG_MAX} or {LLONG_MAX} are returned
50271 on error and are also valid returns on success, an application wishing to check for error
50272 situations should set *errno* to 0, then call *wcstol*( ), then check *errno*.

50273 **RETURN VALUE**

50274 Upon successful completion, these functions shall return the converted value, if any. If no
50275 CX conversion could be performed, 0 shall be returned and *errno* may be set to indicate the error. If
50276 the correct value is outside the range of representable values, {LONG_MAX} or {LONG_MIN}
50277 shall be returned (according to the sign of the value), and *errno* set to [ERANGE].

50278 **ERRORS**

50279 These functions shall fail if:

50280 CX [EINVAL] The value of *base* is not supported.

50281 [ERANGE] The value to be returned is not representable.

50282 These functions may fail if:

50283 CX [EINVAL] No conversion could be performed.

50284 **EXAMPLES**

50285 None.

50286 **APPLICATION USAGE**

50287 None.

50288 **RATIONALE**

50289 None.

50290 **FUTURE DIRECTIONS**

50291 None.

50292 **SEE ALSO**

50293 *iswalpha*( ), *scanf*( ), *wcstod*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>**

50294 **CHANGE HISTORY**

50295 First released in Issue 4. Derived from the MSE working draft.

50296 **Issue 5**

50297 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

50298 **Issue 6**

50299 Extensions beyond the ISO C standard are now marked.

50300 The following new requirements on POSIX implementations derive from alignment with the
50301 Single UNIX Specification:

50302 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
50303 added if no conversion could be performed.

50304 The following changes are made for alignment with the ISO / IEC 9899: 1999 standard:

- The *wcstol*( ) prototype is updated.
- The *wcstoll*( ) function is added.

50307 **NAME**

50308 wcstombs — convert a wide-character string to a character string

50309 **SYNOPSIS**

50310 ```
#include <stdlib.h>
```

50311 ```
size_t wcstombs(char *restrict s, const wchar_t *restrict pwcs,
```

50312 ```
    size_t n);
```

50313 **DESCRIPTION**

50314 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50315 conflict between the requirements described here and the ISO C standard is unintentional. This
50316 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

50317 The *wcstombs*( ) function shall convert the sequence of wide-character codes that are in the array
50318 pointed to by *pwcs* into a sequence of characters that begins in the initial shift state and stores
50319 these characters into the array pointed to by *s*, stopping if a character would exceed the limit of *n*
50320 total bytes or if a null byte is stored. Each wide-character code is converted as if by a call to
50321 *wctomb*( ), except that the shift state of *wctomb*( ) is not affected.

50322 The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

50323 No more than *n* bytes shall be modified in the array pointed to by *s*. If copying takes place
50324 CX between objects that overlap, the behavior is undefined. If *s* is a null pointer, *wcstombs*( ) shall
50325 return the length required to convert the entire array regardless of the value of *n*, but no values
50326 are stored.

50327 The *wcstombs*( ) function need not be reentrant. A function that is not required to be reentrant is
50328 not required to be thread-safe.

50329 **RETURN VALUE**

50330 If a wide-character code is encountered that does not correspond to a valid character (of one or
50331 more bytes each), *wcstombs*( ) shall return (**size_t**)–1. Otherwise, *wcstombs*( ) shall return the
50332 number of bytes stored in the character array, not including any terminating null byte. The array
50333 shall not be null-terminated if the value returned is *n*.

50334 **ERRORS**

50335 The *wcstombs*( ) function may fail if:

50336 CX [EILSEQ] A wide-character code does not correspond to a valid character.

50337 **EXAMPLES**

50338 None.

50339 **APPLICATION USAGE**

50340 None.

50341 **RATIONALE**

50342 None.

50343 **FUTURE DIRECTIONS**

50344 None.

50345 **SEE ALSO**

50346 *mblen*( ), *mbtowc*( ), *mbstowcs*( ), *wctomb*( ), the Base Definitions volume of IEEE Std. 1003.1-200x,
50347 **<stdlib.h>**
```

50348 **CHANGE HISTORY**

50349 First released in Issue 4. Derived from the ISO C standard.

50350 **Issue 6**

50351 The following new requirements on POSIX implementations derive from alignment with the
50352 Single UNIX Specification:

50353 • The DESCRIPTION states the effect of when *s* is a null pointer.

50354 • The [EILSEQ] error condition is added.

50355 The *wcstombs*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

wcstoul, wcstoull — convert a wide-character string to an unsigned long

SYNOPSIS

```
#include <wchar.h>

long wcstoul(const wchar_t *restrict nptr, wchar_t **restrict endptr,
    int base);
long long wcstoull(const wchar_t *restrict nptr,
    wchar_t **restrict endptr, int base);
```

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 convert the initial portion of the wide-character string pointed to by *nptr* to **long**, **long long**, **unsigned long**, and **unsigned long long** representation, respectively. First, they decompose the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by *iswspace*( ))
2. A subject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. A final wide-character string of one or more unrecognized wide-character codes, including the terminating null wide-character code of the input wide-character string

Then it attempts to convert the subject sequence to an unsigned integer, and returns the result.

If *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant, any of which may be preceded by a `'+'` or `'−'` sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix `'0'` optionally followed by a sequence of the digits `'0'` to `'7'` only. A hexadecimal constant consists of the prefix `"0x"` or `"0X"` followed by a sequence of the decimal digits and letters `'a'` (or `'A'`) to `'f'` (or `'F'`) with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a `'+'` or `'−'` sign, but not including an integer suffix. The letters from `'a'` (or `'A'`) to `'z'` (or `'Z'`) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the wide-character codes `"0x"` or `"0X"` may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first wide-character code that is not white space and is of the expected form. The subject sequence contains no wide-character codes if the input wide-character string is empty or consists entirely of white-space wide-character codes, or if the first wide-character code that is not white space is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and *base* is 0, the sequence of wide-character codes starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final wide-character string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

50401 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

50403 If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

50406 The *wcstoul*( ) function shall not change the setting of *errno* if successful.

50407 Because 0, {ULONG_MAX}, and {ULLONG_MAX} are returned on error and 0 is also a valid return on success, an application wishing to check for error situations should set *errno* to 0, then call *wcstoul*( ), then check *errno*.

50410 **RETURN VALUE**

50411 Upon successful completion, these functions shall return the converted value, if any. If no
50412 CX conversion could be performed, 0 shall be returned and *errno* may be set to indicate the error. If
50413 the correct value is outside the range of representable values, {ULONG_MAX} shall be returned
50414 and *errno* set to [ERANGE].

50415 **ERRORS**

50416 These functions shall fail if:

50417 CX [EINVAL] The value of *base* is not supported.

50418 [ERANGE] The value to be returned is not representable.

50419 These functions may fail if:

50420 CX [EINVAL] No conversion could be performed.

50421 **EXAMPLES**

50422 None.

50423 **APPLICATION USAGE**

50424 None.

50425 **RATIONALE**

50426 None.

50427 **FUTURE DIRECTIONS**

50428 None.

50429 **SEE ALSO**

50430 *iswalpha*( ), *scanf*( ), *wcstod*( ), *wcstol*( ), the Base Definitions volume of IEEE Std. 1003.1-200x,
50431 **<wchar.h>**

50432 **CHANGE HISTORY**

50433 First released in Issue 4. Derived from the MSE working draft.

50434 **Issue 5**

50435 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

50436 **Issue 6**

50437 Extensions beyond the ISO C standard are now marked.

50438 The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

50440 - The [EINVAL] error condition is added for when the value of *base* is not supported.

50441 In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
50442 added if no conversion could be performed.

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

- The *wcstoul*() prototype is updated.
- The *wcstoull*() function is added.

NAME

wcswcs — find a wide substring (**LEGACY**)

SYNOPSIS

XSI
```
#include <wchar.h>

wchar_t *wcswcs(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

The *wcswcs*( ) function shall locate the first occurrence in the wide-character string pointed to by *ws1* of the sequence of wide-character codes (excluding the terminating null wide-character code) in the wide-character string pointed to by *ws2*.

RETURN VALUE

Upon successful completion, *wcswcs*( ) shall return a pointer to the located wide-character string or a null pointer if the wide-character string is not found.

If *ws2* points to a wide-character string with zero length, the function shall return *ws1*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

This function was not included in the final ISO/IEC 9899:1990/Amendment 1:1995 (E). Application developers are strongly encouraged to use the *wcsstr*( ) function instead.

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

wcschr( ), *wcsstr*( ), 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

Marked EX.

Issue 6

This function is marked LEGACY.

NAME

wcswidth — number of column positions of a wide-character string

SYNOPSIS

XSI
```
#include <wchar.h>

int wcswidth(const wchar_t *pwcs, size_t n);
```

DESCRIPTION

The *wcswidth*( ) function shall determine the number of column positions required for *n* wide-character codes (or fewer than *n* wide-character codes if a null wide-character code is encountered before *n* wide-character codes are exhausted) in the string pointed to by *pwcs*.

RETURN VALUE

The *wcswidth*( ) function either shall return 0 (if *pwcs* points to a null wide-character code), or return the number of column positions to be occupied by the wide-character string pointed to by *pwcs*, or return –1 (if any of the first *n* wide-character codes in the wide-character string pointed to by *pwcs* is not a printing wide-character code).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

This function was removed from the final ISO/IEC 9899:1990/Amendment 1:1995 (E), and the return value for a non-printable wide character is not specified.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcwidth( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>**, the Base Definitions volume of IEEE Std. 1003.1-200x, Section 3.106, Column Position

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 6

The Open Group corrigenda item U021/11 has been applied. The function is marked as an extension.

NAME

wcsxfrm — wide-character string transformation

SYNOPSIS

```
#include <wchar.h>

size_t wcsxfrm(wchar_t *restrict ws1, const wchar_t *restrict 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 *wcsxfrm*( ) function shall transform the wide-character string pointed to by *ws2* and place the resulting wide-character string into the array pointed to by *ws1*. The transformation is such that if *wcscmp*( ) is applied to two transformed wide strings, it returns a value greater than, equal to, or less than 0, corresponding to the result of *wcscoll*( ) applied to the same two original wide-character strings. No more than *n* wide-character codes are placed into the resulting array pointed to by *ws1*, including the terminating null wide-character code. If *n* is 0, *ws1* is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

CX The *wcsxfrm*( ) function shall not change the setting of *errno* if successful.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to 0, then call *wcsxfrm*( ), then check *errno*.

RETURN VALUE

The *wcsxfrm*( ) function shall return the length of the transformed wide-character string (not including the terminating null wide-character code). If the value returned is *n* or more, the contents of the array pointed to by *ws1* are indeterminate.

On error, the *wcsxfrm*( ) function may set *errno*, but no return value is reserved to indicate an error.

ERRORS

The *wcsxfrm*( ) function may fail if:

CX [EINVAL] The wide-character string pointed to by *ws2* contains wide-character codes outside the domain of the collating sequence.

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 program's locale (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 now 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.

NAME

wctob — wide-character to single-byte conversion

SYNOPSIS

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

int wctob(wint_t c);
```

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 *wctob*( ) function shall determine whether *c* corresponds to a member of the extended character set whose character representation is a single byte when in the initial shift state.

The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

RETURN VALUE

The *wctob*( ) function shall return EOF if *c* does not correspond to a character with length one in the initial shift state. Otherwise, it shall return the single-byte representation of that character as an **unsigned char** converted to **int**.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

btowc( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

50603 **NAME**

50604 wctomb — convert a wide-character code to a character

50605 **SYNOPSIS**

50606 `#include <stdlib.h>`

50607 `int wctomb(char *s, wchar_t wchar);`

50608 **DESCRIPTION**

50609 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50610 conflict between the requirements described here and the ISO C standard is unintentional. This
50611 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

50612 The *wctomb*( ) function shall determine the number of bytes needed to represent the character
50613 corresponding to the wide-character code whose value is *wchar* (including any change in the
50614 shift state). It stores the character representation (possibly multiple bytes and any special bytes
50615 to change shift state) in the array object pointed to by *s* (if *s* is not a null pointer). At most
50616 {MB_CUR_MAX} bytes are stored. If *wchar* is 0, a null byte is stored, preceded by any shift |
50617 sequence needed to restore the initial shift state, and *wctomb*( ) is left in the initial shift state. |

50618 CX The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a
50619 state-dependent encoding, this function is placed into its initial state by a call for which its
50620 character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null
50621 pointer cause the internal state of the function to be altered as necessary. A call with *s* as a null
50622 pointer causes this function to return a non-zero value if encodings have state dependency, and
50623 0 otherwise. Changing the *LC_CTYPE* category causes the shift state of this function to be
50624 indeterminate.

50625 The *wctomb*( ) function need not be reentrant. A function that is not required to be reentrant is
50626 not required to be thread-safe.

50627 The implementation shall behave as if no function defined in this volume of
50628 IEEE Std. 1003.1-200x calls *wctomb*( ).

50629 **RETURN VALUE**

50630 If *s* is a null pointer, *wctomb*( ) shall return a non-zero or 0 value, if character encodings,
50631 respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *wctomb*( )
50632 shall return –1 if the value of *wchar* does not correspond to a valid character, or return the
50633 number of bytes that constitute the character corresponding to the value of *wchar*.

50634 In no case shall the value returned be greater than the value of the {MB_CUR_MAX} macro.

50635 **ERRORS**

50636 No errors are defined.

50637 **EXAMPLES**

50638 None.

50639 **APPLICATION USAGE**

50640 None.

50641 **RATIONALE**

50642 None.

50643 **FUTURE DIRECTIONS**

50644 None.

SEE ALSO

mblen( ), *mbtowc*( ), *mbstowcs*( ), *wcstombs*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

Issue 6

Extensions beyond the ISO C standard are now marked.

In the DESCRIPTION, a note about reentrancy and thread-safety is added.

NAME

wctrans — define character mapping

SYNOPSIS

```
#include <wctype.h>

wctrans_t wctrans(const char *charclass);
```

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 *wctrans*( ) function is defined for valid character mapping names identified in the current locale. The *charclass* is a string identifying a generic character mapping name for which codeset-specific information is required. The following character mapping names are defined in all locales: **tolower** and **toupper**.

The function shall return a value of type **wctrans_t**, which can be used as the second argument to subsequent calls of *towctrans*( ). The *wctrans*( ) function determines values of **wctrans_t** according to the rules of the coded character set defined by character mapping information in the program's locale (category *LC_CTYPE*). The values returned by *wctrans*( ) are valid until a call to *setlocale*( ) that modifies the category *LC_CTYPE*.

RETURN VALUE

CX The *wctrans*( ) function shall return 0 and may set *errno* to indicate the error if the given character mapping name is not valid for the current locale (category *LC_CTYPE*); otherwise, it shall return a non-zero object of type **wctrans_t** that can be used in calls to *towctrans*( ).

ERRORS

The *wctrans*( ) function may fail if:

CX [EINVAL] The character mapping name pointed to by *charclass* is not valid in the current locale.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

towctrans( ), 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).

NAME

wctype — define character class

SYNOPSIS

```
#include <wctype.h>

wctype_t wctype(const char *property);
```

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 *wctype*( ) function is defined for valid character class names as defined in the current locale. The *property* is a string identifying a generic character class for which codeset-specific type information is required. The following character class names are defined in all locales:

alnum	**digit**	**punct**
alpha	**graph**	**space**
blank	**lower**	**upper**
cntrl	**print**	**xdigit**

Additional character class names defined in the locale definition file (category *LC_CTYPE*) can also be specified.

The function shall return a value of type **wctype_t**, which can be used as the second argument to subsequent calls of *iswctype*( ). The *wctype*( ) function determines values of **wctype_t** according to the rules of the coded character set defined by character type information in the program's locale (category *LC_CTYPE*). The values returned by *wctype*( ) are valid until a call to *setlocale*( ) that modifies the category *LC_CTYPE*.

RETURN VALUE

The *wctype*( ) function shall return 0 if the given character class name is not valid for the current locale (category *LC_CTYPE*); otherwise, it shall return an object of type **wctype_t** that can be used in calls to *iswctype*( ).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswctype( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wctype.h>**, **<wchar.h>**

CHANGE HISTORY

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 now made visible by inclusion of the header **<wctype.h>** rather than **<wchar.h>**.

NAME

wcwidth — number of column positions of a wide-character code

SYNOPSIS

XSI
```
#include <wchar.h>

int wcwidth(wchar_t wc);
```

DESCRIPTION

The *wcwidth*( ) function shall determine the number of column positions required for the wide character *wc*. The application shall ensure that the value of *wc* is a character representable as a **wchar_t**, and a wide-character code corresponding to a valid character in the current locale.

RETURN VALUE

The *wcwidth*( ) function shall either return 0 (if *wc* is a null wide-character code), or return the number of column positions to be occupied by the wide-character code *wc*, or return –1 (if *wc* does not correspond to a printing wide-character code).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

This function was removed from the final ISO/IEC 9899:1990/Amendment 1:1995 (E), and the return value for a non-printable wide character is not specified.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcswidth( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released as a World-wide Portability Interface in Issue 4. Derived from MSE working draft.

Issue 6

The Open Group corrigenda item U021/12 has been applied. This function is marked as an extension.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

50770 **NAME**

50771 wmemchr — find a wide character in memory

50772 **SYNOPSIS**

50773 `#include <wchar.h>`

50774 `wchar_t *wmemchr(const wchar_t *ws, wchar_t wc, size_t n);`

50775 **DESCRIPTION**

50776 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50777 conflict between the requirements described here and the ISO C standard is unintentional. This
50778 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

50779 The *wmemchr*( ) function shall locate the first occurrence of *wc* in the initial *n* wide characters of
50780 the object pointed to by *ws*. This function is not affected by locale and all **wchar_t** values are
50781 treated identically. The null wide character and **wchar_t** values not corresponding to valid
50782 characters are not treated specially.

50783 If *n* is zero, the application shall ensure that *ws* is a valid pointer and the function behaves as if
50784 no valid occurrence of *wc* is found.

50785 **RETURN VALUE**

50786 The *wmemchr*( ) function shall return a pointer to the located wide character, or a null pointer if
50787 the wide character does not occur in the object.

50788 **ERRORS**

50789 No errors are defined.

50790 **EXAMPLES**

50791 None.

50792 **APPLICATION USAGE**

50793 None.

50794 **RATIONALE**

50795 None.

50796 **FUTURE DIRECTIONS**

50797 None.

50798 **SEE ALSO**

50799 *wmemcmp*( ), *wmemcpy*( ), *wmemmove*( ), *wmemset*( ), the Base Definitions volume of |
50800 IEEE Std. 1003.1-200x, **<wchar.h>** |

50801 **CHANGE HISTORY**

50802 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
50803 (E).

50804 **Issue 6**

50805 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

wmemcmp — compare wide characters in memory

SYNOPSIS

```
#include <wchar.h>

int wmemcmp(const 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 *wmemcmp*( ) function shall compare the first *n* wide characters of the object pointed to by *ws1* to the first *n* wide characters of the object pointed to by *ws2*. This function is not affected by locale and all **wchar_t** values are treated identically. The null wide character and **wchar_t** values not corresponding to valid characters are not treated specially.

If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function behaves as if the two objects compare equal.

RETURN VALUE

The *wmemcmp*( ) function shall return an integer greater than, equal to, or less than zero, respectively, as the object pointed to by *ws1* is greater than, equal to, or less than the object pointed to by *ws2*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wmemchr( ), *wmemcpy*( ), *wmemmove*( ), *wmemset*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

50843 **NAME**

50844 wmemcpy — copy wide characters in memory

50845 **SYNOPSIS**

50846 `#include <wchar.h>`

50847 `wchar_t *wmemcpy(wchar_t *restrict ws1, const wchar_t *restrict ws2,`
50848 `size_t n);`

50849 **DESCRIPTION**

50850 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50851 conflict between the requirements described here and the ISO C standard is unintentional. This
50852 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

50853 The *wmemcpy*( ) function shall copy *n* wide characters from the object pointed to by *ws2* to the
50854 object pointed to by *ws1*. This function is not affected by locale and all **wchar_t** values are
50855 treated identically. The null wide character and **wchar_t** values not corresponding to valid
50856 characters are not treated specially.

50857 If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function
50858 copies zero wide characters.

50859 **RETURN VALUE**

50860 The *wmemcpy*( ) function shall return the value of *ws1*.

50861 **ERRORS**

50862 No errors are defined.

50863 **EXAMPLES**

50864 None.

50865 **APPLICATION USAGE**

50866 None.

50867 **RATIONALE**

50868 None.

50869 **FUTURE DIRECTIONS**

50870 None.

50871 **SEE ALSO**

50872 *wmemchr*( ), *wmemcmp*( ), *wmemmove*( ), *wmemset*( ), the Base Definitions volume of
50873 IEEE Std. 1003.1-200x, **<wchar.h>**

50874 **CHANGE HISTORY**

50875 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
50876 (E).

50877 **Issue 6**

50878 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

50879 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 takes 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 is not affected by locale and all **wchar_t** values are treated identically. The null wide character and **wchar_t** values not corresponding to valid characters are not treated specially.

If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function copies 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*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

50919 **NAME**

50920 wmemset — set wide characters in memory

50921 **SYNOPSIS**

50922 `#include <wchar.h>`

50923 `wchar_t *wmemset(wchar_t *ws, wchar_t wc, size_t n);`

50924 **DESCRIPTION**

50925 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50926 conflict between the requirements described here and the ISO C standard is unintentional. This
50927 volume of IEEE Std. 1003.1-200x defers to the ISO C standard.

50928 The *wmemset*( ) function shall copy the value of *wc* into each of the first *n* wide characters of the
50929 object pointed to by *ws*. This function is not affected by locale and all **wchar_t** values are treated
50930 identically. The null wide character and **wchar_t** values not corresponding to valid characters
50931 are not treated specially.

50932 If *n* is zero, the application shall ensure that *ws* is a valid pointer, and the function copies zero
50933 wide characters.

50934 **RETURN VALUE**

50935 The *wmemset*( ) functions shall return the value of *ws*.

50936 **ERRORS**

50937 No errors are defined.

50938 **EXAMPLES**

50939 None.

50940 **APPLICATION USAGE**

50941 None.

50942 **RATIONALE**

50943 None.

50944 **FUTURE DIRECTIONS**

50945 None.

50946 **SEE ALSO**

50947 *wmemchr*( ), *wmemcmp*( ), *wmemcpy*( ), *wmemmove*( ), the Base Definitions volume of |
50948 IEEE Std. 1003.1-200x, **<wchar.h>** |

50949 **CHANGE HISTORY**

50950 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
50951 (E).

50952 **Issue 6**

50953 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

wordexp, wordfree — perform word expansions

SYNOPSIS

```
#include <wordexp.h>

int wordexp(const char *restrict words, wordexp_t *restrict pwordexp,
    int flags);
void wordfree(wordexp_t *pwordexp);
```

DESCRIPTION

The *wordexp*( ) function shall perform word expansions and place the list of expanded words into *pwordexp*.

If the implementation supports the utilities defined in the Shell and Utilities volume of IEEE Std. 1003.1-200x, the *wordexp*( ) function performs word expansions as described in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.6, Word Expansions, subject to quoting as in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.2, Quoting, and places the list of expanded words into the structure pointed to by *pwordexp*.

The *words* argument is a pointer to a string containing one or more words to be expanded. The expansions shall be the same as would be performed by the command line interpreter if *words* were the part of a command line representing the arguments to a utility. Therefore, the application shall ensure that *words* does not contain an unquoted <newline> or any of the unquoted shell special characters `'|'`, `'&'`, `';'`, `'<'`, `'>'` except in the context of command substitution as specified in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.6.3, Command Substitution. It also shall not contain unquoted parentheses or braces, except in the context of command or variable substitution. If the argument *words* contains an unquoted comment character (number sign) that is the beginning of a token, *wordexp*( ) shall either treat the comment character as a regular character, or interpret it as a comment indicator and ignore the remainder of *words*.

If the implementation does not support the utilities defined in the Shell and Utilities volume of IEEE Std. 1003.1-200x, the word expansion is unspecified, but should be the same as that used by the command language interpreter used by the *system*( ) and *popen*( ) functions.

The structure type **wordexp_t** is defined in the header **<wordexp.h>** and includes at least the following members:

Member Type	Member Name	Description
size_t	*we_wordc*	Count of words matched by *words*.
char **	*we_wordv*	Pointer to list of expanded words.
size_t	*we_offs*	Slots to reserve at the beginning of *pwordexp->we_wordv*.

The *wordexp*( ) function stores the number of generated words into *pwordexp->we_wordc* and a pointer to a list of pointers to words in *pwordexp->we_wordv*. If the implementation supports the utilities defined in the Shell and Utilities volume of IEEE Std. 1003.1-200x, each individual field created during field splitting (see the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.6.5, Field Splitting) or path name expansion (see the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.6.6, Path Name Expansion) is a separate word in the *pwordexp->we_wordv* list. The words are in order as described in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.6, Word Expansions. The first pointer after the last word pointer shall be a null pointer. The expansion of special parameters described in the Shell and Utilities volume of IEEE Std. 1003.1-200x, Section 2.5.2, Special Parameters is unspecified.

It is the caller's responsibility to allocate the storage pointed to by *pwordexp*. The *wordexp*( ) function allocates other space as needed, including memory pointed to by *pwordexp->we_wordv*. The *wordfree*( ) function frees any memory associated with *pwordexp* from a previous call to *wordexp*( ).

The *flags* argument is used to control the behavior of *wordexp*( ). The value of *flags* is the bitwise-inclusive OR of zero or more of the following constants, which are defined in **<wordexp.h>**:

WRDE_APPEND — Append words generated to the ones from a previous call to *wordexp*( ).

WRDE_DOOFFS — Make use of *pwordexp->we_offs*. If this flag is set, *pwordexp->we_offs* is used to specify how many null pointers to add to the beginning of *pwordexp->we_wordv*. In other words, *pwordexp->we_wordv* shall point to *pwordexp->we_offs* null pointers, followed by *pwordexp->we_wordc* word pointers, followed by a null pointer.

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.

WRDE_REUSE — The *pwordexp* argument was passed to a previous successful call to *wordexp*( ), and has not been passed to *wordfree*( ). The result shall be the same as if the application had called *wordfree*( ) and then called *wordexp*( ) without WRDE_REUSE.

WRDE_SHOWERR — Do not redirect *stderr* to **/dev/null**.

WRDE_UNDEF — Report error on an attempt to expand an undefined shell variable.

The WRDE_APPEND flag can be used to append a new set of words to those generated by a previous call to *wordexp*( ). The following rules apply to applications when two or more calls to *wordexp*( ) are made with the same value of *pwordexp* and without intervening calls to *wordfree*( ):

1. The first such call shall not set WRDE_APPEND. All subsequent calls shall set it.
2. All of the calls shall set WRDE_DOOFFS, or all shall not set it.
3. After the second and each subsequent call, *pwordexp->we_wordv* shall point to a list containing the following:
   a. Zero or more null pointers, as specified by WRDE_DOOFFS and *pwordexp->we_offs*
   b. Pointers to the words that were in the *pwordexp->we_wordv* list before the call, in the same order as before
   c. Pointers to the new words generated by the latest call, in the specified order
4. The count returned in *pwordexp->we_wordc* shall be the total number of words from all of the calls.
5. The application can change any of the fields after a call to *wordexp*( ), but if it does it shall reset them to the original value before a subsequent call, using the same *pwordexp* value, to *wordfree*( ) or *wordexp*( ) with the WRDE_APPEND or WRDE_REUSE flag.

If the implementation supports the utilities defined in the Shell and Utilities volume of IEEE Std. 1003.1-200x, and *words* contains an unquoted character—<newline>, `'|'`, `'&'`, `';'`, `'<'`, `'>'`, `'('`, `')'`, `'{'`, `'}'`—in an inappropriate context, *wordexp*( ) shall fail, and the number of expanded words shall be 0.

Unless WRDE_SHOWERR is set in *flags*, *wordexp*( ) shall redirect *stderr* to **/dev/null** for any utilities executed as a result of command substitution while expanding *words*. If WRDE_SHOWERR is set, *wordexp*( ) may write messages to *stderr* if syntax errors are detected while expanding *words*.

The application shall ensure that if WRDE_DOOFFS is set, then *pwordexp->we_offs* has the same value for each *wordexp*( ) call and *wordfree*( ) call using a given *pwordexp*.

The following constants are defined as error return values:

WRDE_BADCHAR	One of the unquoted characters—<newline>, `'|'`, `'&'`, `';'`, `'<'`, `'>'`, `'('`, `')'`, `'{'`, `'}'`—appears in *words* in an inappropriate context.
WRDE_BADVAL	Reference to undefined shell variable when WRDE_UNDEF is set in *flags*.
WRDE_CMDSUB	Command substitution requested when WRDE_NOCMD was set in flags.
WRDE_NOSPACE	Attempt to allocate memory failed.
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 return no 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 file name (or list of file names) 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 path name 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> character 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 substitution. In any case, it would have to scan *words* for unquoted special characters.

Another proposal is to add the following options to *sh*:

–w *wordlist*
: This option provides a wordlist expansion service to applications. The words in *wordlist* shall be expanded and the following written to standard output:

  1. The count of the number of words after expansion, in decimal, followed by a null byte
  2. The number of bytes needed to represent the expanded words (not including null separators), in decimal, followed by a null byte
  3. The expanded words, each terminated by a null byte

  If an error is encountered during word expansion, *sh* exits with a non-zero status after writing the former to report any words successfully expanded

–P Run in ''protected'' mode. If specified with the **–w** option, no command substitution shall be performed.

With these options, *wordexp*( ) could be implemented fairly simply by creating a subprocess using *fork*( ) and executing *sh* using the line:

```
execl(<shell path>, "sh", "-P", "-w", words, (char *)0);
```

after directing standard error to **/dev/null**.

It seemed objectionable for a library routine to write messages to standard error, unless explicitly requested, so *wordexp*( ) is required to redirect standard error to **/dev/null** to ensure that no messages are generated, even for commands executed for command substitution. The WRDE_SHOWERR flag can be specified to request that error messages be written.

The WRDE_REUSE flag allows the implementation to avoid the expense of freeing and reallocating memory, if that is possible. A minimal implementation can call *wordfree*( ) when WRDE_REUSE is set.

FUTURE DIRECTIONS

None.

SEE ALSO

fnmatch( ), *glob*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<wordexp.h>**, the Shell and Utilities volume of IEEE Std. 1003.1-200x

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 5

Moved from POSIX2 C-language Binding to BASE.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements. |

The **restrict** keyword is added to the *wordexp*() prototype for alignment with the ISO/IEC 9899:1999 standard. | |

51133 **NAME**

51134 wprintf — print formatted wide-character output

51135 **SYNOPSIS**

51136 `#include <stdio.h>`

51137 `#include <wchar.h>`

51138 `int wprintf(const wchar_t *format, ...);`

51139 **DESCRIPTION**

51140 Refer to *fwprintf*( ).

NAME

pwrite, write, writev — write on a file

SYNOPSIS

```
#include <unistd.h>

ssize_t pwrite(int fildes, const void *buf, size_t nbyte,
    off_t offset);
ssize_t write(int fildes, const void *buf, size_t nbyte);

#include <sys/uio.h>

ssize_t writev(int fildes, const struct iovec *iov, int iovcnt);
```

DESCRIPTION

The *pwrite*( ) function performs the same action as *write*( ), except that it writes into a given position without changing the file pointer. The first three arguments to *pwrite*( ) are the same as *write*( ) with the addition of a fourth argument offset for the desired position inside the file.

Notes to Reviewers

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

D3, XSH, ERN 676 says that pwrite() (and pread()) need to be limited to seekable devices or have an explicit statement that the offset argument is ignored. This item is still open.

The *write*( ) function shall attempt to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor, *fildes*.

If *nbyte* is zero and the file is a regular file, the *write*( ) function may detect and return errors as described below. In the absence of errors, or if error detection is not performed, the *write*( ) function shall return zero and have no other results. If *nbyte* is zero and the file is not a regular file, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file offset associated with *fildes*. Before successful return from *write*( ), the file offset is incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file shall be set to this file offset.

On a file not capable of seeking, writing always takes place starting at the current position. The value of a file offset associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file offset shall be set to the end of the file prior to each write and no intervening file modification operation shall occur between changing the file offset and the write operation.

XSI If a *write*( ) requests that more bytes be written than there is room for (for example, the process' file size limit or the physical end of a medium), only as many bytes as there is room for shall be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes shall return 20. The next write of a non-zero number of bytes shall give a XSI failure return (except as noted below) and the implementation shall generate a SIGXFSZ signal for the thread.

If *write*( ) is interrupted by a signal before it writes any data, it shall return –1 with *errno* set to [EINTR].

If *write*( ) is interrupted by a signal after it successfully writes some data, it shall return the number of bytes written.

If the value of *nbyte* is greater than {SSIZE_MAX}, the result is implementation-defined.

After a *write*() to a regular file has successfully returned:

- Any successful *read*() from each byte position in the file that was modified by that write shall return the data specified by the *write*() for that position until such byte positions are again modified.
- Any subsequent successful *write*() to the same byte position in the file shall overwrite that file data.

Write requests to a pipe or FIFO shall be handled the same as a regular file with the following exceptions:

- There is no file offset associated with a pipe, hence each write request shall append to the end of the pipe.
- 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.
- If the O_NONBLOCK flag is clear, a write request may cause the thread to block, but on normal completion it shall return *nbyte*.
- If the O_NONBLOCK flag is set, *write*() requests shall be handled differently, in the following ways:
  - The *write*() function shall not block the thread.
  - 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].
  - A write request for more than {PIPE_BUF} bytes shall cause one of the following:
    - 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.
    - When no data can be written, transfer no data, and return –1 with *errno* set to [EAGAIN].

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:

- If the O_NONBLOCK flag is clear, *write*() shall block the calling thread until the data can be accepted.
- 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].

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.

XSR If *fildes* 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*

bytes shall be written. If *nbyte* does not fall within the range and the minimum packet size value is 0, *write*( ) shall break the buffer into maximum packet size segments prior to sending the data downstream (the last segment may contain less than the maximum packet size). If *nbyte* does not fall within the range and the minimum value is non-zero, *write*( ) shall fail with *errno* set to [ERANGE]. Writing a zero-length buffer (*nbyte* is 0) to a STREAMS device sends 0 bytes with 0 returned. However, writing a zero-length buffer to a STREAMS-based pipe or FIFO sends no message and 0 is returned. The process may issue I_SWROPT *ioctl*( ) to enable zero-length messages to be sent across the pipe or FIFO.

When writing to a STREAM, data messages are created with a priority band of 0. When writing to a STREAM that is not a pipe or FIFO:

- If O_NONBLOCK is clear, and the STREAM cannot accept data (the STREAM write queue is full due to internal flow control conditions), *write*( ) shall block until data can be accepted.
- If O_NONBLOCK is set and the STREAM cannot accept data, *write*( ) shall return –1 and set *errno* to [EAGAIN].
- If O_NONBLOCK is set and part of the buffer has been written while a condition in which the STREAM cannot accept additional data occurs, *write*( ) shall terminate and return the number of bytes written.

In addition, *write*( ) and *writev*( ) shall fail if the STREAM head has processed an asynchronous error before the call. In this case, the value of *errno* does not reflect the result of *write*( ) or *writev*( ), but reflects the prior error.

XSI The *writev*( ) function is equivalent to *write*( ), but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt*–1]. *iovcnt* is valid if greater than 0 and less than or equal to {IOV_MAX}, defined in **<limits.h>**.

Each *iovec* entry specifies the base address and length of an area in memory from which data should be written. The *writev*( ) function shall always write a complete area before proceeding to the next.

If *fildes* refers to a regular file and all of the *iov_len* members in the array pointed to by *iov* are 0, *writev*( ) shall return 0 and have no other effect. For other file types, the behavior is unspecified.

If the sum of the *iov_len* values is greater than {SSIZE_MAX}, the operation fails and no data is transferred.

SIO If the O_DSYNC bit has been set, write I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion.

If the O_SYNC bit has been set, write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.

SHM If *fildes* refers to a shared memory object, the result of the *write*( ) function is unspecified.

TYM If *fildes* refers to a typed memory object, the result of the *write*( ) function is unspecified.

For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with *fildes*.

If *fildes* refers to a socket, *write*( ) is equivalent to *send*( ) with no flags set.

RETURN VALUE

XSI Upon successful completion, *write*( ) and *pwrite*( ) shall return the number of bytes actually written to the file associated with *fildes*. This number shall never be greater than *nbyte*. Otherwise, –1 shall be returned and *errno* set to indicate the error.

51271 XSI Upon successful completion, *writev*( ) shall return the number of bytes actually written.
51272 Otherwise, it shall return a value of −1, the file-pointer shall remain unchanged, and *errno* shall
51273 be set to indicate an error.

51274 **ERRORS**

51275 XSI The *write*( ),*pwrite*( ), and *writev*( )functions shall fail if:

51276 [EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the thread would be
51277 delayed in the *write*( ) operation.

51278 [EBADF] The *fildes* argument is not a valid file descriptor open for writing.

51279 [EFBIG] An attempt was made to write a file that exceeds the implementation-defined
51280 XSI maximum file size or the process' file size limit.

51281 [EFBIG] The file is a regular file, *nbyte* is greater than 0, and the starting position is
51282 greater than or equal to the offset maximum established in the open file
51283 description associated with *fildes*.

51284 [EINTR] The write operation was terminated due to the receipt of a signal, and no data
51285 was transferred.

51286 [EIO] The process is a member of a background process group attempting to write
51287 to its controlling terminal, TOSTOP is set, the process is neither ignoring nor
51288 blocking SIGTTOU, and the process group of the process is orphaned. This
51289 error may also be returned under implementation-defined conditions.

51290 [ENOSPC] There was no free space remaining on the device containing the file.

51291 [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
51292 any process, or that only has one end open. A SIGPIPE signal shall also be sent
51293 to the thread.

51294 XSR [ERANGE] The transfer request size was outside the range supported by the STREAMS
51295 file associated with *fildes*.

51296 MAN The *write*( ) function shall fail if:

51297 [EAGAIN] or [EWOULDBLOCK]
51298 The file descriptor is for a connection-mode socket, is marked
51299 O_NONBLOCK, and write would block.

51300 [ECONNRESET] A write was attempted on a connection-mode socket that is not connected.

51301 [EPIPE] A write was attempted on a connection-mode socket that is shut down for
51302 writing, or is no longer connected. In the latter case, if the socket is of type
51303 SOCK_STREAM, the SIGPIPE signal is generated to the calling process.
51304

51305 The *writev*( ) function shall fail if:

51306 XSI [EINVAL] The sum of the *iov_len* values in the *iov* array would overflow an **ssize_t**.

51307 XSI The *write*( ),*pwrite*( ), and *writev*( )functions may fail if:

51308 XSR [EINVAL] The STREAM or multiplexer referenced by *fildes* is linked (directly or
51309 indirectly) downstream from a multiplexer.

51310 MAN [EIO] A physical I/O error has occurred.

51311 MAN [ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

XSR [ENXIO] A hangup occurred on the STREAM being written to.

XSR A write to a STREAMS file may fail if an error message has been received at the STREAM head. In this case, *errno* is set to the value included in the error message.

MAN The *write*( ) function may fail if:

[EACCES] A write was attempted on a connection-mode socket and the calling process does not have appropriate privileges.

[ENETDOWN] A write was attempted on a connection-mode socket and the local network interface used to reach the destination is down.

[ENETUNREACH] A write was attempted on a connection-mode socket and no route to the network is present.

The *writev*( ) function may fail and set *errno* to:

XSI [EINVAL] The *iovcnt* argument was less than or equal to 0, or greater than {IOV_MAX}.

XSI The *pwrite*( ) function shall fail and the file pointer remain unchanged if:

XSI [EINVAL] The *offset* argument is invalid. The value is negative.

XSI [ESPIPE] *fildes* is associated with a pipe or FIFO.

EXAMPLES

Writing from a Buffer

The following example writes data from the buffer pointed to by *buf* to the file associated with the file descriptor *fd*.

```
#include <sys/types.h>
#include <string.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_written;
int fd;
...
strcpy(buf, "This is a test\n");
nbytes = strlen(buf);

bytes_written = write(fd, buf, nbytes);
...
```

Writing Data from an Array

The following example writes data from the buffers specified by members of the *iov* array to the file associated with the file descriptor *fd*.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
...
ssize_t bytes_written;
int fd;
char *buf0 = "short string\n";
char *buf1 = "This is a longer string\n";
char *buf2 = "This is the longest string in this example\n";
int iovcnt;
struct iovec iov[3];

iov[0].iov_base = buf0;
iov[0].iov_len = strlen(buf0);
iov[1].iov_base = buf1;
iov[1].iov_len = strlen(buf1);
iov[2].iov_base = buf2;
iov[2].iov_len = strlen(buf2);
...
iovcnt = sizeof(iov) / sizeof(struct iovec);

bytes_written = writev(fd, iov, iovcnt);
...
```

APPLICATION USAGE

None.

RATIONALE

See also the RATIONALE section in *read*( ).

An attempt to write to a pipe or FIFO has several major characteristics:

- *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.
- *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 process 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.
- *Complete/partial/deferred*: A write request:

```
int fildes;
size_t nbyte;
ssize_t ret;
char *buf;
```

```
ret = write(fildes, buf, nbyte);
```

may return:

complete *ret=nbyte*

partial *ret<nbyte*

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*.

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**	*nbyte*
nbyte≤{PIPE_BUF}	Atomic blocking *nbyte*	Atomic blocking *nbyte*	Atomic immediate *nbyte*
nbyte>{PIPE_BUF}	Blocking *nbyte*	Blocking *nbyte*	Blocking *nbyte*

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**	*nbyte*
nbyte≤{PIPE_BUF}	–1, [EAGAIN]	–1, [EAGAIN]	Atomic *nbyte*
nbyte>{PIPE_BUF}	–1, [EAGAIN]	<*nbyte* or –1, [EAGAIN]	≤*nbyte* 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*( ).

Writes can be serialized with respect to other reads and writes. If a *read*( ) of file data can be proven (by any means) to occur after a *write*( ) of the data, it must reflect that *write*( ), even if the calls are made by different processes. A similar requirement applies to multiple write operations to the same file position. This is needed to guarantee the propagation of data from *write*( ) calls to subsequent *read*( ) calls. This requirement is particularly significant for networked file systems, where some caching schemes violate these semantics.

Note that this is specified in terms of *read*( ) and *write*( ). Additional calls, such as the common *readv*( ) and *writev*( ), would want to obey these semantics. A new ''high-performance'' write analog that did not follow these serialization requirements would also be permitted by this wording. This volume of IEEE Std. 1003.1-200x is also silent about any effects of application-level caching (such as that done by *stdio*).

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.

This volume of IEEE Std. 1003.1-200x does not specify behavior of concurrent writes to a file from multiple processes. Applications should use some form of concurrency control.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *creat*( ), *dup*( ), *fcntl*( ), *getrlimit*( ), *lseek*( ), *open*( ), *pipe*( ), *ulimit*( ), the Base Definitions volume of IEEE Std. 1003.1-200x, **<limits.h>**, **<stropts.h>**, **<sys/uio.h>**, **<unistd.h>**

CHANGE HISTORY

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

Issue 4

The **<unistd.h>** header is added to the SYNOPSIS section.

Reference to *ulimit* in the DESCRIPTION is marked as an extension.

Reference to the process' file size limit and the *ulimit*( ) function are marked as extensions in the description of the [EFBIG] error.

The [ENXIO] error is marked as an extension.

The APPLICATION USAGE section is removed.

The description of [EINTR] is amended.

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The type of the argument *buf* is changed from **char*** to **const void***, and the type of the argument *nbyte* is changed from **unsigned** to **size_t**.
- The DESCRIPTION is changed as follows:
  - Writing at end-of-file is atomic.
  - {SSIZE_MAX} is now used to determine the maximum value of *nbyte*.
  - The consequences of activities after a call to the *write*( ) function are added.
  - To improve clarity, the text describing operations on pipes or FIFOs when O_NONBLOCK is set is restructured.

Issue 4, Version 2

The following changes are incorporated for X/OPEN UNIX conformance:

- The *writev*( ) function is added to the SYNOPSIS.
- The DESCRIPTION is updated to describe the writing of data to STREAMS files, an operational description of the *writev*( ) function is included, and a statement is added indicating that SIGXFSZ is generated if an attempted write operation would cause the maximum file size to be exceeded.
- The RETURN VALUE section is updated to describe values returned by the *writev*( ) function.
- The ERRORS section has been restructured to describe errors that apply to both *write*( ) and *writev*( ) apart from those that apply to *writev*( ) specifically. The [EIO], [ERANGE], and [EINVAL] errors are also added.

Issue 5

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

Large File Summit extensions are added.

The *pwrite*( ) function is added.

Issue 6

The DESCRIPTION states that the *write*( ) function does not block the thread. Previously this said ‘‘process’’ rather than ‘‘thread’’.

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:

- The DESCRIPTION now states that if *write*( ) is interrupted by a signal after it has successfully written some data, it returns the number of bytes written. In earlier versions of this volume of IEEE Std. 1003.1-200x, it was optional whether *write*( ) returned the number of bytes written, or whether it returned –1 with *errno* set to [EINTR]. This is a FIPS requirement.
- The following changes are made to support large files:
  - For regular files, no data transfer occurs past the offset maximum established in the open file description associated with the *fildes*.

— 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 changed to ‘‘may fail’’.

The [ENOBUFS] error is added for sockets.

The following error conditions are added for operations on sockets: [EACCES], [ENETDOWN], and [ENETUNREACH].

51535 **NAME**

51536 wscanf — convert formatted wide-character input

51537 **SYNOPSIS**

51538 `#include <stdio.h>`

51539 `#include <wchar.h>`

51540 `int wscanf(const wchar_t *format, ... );`

51541 **DESCRIPTION**

51542 Refer to *fwscanf*( ).

51543 **NAME**

51544 y0, y1, yn — Bessel functions of the second kind

51545 **SYNOPSIS**

51546 XSI
```
#include <math.h>

double y0(double x);
double y1(double x);
double yn(int n, double x);
```
51550

51551 **DESCRIPTION**

51552 The *y0*(), *y1*(), and *yn*() functions shall compute Bessel functions of *x* of the second kind of
51553 orders 0, 1, and *n* respectively. The application shall ensure that the value of *x* is positive.

51554 An application wishing to check for error situations should set *errno* to 0 before calling *y0*(),
51555 *y1*(), or *yn*(). If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

51556 **RETURN VALUE**

51557 Upon successful completion, *y0*(), *y1*(), and *yn*() shall return the relevant Bessel value of *x* of
51558 the second kind.

51559 If *x* is NaN, NaN shall be returned and *errno* may be set to [EDOM].

51560 If the *x* argument to *y0*(), *y1*(), or *yn*() is negative, −HUGE_VAL or NaN shall be returned, and
51561 *errno* may be set to [EDOM].

51562 If *x* is 0.0, −HUGE_VAL shall be returned and *errno* may be set to [ERANGE] or [EDOM].

51563 If the correct result would cause underflow, 0.0 shall be returned and *errno* may be set to
51564 [ERANGE].

51565 If the correct result would cause overflow, −HUGE_VAL or 0.0 shall be returned and *errno* may
51566 be set to [ERANGE].

51567 **ERRORS**

51568 The *y0*(), *y1*(), and *yn*() functions may fail if:

51569 [EDOM] The value of *x* is negative or NaN. |

51570 [ERANGE] The value of *x* is too large in magnitude, or *x* is 0.0, or the correct result would |
51571 cause overflow or underflow.

51572 No other errors shall occur.

51573 **EXAMPLES**

51574 None.

51575 **APPLICATION USAGE**

51576 None.

51577 **RATIONALE**

51578 None.

51579 **FUTURE DIRECTIONS**

51580 None.

51581 **SEE ALSO**

51582 *isnan*(), *j0*(), the Base Definitions volume of IEEE Std. 1003.1-200x, **<math.h>** |

51583 **CHANGE HISTORY**

51584 First released in Issue 1. Derived from Issue 1 of the SVID.

51585 **Issue 4**

51586 References to *matherr*( ) are removed.

51587 The RETURN VALUE and ERRORS sections are substantially rewritten to rationalize error
51588 handling in the mathematics functions.

51589 **Issue 5**

51590 The DESCRIPTION is updated to indicate how an application should check for an error. This
51591 text was previously published in the APPLICATION USAGE section.

51592 **Issue 6**

51593 The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

51594