Annex I
(informative)

Device Control Considerations

I.1 Introduction

Realtime systems interact with their physical environment using a variety of devices (such as analog-digital converters, digital-analog converters, counters, and video graphic equipment), which provide a set of services that cannot be fully utilized in terms of read and/or write semantics.

A driver for a special device will normally not be portable between system implementations, but an application that uses such a driver can be made portable if all functions calling the driver are well defined and standardized. Users and integrators of realtime systems often add drivers for special devices, and a standardized function format for interfacing with these devices greatly simplifies this process.

This section defines a general method for interfacing to the widest possible range of special devices. The term "special device" refers to the hardware; access to the driver for this hardware uses the file abstraction "character special file".

I.2 Concepts

The term "special device" refers to hardware; access to the driver for this hardware uses the file abstraction "character special file". The implementation shall provide the means to integrate a device driver into the system and provide a way to create a character special file that provides a binding to the device driver when the *open*() function is called with the name of that character special file. The means available to integrate drivers into the system and the way character special files are created is implementation defined.

I.3 Definitions

⇒ **2.2.2 General Terms** *Replace the contents of the subclause designated the definition of "character special file" with the following:*

I.3.0.1 character special file:

A file that refers to a device. One specific type of character special file is a terminal device file, whose access is defined in 7.1. Other character special files have no structure defined by this part of ISO/IEC 9945, but they can be accessed as defined in 21.

⇒ **2.2.2 General Terms** *Add the following definition, in the right sorted order:*

I.3.0.2 driver:

A part of the implementation (possibly user supplied) that controls a device (2.2.2.x).

I.4 Errors

⇒ **2.4 Error Numbers** *Add the following error value, in the right sorted order:*

[EBADCMD] Inappropriate I/O control operation. A control function was attempted for a file or a special file for which the operation was inappropriate. This error is synonymous with [ENOTTY], but shall be used when control operations are inappropriate for a non-TTY device.

I.5 Functions

⇒ **22 Device Control** *Add a new chapter with the following new interface:*

The following function provides the device control capability:

posix_devctl()
Control a Device

I.5.1 Control a Device

Function: *posix_devctl*()

I.5.1.1 Synopsis

```
#include <sys/types.h>
#include <unistd.h>
#include <devctl.h>

int posix_devctl(int fildes,
                 int dcmd,
                 void *dev_data_ptr,
                 size_t nbyte,
                 int *dev_info_ptr);
```

I.5.1.2 Description

If the Device Control option is supported: B

The *posix_devctl*() function shall cause the device control command *dcmd* to be passed to the driver identified by *fildes*. Associated data shall be passed to and/or from the driver depending on direction information encoded in the *dcmd* argument, or as implied in the *dcmd* argument by the design and implementation of the driver.

The *dev_data_ptr* argument shall be a pointer to a buffer that contains data bytes to be passed to the driver and receives data bytes to be passed back from the driver or both.

If the data is to be passed to the driver, at least *nbyte* bytes of associated data shall be made available to the driver; if the data is to be passed from the driver, no more than *nbyte* bytes shall be passed.

If *nbyte* is zero, the amount of data passed to and/or from the driver is unspecified. This feature is obsolescent, and only provided for compatibility with existing device drivers.

The *dev_info_ptr* argument provides the opportunity to return an additional device information word instead of just a success/failure indication.

The set of valid commands, the associated data interpretation, the returned device information word , and the effect of the command on the device are all defined by the driver identified by *fildes*.

Otherwise:

Either the implementation shall support the *posix_devctl*() function as described above or this function shall not be provided. B

I.5.1.3 Returns

The *posix_devctl*() function shall return zero on success and the corresponding status value on failure. The value returned via the *dev_info_ptr* argument is driver dependent.

I.5.1.4 Errors

If any of the following conditions occur, the *posix_devctl*() function shall fail and shall return the corresponding error number:

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

B

If the following conditions are detected, the *posix_devctl*() function shall fail and shall return the corresponding error number:

[EBADCMD] The *dcmd* argument is not valid for this device.

[EINTR] The *posix_devctl*() function was interrupted by a signal.

[EINVAL] The *nbyte* argument is negative, or exceeds an implementation defined maximum, or is less than the minimum number of bytes required for this command.

The argument *dev_info_ptr* is an invalid address, or the argument *dev_data_ptr* is an invalid address, or the *dev_data_ptr* + *nbytes* - 1 is an invalid address.

[EPERM] The requesting process does not have sufficient privilege to request the device to perform the specified command.

Driver code may detect other errors, but the error numbers returned are driver dependent. *See* 21.4.9.

If the *posix_devctl*() function fails, the effect of this failed function on the device is driver dependent. Corresponding data might be transferred, partially transferred, or not transferred at all.

I.5.1.5 Cross-References

close(), 6.3.1; *dup*(), 6.2.1; *fstat*(), 5.6.2; *open*(), 5.3.1; *read*(), 6.4.1 *write*(), 6.4.2.

I.6 Rationale Relating to Device Control

An interface to be included in the POSIX standard should improve source code portability of application programs. In existing UNIX practice, *ioctl*() is used to handle special hardware. Therefore a general specification of its arguments cannot be written. Based on this fact, many people claim that *ioctl*() or something close to it has no place in POSIX.

Against this perception stands the widespread use of *ioctl*() to interface to all sorts of drivers for a vast variety of hardware used in all areas of real time and embedded computing, such as analog-digital converters, counters and video graphic devices. These devices provide a set of services that cannot be represented or used in terms of read or write system calls.

The arguments in favor of *ioctl*() standardization can be summarized as follows:

Even if *ioctl*() addresses very different hardware, many of these devices are either actually the same, interfaced to different computer systems with different implementations of operating systems, or belong to classes of devices with rather high commonality in their functions, e.g. analog-digital converters or digital-analog converters. Growing standardization of the Control and Status Register (CSR) space of these devices allows or will allow exploitation of a growing similarity of control codes and data for these. A general mechanism is needed to control these devices.

In all these cases a standardized interface from the application program to drivers for these devices will improve source code portability.

Even if control codes and device data have to be changed when porting applications from one system to another, the definition of *ioctl*() largely improves readability of a program handling special devices. Changes are confined to more clearly labeled places.

A driver for a specific device normally cannot be considered portable *per se*, but an application that uses this driver can be made portable if all interfaces needed are well defined and standardized. Users and integrators of real time systems often add device drivers for specific devices and a standard interface simplifies this process. Also, device drivers often follow their special hardware from system to system.

I.6.1 Existing Practice

The *ioctl*() interface is widely used. It has provided the generality mentioned above. This or a similar interface will build upon the current programming practice and existing code base, both at the application and device driver level.

Existing practice encodes into the second parameter information about data size and direction in some systems. An example of such an encoding is BSD's use of two bits of the command word as read/write bits. However, *ioctl*() has definite problems with the way that its sometimes optional 3rd parameter can be interpreted.

This is similar to the existing POSIX.1 *fcntl*() function, in which the 3rd parameter can be optional for `F_GETFD`, `F_GETFL`, an int *fildes* when used with the `F_DUPFD`,

F_SETFD, or F_SETFL commands or a struct *flock*, when used with the F_GETLK, F_SETLD or F_SETLKW commands. However, the *fcntl*() interface defines two distinct and known data types as possible for the 3rd parameter. This is not the case in the *ioctl*() interface, where many device driver specific structures and commands are used.

I.6.2 Relationship to *ioctl*() and the Perceived Needs for Improvement.

POSIX.1 documents, in Annex B.7, the perceived deficiencies in existing implementations of the *ioctl*() function. This discussion is in the context of those *ioctl*() commands used to implement terminal control. The POSIX.1 working group decided that, since the set of such control operations was fairly well defined, suitable encapsulations such as *tcsetattr*(), *tcsendbreak*(), and *tcdrain*() could be standardized. These interfaces, while successfully standardizing portable terminal control operations, are not extensible to arbitrary user-supplied devices. The *posix_devctl*() interface replaces the various *ioctl*() implementations with a standard interface which captures the extensibility of *ioctl*(), but avoids several of the deficiencies:

— The major problem with *ioctl*() is that the third argument is a generic pointer to a memory object which varies in both size and type according to the second *command* argument. It is not unprecedented in POSIX, or standards in general, for a function to accept a generic pointer; consider the ANSI C library functions *fgets*() and *fread*(), or the POSIX functions *read*() and *mmap*(). However, in all such instances, the generic pointer must be accompanied by a user-specified size argument which specifies the size of the pointed-to object. Unlike the Ada language, it is, and has always been, the C programmer's responsibility to ensure that these two arguments form a consistent specification of the passed object. But traditional *ioctl*() implementations do not allow the user to specify the size of the pointed-to object; that size is instead fixed implicitly by the specified command (passed as another argument). The *posix_devctl*() interface improves upon *ioctl*() in that it allows the user to specify the object size, thereby restoring the familiar C paradigm for passing a generic object by pointer/size pair.

— A secondary problem with *ioctl*() is that the third argument is sometimes permitted to be interpreted as an integer (int). This is non-portable to systems where *sizeof*(void *) != *sizeof*(int), not to mention a gross abuse of type casts. The *posix_devctl*() interface clearly requires the *dev_data_ptr* argument to be a pointer.

— A related problem with *ioctl*() is that the direction(s) in which data are transferred to/from the pointed-to object is neither specified explicitly as an argument (as with *mmap*()), nor implied by the *ioctl*() function (as with *read*()/ *write*(), *fread*()/ *fwrite*(), or *fgets*()/ *fputs*()). Instead, the direction is implied by the *command* argument. In traditional implementations, only the device driver knows the interpretation of the commands and whether data is to be transferred to or from the pointed-to object. But in networked implementations, generic portions of the operating system may need to know the direction to ensure that data are passed properly between a client

and a server, separately from device driver concerns. Two implementation-specific solutions to this problem are: a) to always assume data needs to be transferred in both directions; and b) to encode the implied direction into the command word along with the fixed data size. The *posix_devctl*() interface already provides the implementation with an explicit size parameter; since the direction is already known implicitly to both the application and the driver, and since workable methods exist for implementations to ascertain that direction if required, this perceived problem is strictly an implementation issue, and solvable without further impact on the interface.

— Finally, *posix_devctl*() improves upon *ioctl*() by adopting the new style of error return, avoiding all the problems *errno* brings to multi-threaded applications. Because the driver-specific information carried by the non-error return values of *ioctl*() still potentially needs to be passed to the application, *posix_devctl*() adds the *dev_info_ptr* argument to specify where this information should be stored.

I.6.3 Which Changes to *ioctl*() Are Acceptable?

Any change in the definition of *ioctl*() has to be perceived as a clear improvement by the community of people touched by this change. We have to be aware that drivers for *normal* peripherals are typically written by highly specialized professionals. Drivers for the *special devices* are very often written by the end-user or by the hardware designer, sometimes with fairly limited software literacy. Any interface definition which can be seen as overly complicated will simply not be accepted.

Nevertheless, a few simple and useful improvements to *ioctl*() are possible, justifying also the change of name from *ioctl*() to *devctl*().

The major change is the addition of the size of the device data. For enhanced compatibility with existing *ioctl*() implementations, this size may be specified as zero; in this case the amount of data passed is unspecified. (This allows a macro definition of *ioctl*() which converts it into a *posix_devctl*() call.)

The method of indicating error return values differs from traditional *ioctl*() implementations, but it does not preclude the construction of *posix_devctl*() as a macro built upon *ioctl*().

I.6.4 Rationale for the *dev_info_ptr*

The working group felt that it was important to preserve the current *ioctl*() functionality of allowing a device driver to return some arbitrary piece of information instead of just a success/failure indication. Such information might be, for example, the number of bytes received, the number of bytes that would not fit into the buffer pointed at by *dev_data_ptr*, the data type indication, or the device status. Current practice for device drivers and *ioctl*() usage allows such a device dependent return value. Thus the concept of an additional output argument, *dev_info_ptr*, was born.

I.6.5 Rationale for No *direction* Argument

The initial specification for *posix_devctl*() contained an additional argument which specified the direction of data flow — to the driver and/or from the driver. This argument was later removed for the following reasons:

— The argument was redundant. Most (if not all) existing implementations encode the direction data either explicitly or implicitly in the command word.

— The argument increased the probability of programming errors, since it must be made to agree with the direction information already encoded or implied in the command word or an error would occur.

— The only real use of the argument would be if new drivers were written which supported generic commands such as `TRANSFER_CONTROL_DATA`, which was modified by the direction argument to indicate in which direction the data should be transferred. This is contrary to current practice which uses command pairs such as `GET_CONTROL_DATA`, and `PUT_CONTROL_DATA`.

— The primary purpose of the direction argument was to allow higher levels of the system to identify the direction of data transfers, particularly in the case of remote devices, without having to understand all the commands of all the devices on the system. We believe that implementations which need to ascertain the direction of data transfer from a command word will define a consistent convention for encoding the direction into each command word, and all device drivers supplied by the user must adhere to this convention. A standard convention may be defined in the future when device driver interface standardization is undertaken.

Thus the data direction argument was removed.

I.6.6 Rationale for Not Defining the Direction Encoding in the *command* Word

Consideration was given to defining the direction encoding in the command word, but was rejected. No particular benefit was seen to a pre-defined encoding, as long as the encoding was used consistently across the entire implementation and was well known to the implementation.

In addition, although only one encoding (BSD's) was known among the members of the small working group, it could not be ruled out that other encodings already exist, and no reason for precluding these encodings was seen.

Finally, system or architectural constraints might make a chosen standard encoding difficult to use on a given implementation.

Thus, this standard does not define a direction encoding. Specifying a standard encoding is actually a small part of a larger and more contentious objective, that of specifying a complete set of interfaces for portable device drivers; if a future amendment to this standard specifies such interfaces, the issue of device control direction encoding will necessarily be addressed as part of that specification.

I.6.7 Recommended Practice for Handling Data Size Errors

In the event that the data from the device are too large to fit into the specified buffer, as much data as will fit should be transferred, and the error posted. The remaining data will aid in debugging.

I.6.8 Recommended Practice for *nbyte* == 0

The feature which permits an unspecified amount of control data to be transferred if *nbyte* is zero is obsolescent, and exits only for compatibility with existing device driver usage of *ioctl*(), i.e. the device driver always transfers an amount of data implied by the command. Newly developed device drivers should always honor the application's *nbyte* argument or return the error [EINVAL] if the argument is an unacceptable value. Such a device driver should interpret a zero value of *nbyte* as no data to be transferred.

I.6.9 Recommended Practice for Driver Detected Errors

If the driver detects the following error conditions, it is recommended that the *posix_devctl*() function fail and return the corresponding error number:

[EAGAIN] The control operation could not complete successfully because the device was in use by another process or the driver was unable to carry out the request due to an outstanding operation in progress.

[EBADCMD] The driver determined that the *dcmd* argument is not valid for this device or subdevice.

[EINVAL] The arguments *dev_dta_ptr* and *nbyte* define a buffer too small to hold the data expected by or to be returned by this driver.

[EIO] The control operation could not complete successfully because the driver detected a hardware error.