IEEE Standards

1003.26™

IEEE Standard for Information Technology— Portable Operating System Interface (POSIX™)— Part 26: Device Control Application Program Interface (API) [C Language]

IEEE Computer Society

Sponsored by the
Portable Applications Standards Committee

IEEE

3 Park Avenue, New York, NY 10016-5997, USA

9 September 2004

Print: SH95190
PDF: SS95190

IEEE Standard for Information Technology— Portable Operating System Interface (POSIX™)— Part 26: Device Control Application Program Interface (API) [C Language]

Sponsor
Portable Applications Standards Committee of the IEEE Computer Society

Approved 26 April 2004
American National Standards Institute

Approved 10 December 2003
Reaffirmed 17 June 2010
IEEE-SA Standards Board

Abstract: This standard is part of the POSIX series of standards. It defines an application program interface for controlling device drivers. Although it is based on the widely used *ioctl*() system call, the interface is type-safe and has a fixed number of parameters.
Keywords: API, application portability, C (programming language), data processing, information interchange, open systems, operating system, portable application, POSIX, programming language, realtime, device drivers

Abstract: This standard is part of the POSIX series of standards. It defines an application program interface for controlling device drivers. Although it is based on the widely used *ioctl*() system call, the interface is type-safe and has a fixed number of parameters.
Keywords: API, application portability, C (programming language), data processing, information interchange, open systems, operating system, portable application, POSIX, programming language, realtime, device drivers

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Print: ISBN 0-7381-3883-5 SH95190
PDF: ISBN 0-7381-3884-3 SS95190

Contents

Introduction

(This introduction is not a normative part of IEEE Std 1003.26™-2003, IEEE Standard for Information Technology—Portable Operating System Interface (POSIX™)—Part 26: Device Control Application Program Interface (API) [C Language].)

This standard defines systems interfaces to support the source portability of applications with realtime requirements and requiring in addition the control of special devices. The system interfaces are all extensions of, or additions to, ISO/IEC 9945:2003 {2} (identical to IEEE Std 1003.1™-2003).

Within this standard, the term "POSIX.1 {2}" refers to ISO/IEC 9945:2003 {2}.

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. Traditional practice uses a single function, called *ioctl*(), to encapsulate all the control operations on the different devices connected to the system, both special or common devices. Developers of POSIX.1 {2} decided not to standardize this interface because it was not type safe, it had a variable number of parameters, and it had behaviors that could not be specified by the standard because they were driver-dependent. Instead, POSIX.1 {2} defined a device-specific application program interface (API) for a common class of drivers, Terminals; and it restricted the *ioctl*() function to control of STREAMS devices.

Although the POSIX.1 solution for common classes of devices is the best from the point of view of application portability, there is still a need for a way to interact with special, or even common devices, for which developing a full standard API is not practical. This standard defines a general method for interfacing to the widest possible range of devices, through a new service to pass control information and commands between the application and the device drivers.

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 standard has been defined exclusively at the source code level, for the C programming language. Although the interfaces will be portable, some of the parameters used by an implementation may have hardware or configuration dependencies.

Organization of This Standard

This standard is divided into the following elements:

(1) Statement of scope and overview (1)

(2) List of normative references (2)

(3) Definitions and global concepts (3)

(4) Conventions and abbreviations (4)

(5) The various interface facilities (5)

(6) Informative annexes containing the bibliography and rationale

The C language binding for the service interface is given in the subclause labeled Synopsis (see 5.1.1.1). The Description subclause (see 5.1.1.2) provides a specification of the operation performed by the service interface. Some examples may be provided to illustrate the interfaces described.

In most cases, there are also Returns and Errors subclauses specifying return values and possible error conditions. References are used to direct the reader to other related sections. Additional material to complement sections in this standard may be found in Annex B. This annex provides historical perspectives into the technical choices made by the developers of this standard. It also provides information to emphasize consequences of the interfaces described in the corresponding section.

Informative annexes are not part of the standard and are provided for information only. They are provided for guidance and help in understanding. (There is a type of annex called *normative* that is part of a standard and imposes requirements, but there are no such normative annexes in this standard.)

Background

The developers of POSIX standards represent a cross section of hardware manufacturers, vendors of operating systems and other software development tools, software designers, consultants, academics, authors, applications programmers, and others. In the course of their deliberations, the developers reviewed related American and international standards, both published and in progress.

Conceptually, the POSIX base standards describe a set of fundamental services needed for the efficient construction of application programs. Access to these services has been provided by defining APIs, using the C programming language, that establish standard semantics and syntax. The goal of these interfaces is to enable application writers to write portable applications.

Although originated to refer to IEEE Std 1003.1, the name POSIX more correctly refers to a family of related standards; namely, the IEEE 1003 series and International Standard ISO/IEC 9945. This standard belongs to the POSIX family.

Audience

The intended audience for this standard is the same as for ISO/IEC 9945: all persons concerned with an industry-wide standard operating system based on the UNIX®[1] system. This includes at least four groups of people:

(1) Persons buying hardware and software systems,

(2) Persons managing companies that are deciding on future corporate computing directions,

(3) Persons implementing operating systems, and especially

(4) Persons developing applications where portability is an objective.

Purpose

Several principles guided the development of the POSIX standards in general and of this standard in particular.

Application-Oriented

The basic goal was to promote portability of application programs across UNIX system environments by developing a clear, consistent, and unambiguous standard for the interface specification of a portable operating system based on the UNIX system documentation.

Interface, Not Implementation

The POSIX standards define an interface, not an implementation. No distinction is made between library functions and system calls: both are referred to as *functions*. No details of the implementation of any function are given (although historical practice is sometimes indicated in Annex B). Symbolic names are given for constants (such as signals and error numbers) rather than numbers.

1) UNIX is a registered trademark of The Open Group.

Source, Not Object, Portability

The POSIX standards have been written so that a program written and translated for execution on one conforming implementation may also be translated for execution on another conforming implementation. In this standard, source code portability is limited to systems having the same special device drivers; but, if that is not the case, use of this standard will enhance portability by making the unportable parts that access special devices highly visible and uniformly used. This standard does not guarantee that executable (object or binary) code will execute under a conforming implementation other than that for which it was translated, even if the underlying hardware is identical.

The C Language

This standard is written in terms of the standard C language as specified in ISO/IEC 9899:1999 {1}.

No Superuser, No System Administration

There was no intention to specify all aspects of an operating system. System administration facilities and functions are excluded from the POSIX base standards, and functions usable only by the superuser have not been included. Still, an implementation of the standard interface may also implement features not in the POSIX standards. The POSIX standards are also not concerned with hardware constraints or system maintenance.

Minimal Interface, Minimally Defined

In keeping with the historical design principles of the UNIX system, the mandatory core facilities of the POSIX standards have been kept as minimal as possible. Additional capabilities were added as optional extensions or separate standards.

Broadly Implementable

The developers of the POSIX standards endeavored to make all specified functions implementable across a wide range of existing and potential systems, including the following:

(1) All of the current major systems that are ultimately derived from the original UNIX system code (Version 7 or later).

(2) Compatible systems that are not derived from the original UNIX system code.

(3) Emulations hosted on entirely different operating systems.

(4) Networked systems.

(5) Distributed systems.

(6) Systems running on a broad range of hardware.

Minimal Changes to Historical Implementations

There are no known historical implementations that will not have to change in some area to conform to this standard because, in the device control area, this standard does not exactly match any existing system interface. Nonetheless, POSIX standards specify sets of functions, types, definitions, and concepts that form an interface that is common to most historical implementations.

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

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

Minimal Changes to Existing Application Code

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

Realtime Extensions

POSIX.1 {2} defines optional sets of systems interfaces to support the source portability of applications with realtime requirements. The definition of realtime used in defining the scope of this standard is as follows:

> Realtime *in operating systems*: the ability of the operating system to provide a required level of service in a bounded response time.

Specifically within the scope of POSIX.1 {2} is to define interfaces that do not preclude high-performance implementations on traditional uniprocessor realtime systems.

This standard has the same goals as POSIX.1 {2} regarding the definition of interfaces that support the source portability of applications with real-time requirements.

Related Standards Activities

Activities to extend the POSIX family of standards to address additional requirements are in progress, and similar efforts can be anticipated in the future.

The following areas are under active consideration at this time or are expected to become active in the near future[2]:

(1) Additional system APIs in C language

(2) Ada language bindings to (1)

(3) Additional realtime facilities

(4) Profiles describing application- or user-specific combinations of open systems standards

Some extensions may be included in "revisions" to POSIX.1 {2}, following the IEEE and ISO/IEC procedures. Other extensions, such as this standard, are approved as independent standards.

If you have interest in participating in the Portable Application Standards Committee (PASC) working groups addressing these issues, please send your name, address, and telephone number to

Secretary, IEEE Standards Board
Institute of Electrical and Electronics Engineers, Inc.
445 Hoes Lane
Piscataway, NJ 08854
USA

When writing, ask to have your letter forwarded to the chair of the appropriate PASC working group.

If you have interest in participating in this work at the international level, contact your ISO/IEC national body.

2) A Standards Status Report that lists all current IEEE Computer Society standards projects is available from the IEEE Computer Society, 1730 Massachusetts Avenue NW, Washington, DC 20036-1903; Telephone: +1 202 371-0101; FAX: +1 202 728-9614. Working drafts of POSIX standards under development are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854 (http://www.standards.ieee.org/).

Notice to Users

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: http://standards.ieee.org/reading/ieee/updates/errata/index.html. Users are encouraged to check this URL for errata periodically.

Interpretations

Current interpretations can be accessed at the following URL: http:/standards.ieee.org/reading/ieee/interp/index.html.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents or patent applications for which a license may be required to implement an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Participants

This standard was prepared by the System Services Working Group—Realtime, sponsored by the Portable Application Standards Committee of the IEEE Computer Society. At the time this standard was approved, the membership of the System Services Working Group—Realtime was as follows:

IEEE System Services Working Group—Realtime

Chair:	Joseph M. Gwinn
Secretary:	Karen D. Gordon
Technical Editor:	Michael González
Ballot Coordinator:	Jim Oblinger
Technical Reviewer:	Franklin Prindle

Working Group

At the time this standard was completed, the IEEE System Services Working Group—Realtime had the following members:

Pierre-Jean Arcos	Chris Eck	François Riche
Ted Baker	Dave Emery	Doug Robinson
Bob Barned	Andrew Josey	Greg Shelton
Don Cragun	Jim Litchfield	Peter van der Veen
	Ray Richards	

Balloting Committee

The following members of the individual balloting committee voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

Pierre Jean Arcos	Charles Hammons	Vikram Punj
Mitchell Bonnett	Barry Hedquist	Francois Riche
Mark Brown	Karl Heubaum	John Riley
Paul Buerger	Daniel Hinz	Curtis Royster
Donald Cragun	Lowell G. Johnson	Diane Schleicher
John Davies	Andrew Josey	Stephen Schwarm
Juan Antonio de la Puente	Joerg Kampmann	Yuriy Sheynin
Sourav Dutta	Bil LaPlant	Gil Shultz
Yaacov Fenster	David Leciston	Keld Simonsen
Michel Gien	Roger J. Martin	Mark-Rene Uchida
Michael González	George Miao	Srinivasa Vemuru
Karen D. Gordon	James Oblinger	Rina Walach
Scott Gudgel	Peter Petrov	Oren Yuen
Joseph M. Gwinn	Franklin Prindle	Janusz Zalewski

IEEE Standard for Information Technology— Portable Operating System Interface (POSIX™)— Part 26: Device Control Application Program Interface (API) [C Language]

Section 1: Overview

1.1 Scope

This standard defines extensions to POSIX.1 {2} to support application portability at the source-code level. It is intended to be used by both application developers and system implementers.

The scope of this standard is to define a portable application interface for applications with realtime constraints requiring the ability to control special devices from the application itself.

The intent is to take existing realtime operating system practice and add it to the POSIX family of standards. Specifically within the scope is to define interfaces which do not preclude high-performance implementations on traditional uniprocessor realtime systems, as well as on multiprocessors. Wherever possible, the requirements of other application environments were included in the interface definition.

This standard has been defined exclusively at the source code level. Additionally, although the interfaces will be portable, some of the parameters used by an implementation may have hardware dependencies.

1.2 Conformance

1.2.1 Implementation Conformance

1.2.1.1 Requirements

A *conforming implementation* shall meet all of the following criteria:

(1) The implementation shall conform to POSIX.1 {2}[1)].

(2) The system shall support all required interfaces defined within this standard. All supported interfaces shall support the functional behavior described herein.

(3) The system may provide additional functions or facilities not required by (1) or (2) above. Nonstandard extensions of the functions or facilities specified in this standard should be identified as such in the system documentation. Nonstandard extensions, when used, may change the behavior of functions or facilities defined by this standard. The conformance document shall define an environment in which an application can be run with the behavior specified by the standard. In no case shall such an environment require modification of a Strictly Conforming POSIX.26 Application.

1.2.1.2 Documentation

A conformance document with the following information shall be available for an implementation claiming conformance to this standard. The conformance document shall have the structure of an annex to the required POSIX.1 conformance document. The conformance document shall not contain information about extended facilities or capabilities outside the scope of this standard.

1) Note that although POSIX.1 {2} conformance is required for POSIX.26 conformance, the POSIX profiles have the authority to specify the appropriate subset of the union of POSIX.1 {2} and POSIX.26 that is required for a particular application environment. Therefore, an implementation that supports all required POSIX.26 interfaces, but that cannot support all required POSIX.1 interfaces, should be designed to conform to one of the POSIX profiles rather than to POSIX.26.

The conformance document shall contain a statement that indicates the full name, number, and date of the standard that applies. The conformance document may also list international software standards that are available for use by a Conforming POSIX.26 Application. Applicable characteristics where documentation is required by one of these standards, or by standards of government bodies, may also be included.

The conformance document shall describe the limit values found in the `<limits.h>` and `<unistd.h>` headers, stating values, the conditions under which those values may change, and the limits of such variations, if any.

The conformance document shall describe the behavior of the implementation for all implementation-defined features defined in this standard. This requirement shall be met by listing these features and providing either a specific reference to the system documentation or providing full syntax and semantics of these features. When the value or behavior in the implementation is designed to be variable or customized on each instantiation of the system, the implementation provider shall document the nature and permissible ranges of this variation.

The conformance document may specify the behavior of the implementation for those features where this standard states that implementations may vary or where features are identified as undefined or unspecified.

The conformance document shall not contain documentation other than that specified in the preceding paragraphs, except where such documentation is specifically allowed or required by other provisions of this standard.

The phrases “shall document” or “shall be documented” in this standard mean that documentation of the feature shall appear in the conformance document, as described previously, unless there is a precise and explicit reference in the conformance document to show where the information can be found in the system documentation.

The system documentation should also contain the information found in the conformance document.

1.2.1.3 Conforming Implementation Options

No options are defined in this standard.

1.2.2 Application Conformance

All applications claiming conformance to this standard shall fall within one of the following categories.

1.2.2.1 Strictly Conforming POSIX.26 Application

A Strictly Conforming POSIX.26 Application is an application that is a Strictly Conforming POSIX application as defined in POSIX.1 {2}, except that it is allowed to use the interfaces specified in this standard with the constraints that such an application

(1) Shall accept any implementation behavior that results from actions it takes in areas described in this standard as implementation-defined or unspecified, or where this standard indicates that implementations may vary.

(2) Shall not perform any actions that are described as producing undefined results.

(3) For symbolic constants, shall accept any value in the range permitted by this standard, but shall not rely on any value in the range being greater than the minimums listed, or being less than the maximums listed, in this standard.

(4) Shall not use facilities designated as obsolescent.

(5) Is required to tolerate and permitted to adapt to the presence or absence of optional facilities defined in POSIX.1 {2}.

(6) For the C programming language, shall not produce any output dependent on any behavior described in ISO/IEC 9899:1999 {1} as unspecified, undefined, or implementation-defined, unless POSIX.1 {2} or this standard specifies the behavior.

(7) For the C programming language, shall not exceed any minimum implementation limit defined in ISO/IEC 9899:1999 {1}, unless POSIX.1 {2} or this standard specifies a higher minimum implementation limit.

(8) For the C programming language, shall define _POSIX_26_C_SOURCE to be 200312L before any header is included.

Within this standard, any restrictions placed upon a Conforming POSIX.26 Application shall restrict a Strictly Conforming POSIX.26 Application.

1.2.2.2 Conforming POSIX.26 Application

1.2.2.2.1 ISO/IEC Conforming POSIX.26 Application

An ISO/IEC Conforming POSIX.26 Application is an application that is a ISO/IEC Conforming POSIX Application as defined in POSIX.1 {2}, with this standard listed as one of the ISO/IEC standards it uses.

1.2.2.2.2 <National Body> Conforming POSIX.26 Application

A <National Body> Conforming POSIX.26 Application is an application that is a <National Body> Conforming POSIX Application as defined in POSIX.1 {2}, with this standard listed as one of the standards it uses.

1.2.2.3 Conforming POSIX.26 Application Using Extensions

A Conforming POSIX.26 Application Using Extensions is an application that is a Conforming POSIX Application Using Extensions as defined in POSIX.1 {2}, with this standard listed as one of the standards it uses.

1.2.3 Other Language-Related Specifications

This standard is currently specified in terms of ISO/IEC 9899:1999 {1}. Bindings to other programming languages are being developed.

If conformance to this standard is claimed for implementation of any programming language, the implementation of that language shall support the use of external symbols distinct to at least 31 bytes in length in the source program text. (In other words, identifiers that differ at or before the thirty-first byte shall be distinct.) If a national or international standard governing a language defines a maximum length that is less than this value, the language-defined maximum shall be supported. External symbols that differ only by case shall be distinct when the character set in use distinguishes uppercase and lowercase characters and the language permits (or requires) uppercase and lowercase characters to be distinct in external symbols.

This page is left intentionally blank.

Section 2: Normative References

2.1 Normative References

The following standards contain provisions which, through references in this text, constitute provisions of this standard[1]. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this profile of IEEE and ISO are encouraged to investigate the possibility of applying the most recent editions of the standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards.

{1} ISO/IEC 9899:1999, *Information processing systems—Programming languages—C*.[2]

{2} ISO/IEC 9945:2003, *Information Technology—Portable Operating System Interface* (*POSIX*™).[3]

{3} IEEE Std 610™-1990 (W), *IEEE Standard Computer Dictionary—A Compilation of IEEE Standard Computer Glossaries*.[4]

1) Common names for these standards can be found in 4.2. Other references to related standards and other documents can be found in Annex A of this standard.

2) ISO/IEC documents can be obtained from the ISO office, 1 rue de Varembé, Case Postale 56, CH-1211, Genève 20, Switzerland/Suisse (http://www.iso.ch/) and from the IEC office, 3 rue de Varembé, Case Postale 131, CH-1211, Genève 20, Switzerland/Suisse (http://www.iec.ch/). ISO/IEC publications are also available in the United States from the Sales Department, American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

3) Identical to IEEE Std 1003.1™-2003.

4) IEEE Std 610-1990 has been withdrawn; however, copies can be obtained from Global Engineering Documents, 15 Inverness Way East, Englewood, CO 80112-5704, USA, +1 303 792 2181 (http://global.ihs.com/).

This page is left intentionally blank.

Section 3: Definitions and General Concepts

The definitions and general concepts provided in POSIX.1 {2} apply in this standard unless modifications are specified in 3.1.

3.1 General Concepts

The following general concept is defined, in addition to those specified in the Base Definitions volume of POSIX.1 {2}, Section 4.

3.1.1 Special Device

In this standard, a *special device* is a device that requires control operations, other than the operations that are common to most devices [such as *read*(), *write*(), *open*(), and *close*()], but because the device belongs to a class that is not present in the majority of systems, standardization of a device-specific application program interface (API) for controlling it has not been practical. The driver for a special device may respond to the *write*() function to transfer data to the device or the *read*() function to collect information from the device. The interpretation of the information is defined by the implementor of the driver.

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. The means available to integrate drivers into the system and the way character special files are created are implementation defined. Character special files that have no structure defined by POSIX.1 {2} can be accessed as defined in Section 5.

3.1.2 Errors

The error numbers specified in the Errors subclauses of the system interfaces defined in this standard are defined in `<errno.h>` as specified by POSIX.1 {2}.

3.2 C Language Definitions

3.2.1 POSIX.26 Symbols

3.2.1.1 Version Test Macro

The following symbolic constant shall be defined in `<unistd.h>`:

Name	Description
_POSIX_26_VERSION	Integer value indicating the version of IEEE Std 1003.26™ (C-language binding) to which the implementation conforms. For implementations conforming to this standard, the value shall be 200312L.

3.2.1.2 The Compilation Environment

A POSIX.26 Conforming Application should ensure that the feature test macro _POSIX_26_C_SOURCE is defined before inclusion of any header.

When an application includes a header described by this standard and when this feature test macro is defined to have the value 200312L,

(1) All symbols required by this standard to appear when the header is included shall be made visible.

(2) Symbols explicitly permitted, but not required, by this standard 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 standard to be in that header shall not be made visible, except when enabled by another feature test macro.

3.2.1.3 System Configuration

The following symbolic constant shall be defined for *sysconf*() in `<unistd.h>`:

_SC_POSIX_26_VERSION

The implementation shall accept this constant for the name argument of the POSIX.1 *sysconf*() function. The associated variable to be returned for that constant shall be _POSIX_26_VERSION.

3.2.2 The Name Space

All identifiers in this standard are defined in the header file, as described in 3.2.3. When _POSIX_26_C_SOURCE is defined, this 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, as well as additional identifiers reserved for the implementation. In addition, the header may make visible identifiers from other headers as indicated in 3.2.3.

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 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 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_.

If the header in the following table is included, macros with the prefixes or suffixes shown may be defined.

Header	Prefix	Suffix
`<devctl.h>`	`POSIX_, _POSIX_, posix_`	`_t`

After the last inclusion of a given header, an application may use identifiers with the corresponding prefixes or suffixes for its own purpose, provided their use is preceded by a `#undef` of the corresponding macro.

3.2.3 Headers

Headers contain function prototypes, the definition of symbolic constants, common structures, preprocessor macros, and defined types. Each function in this standard specifies the headers that an application shall include in order to use that function. These headers are present on an application development system; they need not be present on the target execution system.

For functions defined in this standard, the prototypes or declarations shall appear in the headers listed below. The requirements about the visibility of symbols in 3.2.1 shall be honored.

`<devctl.h>`	*posix_devctl*()

The *size_t* type shall be defined as described in `<sys/types.h>`.

Inclusion of the `<devctl.h>` header may make visible all symbols from `<sys/types.h>`.

Section 4: Conventions and Abbreviations

4.1 Conventions

This standard uses the following typographic conventions:

(1) The *italic* font is used for

- Symbolic parameters that are generally substituted with real values by the application
- C language data types and function names (except in Synopsis subclauses)
- Global external variable names
- Function families; references to groups of closely related functions

(2) The **bold** font is used for the term "**NULL** pointer."

(3) The `constant-width` (Courier) font is used

- For C language data types and function names within function Synopsis subclauses
- To illustrate examples of system input or output where exact usage is depicted
- For references to utility names and C language headers
- For names of attributes in attributes objects

(4) Symbolic constants returned by many functions as error numbers are represented as

[ERRNO]

(5) Symbolic constants or limits defined in certain headers are represented as

OPEN_MAX

(6) Normative references listed in 2.1 are represented as

{1}

In some cases, tabular information is presented “inline”; in others it is presented in a separately labeled table. This arrangement was employed purely for ease of typesetting and there is no normative difference between these two cases.

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

Notes provided as parts of labeled tables and figures are integral parts of this standard (normative). Footnotes and notes within the body of the text are for information only (informative).

4.2 Abbreviations

For the purposes of this standard, the following abbreviations apply:

4.2.1 **POSIX.1:** ISO/IEC 9945:2003 {2}.

4.2.2 **POSIX.26:** *This standard.*

Section 5: Device Control

5.1 Functions

5.1.1 Control a Device

Function: *posix_devctl*()

5.1.1.1 Synopsis

```
#include <devctl.h>

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

5.1.1.2 Description

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.

If the *dev_data_ptr* argument is not a **NULL** pointer, it shall be a pointer to a buffer that is provided by the caller and that contains data bytes to be passed to the driver or provides space for receiving data bytes to be passed back from the driver, or both.

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

The driver may be executing in an address space different from the space of the calling thread. Therefore, if the data passed to the driver (i.e., the contents of the memory area starting at *dev_data_ptr* and continuing for *nbyte* bytes) contain pointers to data in the user's address space and the driver uses these pointers to access those data, the effects are unspecified.

If *dev_data_ptr* is not a **NULL** pointer and *nbyte* is zero, the amount of data passed to and/or from the driver is unspecified. This feature is obsolescent and is only provided for compatibility with existing device drivers.

If *dev_data_ptr* is a **NULL** pointer, there shall be no data passed between the caller and the driver other than the data specified in the rest of the arguments to *posix_devctl*() and in its return value.

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

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

The *posix_devctl*() function may be a cancellation point (see the System Interfaces volume of POSIX.1 {2}, Section 2.9.5).

The *posix_devctl*() function shall be reentrant to threads.

5.1.1.3 Return Value

Upon successful completion, *posix_devctl*() shall return zero; otherwise an error number shall be returned to indicate the error. The value returned via the *dev_info_ptr* argument is driver dependent.

5.1.1.4 Errors

The *posix_devctl*() function shall fail if

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

The *posix_devctl*() function may fail if

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

[ENOTTY]	The *fildes* argument is not associated with a character special file that accepts control functions.
[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 B.10.

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.

5.1.1.5 See Also

The System Interfaces volume of POSIX.1 {2} *close*(), *dup*(), *fstat*(), *ioctl*(), *open*(), *read*(), *write*().

This page is left intentionally blank.

Annex A
(informative)

Bibliography

This annex contains lists of related open systems standards and suggested reading on historical implementations and application programming.

A.1 Related Open Systems Standards

{B1} IEEE Std 1003.13™-2003, *IEEE Standard for Information Technology—Standardized Application Environment Profile—POSIX® Realtime Application Support (AEP).*1), 2)

{B2} IEEE Std 1003.5™-1999, *IEEE Standard for Information Technology—POSIX® Ada Language Interfaces—Part 1: Binding for System Application Programming Interface (API).*

{B3} ISO/IEC 8652:1995, *Information technology—Programming Languages—Ada.*3)

{B4} ISO 8859-1:1998, *Information technology—8-Bit Single-Byte Coded Graphic Character Sets—Part 1: Latin Alphabet No. 1.*

{B5} ISO/IEC 10646:2003, *Information technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 1: Architecture and Basic Multilingual Plane.*

1) IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).

2) The IEEE standards or products referred to in Annex A are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

3) ISO/IEC documents can be obtained from the ISO office, 1 rue de Varembé, Case Postale 56, CH-1211, Genève 20, Switzerland/Suisse (http://www.iso.ch/) and from the IEC office, 3 rue de Varembé, Case Postale 131, CH-1211, Genève 20, Switzerland/Suisse (http://www.iec.ch/). ISO/IEC publications are also available in the United States from the Sales Department, American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

A.2 Other Documents

{B6} *The Authorized Guide to the Single UNIX Specification,* Version 3, The Open Group, March 2002. UK ISBN: 1-85912-277-9. US ISBN 1-931624-13-5.[4)]

{B7} *Solaris 9 Reference Manual Collection—Man Pages, Section 2: System Calls*. Sun Microsystems Inc., May 2002.

{B8} University of California at Berkeley—Computer Science Research Group. 4.3 Berkeley Software Distribution, Virtual VAX-11 Version. Berkeley, CA: The Regents of the University of California, April 1986.

{B9} *X/Open CAE Specification System Interfaces and Headers,* Issue 4, Version 2. X/Open Company Ltd, September 1994. UK ISBN: 1-85912-037-3.

4) This publication is available from The Open Group at http://www.unix-systems.org/version3/theguide.html.

Annex B
(informative)

Rationale

B.1 Background

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 devices. Therefore, a general specification of its arguments cannot be written. Based on this fact, in the past many people claimed that *ioctl*(), or something close to it, had no place in the POSIX standards.

Against this perception stood the widespread use of *ioctl*() to interface to all sorts of drivers for a vast variety of hardware used in all areas of general-purpose, realtime, 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 either are 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 devices. 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 realtime 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.

In recognition of these reasons, The Open Group defined *ioctl*() in the X/Open System Interfaces and Headers {B9}, and the interface was later incorporated into POSIX.1 {2} under the XSI STREAMS option.

The *posix_devctl*() interface defined in this standard provides an alternative to the the various *ioctl*() implementations with a standard interface that captures the extensibility of *ioctl*(), but avoids several of its deficiencies, which is mentioned in B.3.

B.2 Existing Practice

The *ioctl*() interface is widely used. It has provided the generality mentioned in B.1. This *ioctl*() interface, 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 the use in BSD1) {B8} of two bits of the command word as read/write bits. However, *ioctl*() has definite problems with the way that its sometimes optional third parameter can be interpreted.

This practice is similar to the existing POSIX *fcntl*() function, in which the third 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 third parameter. This is not the case in the *ioctl*() interface, where any number of device driver specific structures and commands are used.

B.3 Relationship to *ioctl*() and the Perceived Needs for Improvement

The Rationale volume of POSIX.1 {2} briefly mentions some of the perceived deficiencies in existing implementations of the *ioctl*() function, in the context of those *ioctl*() commands used to implement terminal control. The POSIX.1 working group

1) BSD is a trademark of the University of California, Berkeley, CA, USA.

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. For that reason, POSIX.1 {2} defined the *ioctl*() function.

There are several perceived deficiencies with the *ioctl*() function that drove the development of the *posix_devctl*() interface as an alternative:

— The major problem with *ioctl*() is that the third argument is a generic pointer to a memory object that 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 that 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 nonportable 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 or 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 are 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 need 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 multithreaded applications. Because the driver-specific information carried by the nonerror 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.

B.4 Which Differences Between *posix_devctl*() and *ioctl*() Are Acceptable?

Any differences between the definitions of *posix_devctl*() and *ioctl*() have to be perceived as a clear improvement by the community of potential users. 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. Any interface definition that can be seen as overly complicated will simply not be accepted.

Nevertheless, a few simple and useful improvements to *ioctl*() are possible, specifically the improvement of type checking, and justify the definition of a new interface.

The major difference between the two interfaces 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*() that converts it into a *posix_devctl*() call.] In any case, the data size argument does not contradict the general goal of being able to implement *posix_devctl*() using the existing *ioctl*() interfaces provided in current UNIX systems and other POSIX implementations because the standard allows but does not require checking the size of the device data. Although the third argument of the *ioctl*() function does not specify a size, it is implicit in the specific combination of control command and driver and, therefore, known to the driver implementation.

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*(), which was one of the original design goals.

B.5 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.

B.6 Rationale for No *direction* Argument

The initial specification for *posix_devctl*() contained an additional argument that specified the direction of data flow, i.e., 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 that 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 that 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. Implementations that 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.

B.7 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 predefined 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.

B.8 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 retained data will aid in debugging, even if some data are lost.

B.9 Recommended Practice for *nbyte*== 0

The feature that permits an unspecified amount of control data to be transferred if *nbyte* is zero exists only for compatibility with existing device driver usage of *ioctl*(), i.e., when *ioctl*() is implemented on top of *posix_devctl*() and the device driver transfers an amount of data implied by the command.

Implementations in which *posix_devctl*() is built as a library routine on top of *ioctl*() may not be able to make checks on the *nbyte* argument. However, newly developed applications using *posix_devctl*() should always use an appropriate value for the *nbyte* argument, for portability to implementations directly supporting *posix_devctl*() in which the device drivers may be able to honor the application's *nbyte* argument or return the error [EINVAL] if the argument is an unacceptable value. Device drivers designed for those systems should interpret a zero value of *nbyte* as no data to be transferred.

B.10 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:

[EBUSY]	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.
[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.

This page is left intentionally blank.

Alphabetic Topical Index