

Introduction

1.1 Overview

This document describes the interfaces offered to application programs by the X/Open System Interface (XSI). It defines these interfaces and their run-time behaviour without imposing any particular restrictions on the way in which the interfaces are implemented.

The interfaces are defined in terms of the source code interfaces for the C programming language, which is defined in the ISO C standard. It is possible that some implementations may make the interfaces available to languages other than C, but this specification does not currently define the source code interfaces for any other language.

This specification allows an application to be built using a set of services that are consistent across all systems that conform to this specification (see Section 1.2). Such systems are termed XSI-conformant systems. Applications written in C using only these interfaces and avoiding implementation-dependent constructs are portable to all XSI-conformant systems.

This specification does not define networking interfaces; these are specified in the referenced **Networking Services, Issue 5** specification.

1.2 Conformance

An implementation conforming to this specification shall meet the requirements specified by BASE conformance (see Section 1.2.1).

1.2.1 BASE Conformance

An implementation conforming to this specification shall meet the following criteria for BASE conformance:

- The system shall support all the interfaces and headers defined within this specification that are part of the BASE capability. The BASE capability includes everything not listed in one of the Feature Groups defined in Section 1.3 on page 2.
- The system may provide one or more of the following Feature Groups:
 - Encryption
 - Realtime
 - Realtime Threads
 - Legacy.
- When an implementation claims that a feature is provided, all of its constituent parts shall be provided and shall comply with this specification.

Note: Whether support for a particular Feature Group is optional or mandatory is defined in the referenced **XPG4, Version 2** document. Some interfaces in Feature Groups define optional behaviour. To determine whether an implementation supports an optional Feature Group or optional behaviour, refer to the implementation's Conformance Statement.

- The system may provide additional or enhanced interfaces, headers and facilities not required by this specification, provided that such additions or enhancements do not affect the behaviour of an application that requires only the facilities described in this specification.

1.3 Feature Groups

For all Feature Groups, interfaces to all elements of the Feature Group shall exist. On implementations that do not support individual interfaces, each unsupported interface shall indicate an error, with *errno* set to [ENOSYS] unless otherwise specified.

If individual interfaces are supported, but the whole Feature Group is not supported, the interfaces will behave as defined in this specification.

1.3.1 Encryption

The Encryption Feature Group includes the following interfaces:

```
crypt()
encrypt()
setkey()
```

These are marked **CRYPT**.

Due to U.S. Government export restrictions on the decoding algorithm, implementations are restricted in making these functions available. All the functions in the Encryption Feature Group may therefore return [ENOSYS] or alternatively, *encrypt()* shall return [ENOSYS] for the decryption operation.

An implementation that claims conformance to this Feature Group shall set `_XOPEN_CRYPT` to a value other than `-1`. An implementation that does not claim conformance to this Feature Group shall set `_XOPEN_CRYPT` to `-1`.

1.3.2 Realtime

This document includes all the interfaces defined in the POSIX Realtime Extension.

Where entire manual pages have been added, they are marked **REALTIME**. Where additional semantics have been added to existing manual pages, the new material is identified by use of the RT margin legend.

An implementation that claims conformance to this Feature Group shall set the macro `_XOPEN_REALTIME` to a value other than `-1`. An implementation that does not claim conformance shall set `_XOPEN_REALTIME` to `-1`.

The POSIX Realtime Extension defines the following symbolic constants and their meaning:

`_POSIX_ASYNCIO`

Implementation supports the Asynchronous Input and Output option.

`_POSIX_FSYNC`

Implementation supports the File Synchronisation option. XSI-conformant systems always support the functionality associated with this symbol.

`_POSIX_MAPPED_FILES`

Implementation supports the Memory Mapped Files option. XSI-conformant systems always support the functionality associated with this symbol.

76	<code>_POSIX_MEMLOCK</code>	
77	Implementation supports the Process Memory Locking option.	
78	<code>_POSIX_MEMLOCK_RANGE</code>	
79	Implementation supports the Range Memory Locking option.	
80	<code>_POSIX_MEMORY_PROTECTION</code>	
81	Implementation supports the Memory Protection option. XSI-conformant systems always	
82	support the functionality associated with this symbol.	
83	<code>_POSIX_MESSAGE_PASSING</code>	
84	Implementation supports the Message Passing option.	
85	<code>_POSIX_PRIORITIZED_IO</code>	
86	Implementation supports the Prioritized Input and Output option.	
87	<code>_POSIX_PRIORITY_SCHEDULING</code>	
88	Implementation supports the Process Scheduling option.	
89	<code>_POSIX_REALTIME_SIGNALS</code>	
90	Implementation supports the Realtime Signals Extension option.	
91	<code>_POSIX_SEMAPHORES</code>	
92	Implementation supports the Semaphores option.	
93	<code>_POSIX_SHARED_MEMORY_OBJECTS</code>	
94	Implementation supports the Shared Memory Objects option.	
95	<code>_POSIX_SYNCHRONIZED_IO</code>	
96	Implementation supports the Synchronised Input and Output option.	
97	<code>_POSIX_TIMERS</code>	
98	Implementation supports the Timers option.	
99	If the symbol <code>_XOPEN_REALTIME</code> is defined to have a value other than <code>-1</code> , then the following	
100	symbolic constants will be defined to an unspecified value:	
101	<code>_POSIX_ASYNCHRONOUS_IO</code>	
102	<code>_POSIX_MEMLOCK</code>	
103	<code>_POSIX_MEMLOCK_RANGE</code>	
104	<code>_POSIX_MESSAGE_PASSING</code>	
105	<code>_POSIX_PRIORITY_SCHEDULING</code>	
106	<code>_POSIX_REALTIME_SIGNALS</code>	
107	<code>_POSIX_SEMAPHORES</code>	
108	<code>_POSIX_SHARED_MEMORY_OBJECTS</code>	
109	<code>_POSIX_SYNCHRONIZED_IO</code>	
110	<code>_POSIX_TIMERS</code>	
111	These are identified as ENHANCED I18N at the tops of applicable pages.	
112	Interfaces in the <code>_XOPEN_REALTIME</code> Feature Group are marked REALTIME .	
113	The functionality associated with <code>_POSIX_MAPPED_FILES</code> , <code>_POSIX_MEMORY_PROTECTION</code>	
114	and <code>_POSIX_FSYNC</code> is always present on XSI-conformant systems.	
115	Support of <code>_POSIX_PRIORITIZED_IO</code> is optional. If this functionality is supported, then	
116	<code>_POSIX_PRIORITIZED_IO</code> will be set to a value other than <code>-1</code> . Otherwise it will be undefined.	
117	If <code>_POSIX_PRIORITIZED_IO</code> is supported, then asynchronous I/O operations performed by	
118	<code>aio_read()</code> , <code>aio_write()</code> and <code>lio_listio()</code> will be submitted at a priority equal to the scheduling	
119	priority of the process minus <code>aiochp->aio_reqprio</code> . The implementation will also document for	

which files I/O prioritization is supported.

1.3.3 Realtime Threads

The Realtime Threads Feature Group includes the interfaces covered by the POSIX Threads compile-time symbolic constants `_POSIX_THREAD_PRIO_INHERIT`, `_POSIX_THREAD_PRIO_PROTECT` and `_POSIX_THREAD_PRIORITY_SCHEDULING` as defined in `<unistd.h>`. This includes the following interfaces:

```
pthread_attr_getinheritsched()
pthread_attr_getschedpolicy()
pthread_attr_getscope()
pthread_attr_setinheritsched()
pthread_attr_setschedpolicy()
pthread_attr_setscope()
pthread_getschedparam()
pthread_mutex_getprioceiling()
pthread_mutex_setprioceiling()
pthread_mutexattr_getprioceiling()
pthread_mutexattr_getprotocol()
pthread_mutexattr_setprioceiling()
pthread_mutexattr_setprotocol()
pthread_setschedparam()
```

Where applicable, pages are marked **REALTIME THREADS**, together with the RTT margin legend for the SYNOPSIS section.

An implementation that claims conformance to this Feature Group shall set `_XOPEN_REALTIME_THREADS` to a value other than `-1`. An implementation that does not claim conformance to this Feature Group shall set the value of `_XOPEN_REALTIME_THREADS` to `-1`.

If the symbol `_XOPEN_REALTIME_THREADS` is defined to have a value other than `-1`, then the symbols:

```
_POSIX_THREAD_PRIORITY_SCHEDULING
_POSIX_THREAD_PRIO_PROTECT
_POSIX_THREAD_PRIO_INHERIT
```

will also be defined; otherwise these symbols will be undefined.

1.3.4 Legacy

The Legacy Feature Group includes the interfaces and headers which were mandatory in previous versions of this specification but are optional in this version of the specification.

These interfaces and headers are retained in this specification because of their widespread use. Application writers should not rely on the existence of these interfaces or headers in new applications, but should follow the migration path detailed in the APPLICATION USAGE sections of the relevant pages.

Various factors may have contributed to the decision to mark an interface or header **LEGACY**. In all cases, the specific reasons for the withdrawal of an interface or header are documented on the relevant pages.

Once an interface or header is marked **LEGACY**, no modifications will be made to the specifications of such interfaces or headers other than to the APPLICATION USAGE sections of the relevant pages.

The interfaces and headers which form this Feature Group are as follows:

Legacy Interfaces, Headers and External Variables				
<i>advance()</i>	<i>gamma()</i>	<i>putw()</i>	<i>sbrk()</i>	<i>wait3()</i>
<i>brk()</i>	<i>getdtablesize()</i>	<i>re_comp()</i>	<i>sigstack()</i>	
<i>chroot()</i>	<i>getpagesize()</i>	<i>re_exec()</i>	<i>step()</i>	
<i>compile()</i>	<i>getpass()</i>	<i>regcmp()</i>	<i>ttyslot()</i>	
<i>cuserid()</i>	<i>getw()</i>	<i>regex()</i>	<i>valloc()</i>	
<regex.h>	<varargs.h>	<re_comp.h>		
<i>loc1</i>	<i>__loc1</i>	<i>loc2</i>	<i>locs</i>	

An implementation that claims conformance to this Feature Group shall set the macro `_XOPEN_LEGACY` to a value other than `-1`. An implementation that does not claim conformance shall set `_XOPEN_LEGACY` to `-1`.

1.4 Changes from Issue 4

The following sections describe changes made to this specification 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 summarises the major changes that were made in this specification from Issue 4 to Issue 4, Version 2:

- The X/Open UNIX extension has been added. This specifies the common core APIs of 4.3 Berkeley Software Distribution (BSD 4.3), the OSF AES and SVID Issue 3.
- STREAMS have been added as part of the X/Open UNIX extension.
- Existing XPG4 interfaces have been clarified as a result of industry feedback.

1.4.2 Changes from Issue 4, Version 2 to Issue 5

The following list summarises the major changes that have been made in this specification since Issue 4, version 2:

- Interfaces previously defined in the ISO POSIX-2 standard C-language Binding, Shared Memory, Enhanced Internationalisation and X/Open UNIX Extension Feature Groups are moved to the BASE in this issue.
- Threads are added to the BASE for alignment with the POSIX Threads Extension.
- The Realtime Threads Feature Group is added.
- The Realtime Feature Group is added for alignment with the POSIX Realtime Extension.
- Multibyte Support Extensions (MSE) are added to the BASE for alignment with ISO/IEC 9899:1990/Amendment 1:1994 (E).
- Large File Summit (LFS) Extensions are added to the BASE for support of 64-bit or larger files and file-systems.
- X/Open-specific Threads extensions are added to the BASE.
- X/Open-specific dynamic linking interfaces are added to the BASE.
- A new category Legacy has been added; see Section 1.3.4 on page 4.
- The categories TO BE WITHDRAWN and WITHDRAWN have been removed.

1.4.3 New Features

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

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

256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277

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

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

New Interfaces, 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()	fntmsg()	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()	setpggrp()	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 Terminology

The following terms are used in this specification:

can

This describes a permissible optional feature or behaviour available to the user or application; all systems support such features or behaviour as mandatory requirements.

implementation-dependent

The value or behaviour is not consistent across all implementations. The provider of an implementation normally documents the requirements for correct program construction and correct data in the use of that value or behaviour. When the value or behaviour in the implementation is designed to be variable or customisable on each instantiation of the system, the provider of the implementation normally documents the nature and permissible ranges of this variation. Applications that are intended to be portable must not rely on implementation-dependent values or behaviour.

legacy

Certain features are *legacy*, which means that they are being retained for compatibility with older applications, but have limitations which make them inappropriate for developing portable applications. New applications should use alternative means of obtaining equivalent functionality. Legacy features are marked **LEGACY**.

may

With respect to implementations, the feature or behaviour is optional. Applications should not rely on the existence of the feature. To avoid ambiguity, the reverse sense of *may* is expressed as *need not*, instead of *may not*.

must

This describes a requirement on the application or user.

should

With respect to implementations, the feature is recommended, but it is not mandatory. Applications should not rely on the existence of the feature.

With respect to users or applications, the word means recommended programming practice that is necessary for maximum portability.

undefined

A value or behaviour is undefined if this document imposes no portability requirements on applications for erroneous program constructs or erroneous data. Implementations may specify the result of using that value or causing that behaviour, but such specifications are not guaranteed to be consistent across all implementations. An application using such behaviour is not fully portable to all systems.

unspecified

A value or behaviour is unspecified if this document imposes no portability requirements on applications for correct program construct or correct data. Implementations may specify the result of using that value or causing that behaviour, but such specifications are not guaranteed to be consistent across all implementations. An application requiring a specific behaviour, rather than tolerating any behaviour when using that functionality, is not fully portable to all systems.

will

This means that the behaviour described is a requirement on the implementation and applications can rely on its existence.

1.6 Relationship to Formal Standards

Great care has been taken to ensure that this specification is fully aligned with the following formal standards:

- ISO/IEC 9945-1: 1996
- ISO/IEC 9945-2: 1993
- ISO/IEC 9899: 1990
- ISO/IEC 9899:1990/Amendment 1:1994 (E) (MSE)
- Federal Information Procurement Standards (FIPS) 151-2.

Any conflict between this specification and any of these standards is unintentional. This document defers to the formal standards, which The Open Group recognises as superior. In particular, from time to time, when ambiguities are found in the formal standards, the responsible bodies will make interpretations of them, whose findings become binding on the standard. Where, as the result of such an interpretation, or for any other reason, any of these formal standards are found to conflict with this specification, XSI-conformant systems are required to behave in the manner defined either by the formal standard or by this specification. Application writers should clearly avoid depending exclusively on either behaviour in such cases; the list of all conflicts found since publication of this specification is available on request. (See page ii for how to contact The Open Group.)

This document also allows, but does not require, mathematics functions to support IEEE Std 754-1985 and IEEE Std 854-1987.

1.6.1 Relationship to Emerging Formal Standards

This document also allows, but does not require, mathematics functions to behave as specified by the IEEE Floating Point draft report of ANSI X3J11.1 (NCEG).

Where function specifications in the draft ANSI X3J11.1 require behaviour that is different from this specification, but not in conflict with the ISO C standard, an XSI-conformant system may behave either in the manner defined by the draft ANSI X3J11.1 or by this specification.

1.7 Portability

This document describes a superset of the requirements of the ISO POSIX-1 standard and the ISO C standard. It also contains parts of the ISO POSIX-2 standard **Shell and Utilities** which The Open Group feels are better suited to inclusion in this specification, rather than in the XCU specification. (The ISO POSIX-1 standard is identical to IEEE Std 1003.1-1996, which is often referred to as the POSIX.1 standard. The ISO C standard is technically identical in normative content to the ANSI C standard.)

Some of the utilities in CAE Specification, **Commands and Utilities, Issue 5** and functions in this document describe functionality that might not be fully portable to systems based on the ISO POSIX-1 or ISO POSIX-2 standards. Where enhanced or reduced functionality is specified, the text is shaded and a code in the margin identifies the nature of the extension or warning (see Section 1.7.1). For maximum portability, an application should avoid such functionality.

1.7.1 Codes

The codes and their meanings are as follows:

EX Extension.

The functionality described is an extension to the standards referenced above. Application writers may confidently make use of an extension as it will be supported on all XSI-conformant systems. These extensions are designed not to conflict with the published standards.

If an entire **SYNOPSIS** section is shaded and marked with one EX, all the functionality described in that entry is an extension.

Some behaviour which is allowed to be optional in the formal standards is mandated on XSI-conformant systems. Such behaviours (for example, those dependent on the availability of job control) might not be individually marked as extensions, but the mandatory nature of the feature is marked as an extension where the option is described, typically in the header where the corresponding symbolic constant is defined.

FIPS FIPS Requirements.

The **Federal Information Processing Standards (FIPS)** are a series of U.S. government procurement standards managed and maintained on behalf of the U.S. Department of Commerce by the National Institute of Standards and Technology (NIST). Where restrictions have been made in order to align with the FIPS requirements, they have the special mark shown here, and appear in the index under FIPS alignment (as well as under EX).

The following restrictions are required by FIPS 151-2:

- The implementation will support {_POSIX_CHOWN_RESTRICTED}.
- The limit {NGROUPS_MAX} will be greater than or equal to 8.
- The implementation will support the setting of the group ID of a file (when it is created) to that of the parent directory.
- The implementation will support {_POSIX_SAVED_IDS}.
- The implementation will support {_POSIX_VDISABLE}.
- The implementation will support {_POSIX_JOB_CONTROL}.
- The implementation will support {_POSIX_NO_TRUNC}.
- The *read()* call returns the number of bytes read when interrupted by a signal and will not return -1.

- 431 • The `write()` call returns the number of bytes written when interrupted by a signal and will
432 not return `-1`.
- 433 • In the environment for the login shell, the environment variables `LOGNAME` and `HOME` will
434 be defined and have the properties described in Chapter 5 of CAE Specification, **System**
435 **Interface Definitions, Issue 5**.
- 436 • The value of `{CHILD_MAX}` will be greater than or equal to 25.
- 437 • The value of `{OPEN_MAX}` will be greater than or equal to 20.
- 438 • The implementation will support the functionality associated with the symbols `CS7`, `CS8`,
439 `CSTOPB`, `PARODD` and `PARENB` defined in `<termios.h>`.
- 440 OH **Optional header.**
441 In the **SYNOPSIS** section of some interfaces in this document an included header is marked as
442 in the following example:
- 443 OH `#include <sys/types.h>`
444 `#include <grp.h>`
445 `struct group *getgrnam(const char *name);`
- 446 This indicates that the marked header is not required on XSI-conformant systems. This is an
447 extension to certain formal standards where the full synopsis is required.
- 448 RT **Realtime.**
449 This identifies the interfaces and additional semantics in the Realtime Feature Group.
- 450 RTT **Realtime Threads.**
451 This identifies the interfaces and additional semantics in the Realtime Threads Feature Group.

1.8 Format of Entries

The entries in Chapter 3 and Chapter 4 are based on a common format.

NAME

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

SYNOPSIS

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

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

DESCRIPTION

This section describes the functionality of the interface 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 will be 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 will be 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 gives examples of usage, where appropriate. This section is non-normative. In the event of conflict between an example and a normative part of the specification, the normative material is to be taken as correct.

APPLICATION USAGE

This section gives warnings and advice to application writers about the entry. This section is non-normative. In the event of conflict between warnings and advice and a normative part of the specification, the normative material is to be taken as correct.

FUTURE DIRECTIONS

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

SEE ALSO

This section gives references to related information.

CHANGE HISTORY

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

The only sections relating to conformance are the SYNOPSIS, DESCRIPTION, RETURN VALUE and ERRORS sections.

General Information

This chapter covers information that is relevant to all the Interfaces specified in Chapter 3 and Chapter 4:

- the use and implementation of interfaces (see Section 2.1)
- the compilation environment (see Section 2.2 on page 17)
- error numbers (see Section 2.3 on page 22)
- standard I/O streams (see Section 2.4 on page 30)
- STREAMS (see Section 2.5 on page 34)
- interprocess communication (IPC) (see Section 2.6 on page 36)
- realtime (see Section 2.7 on page 38)
- threads (see Section 2.8 on page 46)
- data types (see Section 2.9 on page 55).

2.1 Use and Implementation of Interfaces

Each of the following statements applies unless explicitly stated otherwise in the detailed descriptions that follow. 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 behaviour is undefined. 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 will also ensure that an actual function is referred to. Any invocation of a library function that is implemented as a macro will 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.

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. If a function that accepts a variable number of arguments is not declared (explicitly or by including its associated header), the behaviour is undefined.

As a result of changes in this issue of this specification, application writers are only required to include the minimum number of headers. Implementations of XSI-conformant systems will make all necessary symbols visible as described in the Headers section of this specification.

531 2.1.1 Use of File System Interfaces

532 The Interfaces in this volume that operate on files can behave differently if the file that is being
533 operated on has been made available by a network file system. If the network file system is an
534 XSI-conformant system conforming to the **XNFS** specification, the differences that can occur are
535 detailed in Appendices A and B of that document.

2.2 The Compilation Environment

Applications should ensure that the feature test macro `_XOPEN_SOURCE` is defined with the value 500 before inclusion of any header. This is needed to enable the functionality described in this specification, and possibly to enable functionality defined elsewhere in the Common Applications Environment.

Identifiers in this specification may only be undefined using the `#undef` directive as described in Section 2.1 on page 15 or Section 2.2.1. These `#undef` directives must follow all `#include` directives of any XSI headers.

Most strictly conforming POSIX and ISO C applications will compile on systems compliant to this specification. However, an application which uses any of the items marked as an extension to POSIX and ISO C, for any purpose other than that shown here, will not necessarily compile. In such cases, it may be necessary to alter those applications to use alternative identifiers.

Since this specification 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 199506L should be enabled by `_XOPEN_SOURCE` set equal to 500, 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 500 and `_POSIX_SOURCE` is defined, or `_POSIX_C_SOURCE` is set greater than zero and less than or equal to 199506L, the behaviour is the same as if only `_XOPEN_SOURCE` is defined and set equal to 500. However, should `_POSIX_C_SOURCE` be set to a value greater than 199506L, the behaviour is undefined.

2.2.1 The X/Open Name Space

All identifiers in this specification except *environ* are defined in at least one of the headers, as shown in Chapter 4. When `_XOPEN_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.

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.

If any header in the following table is included, identifiers with the prefixes, suffixes or complete names shown are reserved for any use by the implementation.

Header	Prefix	Suffix	Complete Name
<aio.h>	aio_, lio_, AIO_, LIO_		
<dirent.h>	d_		
<errno.h>	E		
<fcntl.h>	l_		
<glob.h>	gl_		
<grp.h>	gr_		
<limits.h>		_MAX	
<locale.h>	LC_[A-Z]		
<mqueue.h>	mq_, MQ_		
<ndbm.h>	dbm_		
<poll.h>	pd_, ph_, ps_		
<pthread.h>	pthread_, PTHREAD_		
<pwd.h>	pw_		
<regex.h>	re_, rm_		
<sched.h>	sched_, SCHED_		
<semaphore.h>	sem_, SEM_		
<signal.h>	sa_, SIG[A-Z], SIG_[A-Z]		
	ss_, sv_		
<stropts.h>	si_, SI_, sigev_, SIGEV_, sival_		
<sys/ipc.h>	bi_, ic_, l_, sl_, str_		key, pad, seq
<sys/mman.h>	ipc_		
<sys/msg.h>	shm_, MAP_, MCL_, MS_, PROT_		msg
<sys/resource.h>	msg		
<sys/sem.h>	rlim_, ru_		sem
<sys/shm.h>	sem		
<sys/stat.h>	shm		
<sys/statvfs.h>	st_		
<sys/time.h>	f_		
<sys/times.h>	fds_, it_, tv_, FD_		
<sys/times.h>	tms_		
<sys/uio.h>	io_		
<sys/utsname.h>	uts_		
<sys/wait.h>	si_, W[A-Z], P_		
<termios.h>	c_		
<time.h>	tm_		
	clock_, timer_, it_, tv_,		
	CLOCK_, TIMER_		
<ucontext.h>	uc_		
<ulimit.h>	UL_		
<utime.h>	utim_		
<utmpx.h>	ut_	_LVL, _TIME, _PROCESS	
<wordexp.h>	we_		
ANY header		_t	

Note: The notation [A-Z] indicates any upper-case letter in the portable character set. The notation [a-z] indicates any lower-case 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				
<fcntl.h>	F_	O_	S_		
<fmtmsg.h>	MM_				
<fnmatch.h>	FNM_				
<ftw.h>	FTW				
<glob.h>	GLOB_				
<ndbm.h>	DBM_				
<nl_types.h>	NL_				
<poll.h>	POLL				
<re_comp.h>	REG_				
<regex.h>	REG_				
<signal.h>	SA_	SIG_[0-9a-z_]			
<stropts.h>	BUS_ SEGV_ FLUSH[A-Z] S_	CLD_ SI_ I_ SND[A-Z]	FPE_ SS_ M_ STR	ILL_ SV_ MUXID_R[A-Z]	POLL_ TRAP_
<syslog.h>	LOG_				
<sys/ipc.h>	IPC_				
<sys/mman.h>	PROT_	MAP_	MS_		
<sys/msg.h>	MSG[A-Z]	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_	MSG_	PF_	SO	
<sys/stat.h>	S_				
<sys/statvfs.h>	ST_				
<sys/time.h>	FD_	ITIMER_			
<sys/uio.h>	IOV_				
<sys/wait.h>	BUS_ SEGV_	CLD_ SI_	FPE_ TRAP_	ILL_	POLL_
<termios.h>	V	I	O	TC	B[0-9]
<wordexp.h>	WRDE_				

Note: The notation [0-9] indicates any digit. The notation [A-Z] indicates any upper-case letter in the portable character set. The notation [0-9a-z_] indicates any digit, any lower-case letter in the portable character set or underscore.

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

1. All identifiers that begin with an underscore and either an upper-case 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 specification, but are reserved for future use by the ISO C standard.

abort	cosl	fputwc	log	raise	tanhf
abs	ctime	fputws	log10	rand	tanh
acos	difftime	fread	log10f	realloc	tanl
acosh	div	free	log10l	remove	time
acosl	errno	freopen	logf	rename	tmpfile
asctime	exit	frexp	logl	rewind	tmpnam
asin	exp	frexpf	longjmp	scanf	to[a-z]*
asinh	expf	frexpl	malloc	setbuf	ungetc
asinl	expl	fscanf	mblen	setjmp	ungetwc
atan	fabs	fseek	mbrlen	setlocale	va_end
atan2	fabsf	fsetpos	mbrtowc	setvbuf	vfprintf
atan2f	fabsl	ftell	mbsinit	signal	vfwprintf
atan2l	fclose	fwide	mbsrtowcs	sin	vprintf
atanf	feof	fwprintf	mbstowcs	sinf	vsprintf
atanl	ferror	fwrite	mbtowc	sinh	vswprintf
atexit	fflush	fwscanf	mem[a-z]*	sinh	vwprintf
atof	fgetc	getc	mktime	sinhl	wcrtomb
atoi	fgetpos	getchar	modf	sinl	wcs[a-z]*
atol	fgets	getenv	modff	sprintf	wctob
bsearch	fgetwc	gets	modfl	sqrt	wctomb
calloc	fgetws	getwc	perror	sqrtf	wctrans
ceil	floor	getwchar	pow	sqrtl	wctype
ceilf	floorf	gmtime	powf	srand	wcwidth
ceil	floorl	is[a-z]*	powl	sscanf	wmem[a-z]*
clearerr	fmod	labs	printf	str[a-z]*	wprintf
clock	fmodf	ldexp	putc	swprintf	wscanf
cos	fmodl	ldexpf	putchar	swscanf	
cosf	fopen	ldexpl	puts	system	
cosh	fprintf	ldiv	putwc	tan	
coshf	fputc	localeconv	putwchar	tanf	
coshl	fputs	localtime	qsort	tanh	

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

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

_longjmp	endgrent	getmsg	lockf	realpath	sigpause
_setjmp	endpwent	getpagesize	log1p	regcmp	sigrelse
a64l	endservent	getpgid	logb	regex	sigset
acosh	endutxent	getpmsg	lstat	remainder	sigstack
asinh	expm1	getpriority	makecontext	remque	srandom
atanh	fattach	getpwent	mknod	rindex	statvfs
basename	fchdir	getrlimit	mkstemp	rint	strcasecmp
bcmp	fchmod	getrusage	mktemp	sbrk	strdup
bcopy	fchown	getsid	mmap	scalb	strncasecmp
brk	fcvt	getsubopt	mprotect	select	swapcontext
bsd_signal	fdetach	gettimeofday	msync	setcontext	symlink
bzero	ffs	getutxent	munmap	setgrent	sync
cbrt	fmtmsg	getutxid	nextafter	setitimer	syslog
closelog	fstatvfs	getutxline	nftw	setlogmask	tcgetsid
dbm_clearerr	ftime	getwd	openlog	setpgrp	truncate
dbm_close	ftok	grantpt	poll	setpriority	ttyslot
dbm_delete	ftruncate	ilogb	ptsname	setpwent	ualarm
dbm_error	gcvt	index	putmsg	setreuid	unlockpt
dbm_fetch	getcontext	initstate	putpmsg	setrlimit	usleep
dbm_firstkey	getdate	insque	pututxline	setstate	utimes
dbm_nextkey	getdtablesize	ioctl	random	setutxent	valloc
dbm_open	getgrent	isastream	re_comp	sigaltstack	vfork
dbm_store	getgrgid	killpg	re_exec	sighold	wait3
dirname	gethostid	l64a	readlink	sigignore	waitid
ecvt	getitimer	lchown	readv	siginterrupt	writev

All the identifiers defined in this specification that have external linkage are always reserved for use as identifiers with external linkage.

No other identifiers are reserved.

Applications must 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 must not be defined 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, a header must be included outside of any external declaration or definition, and it must 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 program must 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 header `<errno.h>`, 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 specification sets *errno* to zero to indicate an error. For each thread of a process, the value of *errno* is not 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 prevent some errors from occurring. The ERRORS section on each page specifies whether an error will be returned, or whether it may be returned. Implementations will not generate a different error number from the ones described here for error conditions described in this specification, but may generate additional errors unless explicitly disallowed for a particular function.

The following symbolic names identify the possible error numbers, in the context of the functions specifically defined in this specification; 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 are unique except as noted below. The values for all these names can be found in the header `<errno.h>`.

[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.

[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

795		This is a temporary condition and later calls to the same routine may complete normally.	
796	EX	[EALREADY]	
797		Connection already in progress	
798		A connection request is already in progress for the specified socket.	
799		[EBADF]	
800		Bad file descriptor	
801		A file descriptor argument is out of range, refers to no open file, or a read (write) request is	
802		made to a file that is only open for writing (reading).	
803	EX	[EBADMSG]	
804		Bad message	
805		During a <i>read()</i> , <i>getmsg()</i> or <i>ioctl()</i> I_RECVFD request to a STREAMS device, a message	
806		arrived at the head of the STREAM that is inappropriate for the function receiving the	
807		message.	
808		• <i>read()</i> — message waiting to be read on a STREAM is not a data message.	
809		• <i>getmsg()</i> — a file descriptor was received instead of a control message.	
810		• <i>ioctl()</i> — control or data information was received instead of a file descriptor when	
811		I_RECVFD was specified.	
812	RT	[EBADMSG]	
813		Bad Message	
814		The implementation has detected a corrupted message.	
815		[EBUSY]	
816		Resource busy	
817		An attempt was made to make use of a system resource that is not currently available, as it	
818		is being used by another process in a manner that would have conflicted with the request	
819		being made by this process.	
820	RT	[ECANCELED]	
821		Operation canceled	
822		The associated asynchronous operation was canceled before completion.	
823		[ECHILD]	
824		No child process	
825		A <i>wait()</i> or <i>waitpid()</i> function was executed by a process that had no existing or unwaited-	
826		for child process.	
827	EX	[ECONNABORTED]	
828		Connection aborted	
829		The connection has been aborted.	
830	EX	[ECONNREFUSED]	
831		Connection refused	
832		An attempt to connect to a socket was refused because there was no process listening or	
833		because the queue of connection requests was full and the underlying protocol does not	
834		support retransmissions.	
835	EX	[ECONNRESET]	
836		Connection reset	
837		The connection was forcibly closed by the peer.	
838		[EDEADLK]	
839		Resource deadlock would occur	

840		An attempt was made to lock a system resource that would have resulted in a deadlock situation.	
841			
842	EX	[EDESTADDRREQ]	
843		Destination address required	
844		No bind address was established.	
845		[EDOM]	
846		Domain error	
847		An input argument is outside the defined domain of the mathematical function. (Defined in the ISO C standard.)	
848			
849	EX	[EDQUOT]	
850		Reserved	
851		[EEXIST]	
852		File exists	
853		An existing file was mentioned in an inappropriate context, for instance, as a new link name in the <i>link()</i> function.	
854			
855		[EFAULT]	
856		Bad address	
857		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.	
858			
859			
860	EX	[EFBIG]	
861		File too large	
862		The size of a file would exceed the maximum file size of an implementation or offset maximum established in the corresponding file description.	
863			
864	EX	[EHOSTUNREACH]	
865		Host is unreachable	
866		The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).	
867			
868	EX	[EIDRM]	
869		Identifier removed	
870		Returned during interprocess communication if an identifier has been removed from the system.	
871			
872	RT	[EINPROGRESS]	
873		Operation in progress	
874		This code is used to indicate that an asynchronous operation has not yet completed.	
875	EX	[EINPROGRESS]	
876		O_NONBLOCK is set for the socket file descriptor and the connection cannot be immediately established.	
877			
878		[EILSEQ]	
879		Illegal byte sequence	
880		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.	
881			
882		[EINTR]	
883		Interrupted function call	
884		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	
885			

886		return this condition. (See <signal.h>.)	
887		[EINVAL]	
888		Invalid argument	
889		Some invalid argument was supplied; (for example, specifying an undefined signal in a	
890		<i>signal()</i> function or a <i>kill()</i> function).	
891		[EIO]	
892		Input/output error	
893		Some physical input or output error has occurred. This error may be reported on a	
894		subsequent operation on the same file descriptor. Any other error-causing operation on the	
895		same file descriptor may cause the [EIO] error indication to be lost.	
896	EX	[EISCONN]	
897		Socket is connected	
898		The specified socket is already connected.	
899		[EISDIR]	
900		Is a directory	
901		An attempt was made to open a directory with write mode specified.	
902	EX	[ELOOP]	
903		Too many levels of symbolic links	
904		Too many symbolic links were encountered in resolving a pathname.	
905		[EMFILE]	
906		Too many open files	
907		An attempt was made to open more than the maximum number of {OPEN_MAX} file	
908		descriptors allowed in this process.	
909		[EMLINK]	
910		Too many links	
911		An attempt was made to have the link count of a single file exceed {LINK_MAX}.	
912	EX	[EMSGSIZE]	
913		Message too large	
914		A message sent on a transport provider was larger than an internal message buffer or some	
915		other network limit.	
916	RT	[EMSGSIZE]	
917		Inappropriate message buffer length.	
918	EX	[EMULTIHOP]	
919		Reserved	
920		[ENAMETOOLONG]	
921		Filename too long	
922		The length of a pathname exceeds {PATH_MAX}, or a pathname component is longer than	
923		{NAME_MAX} and {_POSIX_NO_TRUNC} was in effect for that file.	
924	EX	[ENETDOWN]	
925		Network is down	
926		The local interface used to reach the destination is down.	
927	EX	[ENETUNREACH]	
928		Network unreachable	
929		No route to the network is present.	

930		[ENFILE]	
931		Too many files open in system	
932		Too many files are currently open in the system. The system has reached its predefined limit	
933		for simultaneously open files and temporarily cannot accept requests to open another one.	
934	EX	[ENOBUFFS]	
935		No buffer space available	
936		Insufficient buffer resources were available in the system to perform the socket operation.	
937	EX	[ENODATA]	
938		No message available	
939		No message is available on the STREAM head read queue.	
940		[ENODEV]	
941		No such device	
942		An attempt was made to apply an inappropriate function to a device; for example, trying to	
943		read a write-only device such as a printer.	
944		[ENOENT]	
945		No such file or directory	
946		A component of a specified pathname does not exist, or the pathname is an empty string.	
947		[ENOEXEC]	
948		Executable file format error	
949		A request is made to execute a file that, although it has the appropriate permissions, is not	
950		in the format required by the implementation for executable files.	
951		[ENOLCK]	
952		No locks available	
953		A system-imposed limit on the number of simultaneous file and record locks has been	
954		reached and no more are currently available.	
955	EX	[ENOLINK]	
956		Reserved	
957		[ENOMEM]	
958		Not enough space	
959		The new process image requires more memory than is allowed by the hardware or system-	
960		imposed memory management constraints.	
961	EX	[ENOMSG]	
962		No message of the desired type	
963		The message queue does not contain a message of the required type during interprocess	
964		communication.	
965	EX	[ENOPROTOPT]	
966		Protocol not available	
967		The protocol option specified to <i>setsockopt()</i> is not supported by the implementation.	
968		[ENOSPC]	
969		No space left on a device	
970		During the <i>write()</i> function on a regular file or when extending a directory, there is no free	
971		space left on the device.	
972	EX	[ENOSR]	
973		No STREAM resources	
974		Insufficient STREAMS memory resources are available to perform a STREAMS related	
975		function. This is a temporary condition; one may recover from it if other processes release	

976		resources.	
977	EX	[ENOSTR]	
978		Not a STREAM	
979		A STREAM function was attempted on a file descriptor that was not associated with a	
980		STREAMS device.	
981		[ENOSYS]	
982		Function not implemented	
983		An attempt was made to use a function that is not available in this implementation.	
984	EX	[ENOTCONN]	
985		Socket not connected	
986		The socket is not connected.	
987		[ENOTDIR]	
988		Not a directory	
989		A component of the specified pathname exists, but it is not a directory, when a directory	
990		was expected.	
991		[ENOTEMPTY]	
992		Directory not empty	
993		A directory with entries other than dot and dot-dot was supplied when an empty directory	
994		was expected.	
995	EX	[ENOTSOCK]	
996		Not a socket	
997		The file descriptor does not refer to a socket.	
998		[ENOTSUP]	
999		Not supported	
1000		The implementation does not support this feature of the Realtime Feature Group.	
1001		[ENOTTY]	
1002		Inappropriate I/O control operation	
1003		A control function has been attempted for a file or special file for which the operation is	
1004		inappropriate.	
1005		[ENXIO]	
1006		No such device or address	
1007		Input or output on a special file refers to a device that does not exist, or makes a request	
1008		beyond the capabilities of the device. It may also occur when, for example, a tape drive is	
1009		not on-line.	
1010	EX	[EOPNOTSUPP]	
1011		Operation not supported on socket	
1012		The type of socket (address family or protocol) does not support the requested operation.	
1013	EX	[EOVERFLOW]	
1014		Value too large to be stored in data type	
1015		The user ID or group ID of an IPC or file system object was too large to be stored into	
1016		appropriate member of the caller-provided structure. This error will only occur on	
1017		implementations that support a larger range of user ID or group ID values than the declared	
1018		structure member can support. This usually occurs because the IPC or file system object	
1019		resides on a remote machine with a larger value of the type uid_t , off_t or gid_t than the	
1020		local system.	

1021		[EPERM]	
1022		Operation not permitted	
1023		An attempt was made to perform an operation limited to processes with appropriate privileges or to the owner of a file or other resource.	
1024			
1025		[EPIPE]	
1026		Broken pipe	
1027	EX	A write was attempted on a <code>socket</code> , pipe or FIFO for which there is no process to read the data.	
1028			
1029	EX	[EPROTO]	
1030		Protocol error	
1031		Some protocol error occurred. This error is device specific, but is generally not related to a hardware failure.	
1032			
1033	EX	[EPROTONOSUPPORT]	
1034		Protocol not supported	
1035		The protocol is not supported by the address family, or the protocol is not supported by the implementation.	
1036			
1037	EX	[EPROTOTYPE]	
1038		Socket type not supported	
1039		The socket type is not supported by the protocol.	
1040		[ERANGE]	
1041		Result too large or too small	
1042		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.)	
1043			
1044		[EROFS]	
1045		Read-only file system	
1046		An attempt was made to modify a file or directory on a file system that is read only.	
1047		[ESPIPE]	
1048		Invalid seek	
1049		An attempt was made to access the file offset associated with a pipe or FIFO.	
1050		[ESRCH]	
1051		No such process	
1052		No process can be found corresponding to that specified by the given process ID.	
1053	EX	[ESTALE]	
1054		Reserved	
1055	EX	[ETIME]	
1056		STREAM <code>ioctl()</code> timeout	
1057		The timer set for a STREAMS <code>ioctl()</code> 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 <code>ioctl()</code> operation is indeterminate.	
1058			
1059			
1060	EX	[ETIMEDOUT]	
1061		Connection timed out	
1062		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 behaviour associated with a successful completion of the function.	
1063			
1064			
1065			

1066	RT	[ETIMEDOUT]	
1067		Operation timed out	
1068		The time limit associated with the operation was exceeded before the operation completed.	
1069	EX	[ETXTBSY]	
1070		Text file busy	
1071		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	
1072		being executed.	
1073			
1074	EX	[EWOULDBLOCK]	
1075		Operation would block	
1076		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.	
1077			
1078		An XSI-conforming implementation may assign the same values for [EWOULDBLOCK]	
1079		and [EAGAIN].	
1080		[EXDEV]	
1081		Improper link	
1082		A link to a file on another file system was attempted.	
1083	2.3.1	Additional Error Numbers	
1084		Additional implementation-dependent error numbers may be defined in <code><errno.h></code> .	

2.4 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-dependent 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-dependent, 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 startup, 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.

2.4.1 Interaction of File Descriptors and Standard I/O Streams

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 will be 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*) are defined elsewhere in this specification, but if two or more handles are used, and any one of them is a stream, their actions must be 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 actions below must be 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 must 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 will 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 must assure that, if both handles will ever be accessed, that they will both be in a state where the other could become the active handle first. The application must 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), either an *fflush()* must be done, or the stream must 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, either an *fflush()* must occur or the stream must 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 must 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 will ensure that an application, even one consisting of several processes, will yield correct results: no data will be lost or duplicated when writing, and all data will be written in order, except as requested by seeks. It is implementation-dependent whether, and under what conditions, all input is seen exactly once.

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

2.4.2 Stream Orientation

For conformance to the Multibyte Support Extension, the definition of a stream is adjusted to include an *orientation* for both text and binary streams. 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 becomes *wide-orientated*. Similarly, once a byte input/output function has been applied to a stream without orientation, the stream becomes *byte-orientated*. Only a call to the *freopen()* function or the *fwide()* function can otherwise alter the orientation of a stream.

A successful call to *freopen()* removes any orientation. The three predefined streams *standard input*, *standard output* and *standard error* are unorientated at program startup.

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

- Binary wide-orientated streams have the file positioning restrictions ascribed to both text and binary streams.
- For wide-orientated 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-orientated stream has an associated **mbstate_t** object that stores the current parse state of the stream. A successful call to *fgetpos()* stores 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 restores the value of the associated **mbstate_t** object as well as the position within the controlled stream.

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

- Multibyte encodings within files may contain embedded null bytes (unlike multibyte encodings valid for use internal to the program).

1217	<ul style="list-style-type: none">• A file need not begin nor end in the initial shift state.	
1218	Moreover, the encodings used for characters may differ among files. Both the nature and choice	
1219	of such encodings are implementation-dependent.	
1220	The wide-character input functions read characters from the stream and convert them to wide-	
1221	characters as if they were read by successive calls to the <i>fgetwc()</i> function. Each conversion	
1222	occurs as if by a call to the <i>mbrtowc()</i> function, with the conversion state described by the	
1223	stream's own mbstate_t object.	
1224	The wide-character output functions convert wide-characters to (possibly multibyte) characters	
1225	and write them to the stream as if they were written by successive calls to the <i>fputwc()</i> function.	
1226	Each conversion occurs as if by a call to the <i>wcrtomb()</i> function, with the conversion state	
1227	described by the stream's own mbstate_t object.	
1228	An <i>encoding error</i> occurs if the character sequence presented to the underlying <i>mbrtowc()</i>	
1229	function does not form a valid (generalised) character, or if the code value passed to the	
1230	underlying <i>wcrtomb()</i> function does not correspond to a valid (generalised) character. The	
1231	wide-character input/output functions and the byte input/output functions store the value of	
1232	the macro <i>EILSEQ</i> in <i>errno</i> if and only if an encoding error occurs.	

2.5 STREAMS

EX STREAMS provides a uniform mechanism for implementing networking services and other character-based I/O. The STREAMS interface 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 interface 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 neighbour modules in the STREAM, and passes them to the other neighbour. 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 interface between the STREAM and the rest of the implementation is provided by a set of functions at the STREAM head. When a process calls *write()*, *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 queueing 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 after any other high-priority messages already 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.

1279 **Message Parts**

1280 A process may access STREAMS messages that contain a data part, control part, or both. The
 1281 data part is that information which is transmitted over the communication medium and the
 1282 control information is used by the local STREAMS modules. The other types of messages are
 1283 used between modules and are not accessible to processes. Messages containing only a data part
 1284 are accessible via *putmsg()*, *putpmsg()*, *getmsg()*, *getpmsg()*, *read()* or *write()*. Messages
 1285 containing a control part with or without a data part are accessible via calls to *putmsg()*,
 1286 *putpmsg()*, *getmsg()* or *getpmsg()*.

1287 **2.5.1 Accessing STREAMS**

1288 A process accesses STREAMS-based files using the standard functions *open()*, *close()*, *read()*,
 1289 *write()*, *ioctl()*, *pipe()*, *putmsg()*, *putpmsg()*, *getmsg()*, *getpmsg()* or *poll()*. Refer to the applicable
 1290 function definitions for general properties and errors.

1291 Calls to *ioctl()* are used to perform control functions with the STREAMS-based device associated
 1292 with the file descriptor *fildes*. The arguments *command* and *arg* are passed to the STREAMS file
 1293 designated by *fildes* and are interpreted by the STREAM head. Certain combinations of these
 1294 arguments may be passed to a module or driver in the STREAM.

1295 Since these STREAMS requests are a subset of *ioctl()*, they are subject to the errors described
 1296 there.

1297 STREAMS modules and drivers can detect errors, sending an error message to the STREAM
 1298 head, thus causing subsequent functions to fail and set *errno* to the value specified in the
 1299 message. In addition, STREAMS modules and drivers can elect to fail a particular *ioctl()* request
 1300 alone by sending a negative acknowledgement message to the STREAM head. This causes just
 1301 the pending *ioctl()* request to fail and set *errno* to the value specified in the message.

2.6 Interprocess Communication

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

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

Another Interprocess Communication facility is provided by functions in the Realtime Feature Group.

2.6.1 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 on *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 `<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 `<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.

1346 • The effective user ID of the process does not match `xxx_perm.cuid` or `xxx_perm.uid` and the
1347 effective group ID of the process does not match `xxx_perm.cgid` or `xxx_perm.gid` in the data
1348 structure associated with the IPC identifier, but the appropriate bit of the *other* field in
1349 `xxx_perm.mode` is set.

1350 Otherwise, the permission is denied. |

2.7 Realtime

This section defines system interfaces to support the source portability of applications with realtime requirements.

The definition of *realtime* used in defining the scope of XSI provisions is:

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

The key elements of defining the scope are:

1. defining a sufficient set of functionality to cover a significant part of the realtime application program domain, and
2. defining sufficient performance constraints and performance-related functions to allow a realtime application to achieve deterministic response from the system.

Specifically within the scope, it is required to define interfaces that do not preclude high-performance implementations on traditional uniprocessor realtime systems.

Wherever possible, the requirements of other application environments are included in this interface definition. It is beyond the scope of these interfaces to support networking or multiprocessor functionality.

The specific functional areas included in this section and their scope include:

- *Semaphores*: A minimum synchronisation primitive to serve as a basis for more complex synchronisation mechanisms to be defined by the application program.
- *Process memory locking*: A performance improvement facility to bind application programs into the high-performance random access memory of a computer system. This avoids potential latencies introduced by the operating system in storing parts of a program that were not recently referenced on secondary memory devices.
- *Memory mapped files and shared memory objects*: A performance improvement facility to allow for programs to access files as part of the address space and for separate application programs to have portions of their address space commonly accessible.
- *Priority scheduling*: A performance and determinism improvement facility to allow applications to determine the order in which threads that are ready to run are granted access to processor resources.
- *Realtime signal extension*: A determinism improvement facility that augments the BASE signals mechanism to enable asynchronous signal notifications to an application to be queued without impacting compatibility with the existing signals interface.
- *Timers*: A functionality and determinism improvement facility to increase the resolution and capabilities of the time-base interface.
- *POSIX Interprocess communication*: A functionality enhancement to add a high-performance, deterministic interprocess communication facility for local communication. Network transparency is beyond the scope of this interface.
- *Synchronised input and output*: A determinism and robustness improvement mechanism to enhance the data input and output mechanisms, so that an application can insure that the data being manipulated is physically present on secondary mass storage devices.
- *Asynchronous input and output*: A functionality enhancement to allow an application process to queue data input and output commands with asynchronous notification of completion. This facility includes in its scope the requirements of supercomputer applications.

1394 All the interfaces defined in the Realtime Feature Group will be portable, although some of the
1395 numeric parameters used by an implementation may have hardware dependencies.

1396 2.7.1 Signal Generation and Delivery

1397 Some signal-generating functions, such as high-resolution timer expiration, asynchronous I/O
1398 completion, interprocess message arrival, and the *sigqueue()* function, support the specification
1399 of an application-defined value, either explicitly as a parameter to the function or in a **sigevent**
1400 structure parameter. The **sigevent** structure is defined in **<signal.h>** and contains at least the
1401 following members:

1402	Member Type	Member Name	Description
1404	int	sigev_notify	Notification type
1405	int	sigev_signo	Signal number
1406	union sigval	sigev_value	Signal value
1407	void(*) (unsigned sigval)	sigev_notify_function	Notification
1408	(pthread_attr_t*)	sigev_notify_attributes	Notification attributes

1409 The *sigev_notify* member specifies the notification mechanism to use when an asynchronous
1410 event occurs. This document defines the following values for the *sigev_notify* member:

1411 SIGEV_NONE No asynchronous notification will be delivered when the event of interest
1412 occurs.

1413 SIGEV_SIGNAL A queued signal, with an application-defined value, will be generated
1414 when the event of interest occurs.

1415 SIGEV_THREAD A notification function will be called to perform notification.

1416 An implementation may define additional notification mechanisms.

1417 The *sigev_signo* member specifies the signal to be generated. The *sigev_value* member is the
1418 application-defined value to be passed to the signal-catching function at the time of the signal
1419 delivery as the *si_value* member of the **siginfo_t** structure.

1420 The **sigval** union is defined in **<signal.h>** and contains at least the following members:

1421	Member Type	Member Name	Description
1422	int	sival_int	Integer signal value
1423	void *	sival_ptr	Pointer signal value
1424			

1425 The *sival_int* member is used when the application-defined value is of type **int**; the *sival_ptr*
1426 member is used when the application-defined value is a pointer.

1427 When a signal is generated by the *sigqueue()* function or any signal-generating function that
1428 supports the specification of an application-defined value, the signal is marked pending and, if
1429 the SA_SIGINFO flag is set for that signal, the signal is queued to the process along with the
1430 application-specified signal value. Multiple occurrences of signals so generated are queued in
1431 FIFO order. It is unspecified whether signals so generated are queued when the SA_SIGINFO
1432 flag is not set for that signal.

1433 Signals generated by the *kill()* function or other events that cause signals to occur, such as
1434 detection of hardware faults, *alarm()* timer expiration, or terminal activity, and for which the
1435 implementation does not support queuing, have no effect on signals already queued for the
1436 same signal number.

When multiple unblocked signals, all in the range SIGRTMIN to SIGRTMAX, are pending, the behaviour is 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.

2.7.2 Asynchronous I/O

An asynchronous I/O control block structure **aiocb** is used in many asynchronous I/O function interfaces. It is defined in **<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 to be 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-dependent 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 will be within the range zero through AIO_PRIO_DELTA_MAX, inclusive. The order of processing of requests submitted by processes whose schedulers are not SCHED_FIFO or SCHED_RR 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 will be 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 will 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-

1485 dependent. If `_POSIX_PRIORITIZED_IO` is defined, the implementation defines for which files
1486 I/O prioritization is supported.

1487 The `aio_sigevent` determines how the calling process will be notified upon I/O completion as
1488 specified in **Signal Generation and Delivery** on page 808. If `aio_sigevent.sigev_notify` is
1489 `SIGEV_NONE`, then no signal will be posted upon I/O completion, but the error status for the
1490 operation and the return status for the operation will be set appropriately.

1491 The `aio_lio_opcode` field is used only by the `lio_listio()` call. The `lio_listio()` call allows multiple
1492 asynchronous I/O operations to be submitted at a single time. The function takes as an
1493 argument an array of pointers to **aiocb** structures. Each **aiocb** structure indicates the operation
1494 to be performed (read or write) via the `aio_lio_opcode` field.

1495 The address of the **aiocb** structure is used as a handle for retrieving the error status and return
1496 status of the asynchronous operation while it is in progress.

1497 The **aiocb** structure and the data buffers associated with the asynchronous I/O operation are
1498 being used by the system for asynchronous I/O while, and only while, the error status of the
1499 asynchronous operation is equal to `EINPROGRESS`. Applications must not modify the **aiocb**
1500 structure while the structure is being used by the system for asynchronous I/O.

1501 The return status of the asynchronous operation is the number of bytes transferred by the I/O
1502 operation. If the error status is set to indicate an error completion, then the return status is set to
1503 the return value that the corresponding `read()`, `write()`, or `fsync()` call would have returned.
1504 When the error status is not equal to `EINPROGRESS`, the return status reflects the return status
1505 of the corresponding synchronous operation.

1506 2.7.3 Memory Management

1507 Range memory locking and memory mapping operations are defined in terms of pages.
1508 Implementations may restrict the size and alignment of range lockings and mappings to be on
1509 page-size boundaries. The page size, in bytes, is the value of the configurable system variable
1510 `{PAGESIZE}`. If an implementation has no restrictions on size or alignment, it may specify a 1
1511 byte page size.

1512 Memory locking guarantees the residence of portions of the address space. It is
1513 implementation-dependent whether locking memory guarantees fixed translation between
1514 virtual addresses (as seen by the process) and physical addresses. Per-process memory locks are
1515 not inherited across a `fork()`, and all memory locks owned by a process are unlocked upon `exec` or
1516 process termination. Unmapping of an address range removes any memory locks established on
1517 that address range by this process.

1518 Memory Mapped Files provide a mechanism that allows a process to access files by directly
1519 incorporating file data into its address space. Once a file is mapped into a process address space,
1520 the data can be manipulated as memory. If more than one process maps a file, its contents are
1521 shared among them. If the mappings allow shared write access then data written into the
1522 memory object through the address space of one process appears in the address spaces of all
1523 processes that similarly map the same portion of the memory object.

1524 RT Shared memory objects are named regions of storage that may be independent of the file system
1525 and can be mapped into the address space of one or more processes to allow them to share the
1526 associated memory.

1527 RT An `unlink()` of a file or `shm_unlink()` of a shared memory object, while causing the removal of the
1528 name, does not unmap any mappings established for the object. Once the name has been
1529 removed, the contents of the memory object are preserved as long as it is referenced. The
1530 memory object remains referenced as long as a process has the memory object open or has some

1531 area of the memory object mapped.

1532 Mapping may be restricted to disallow some types of access. References to whole pages within
 1533 the mapping but beyond the current length of an object result in a SIGBUS signal. SIGBUS is
 1534 used in this context to indicate an error using the object. The size of the object is unaffected by
 1535 access beyond the end of the object. Write attempts to memory that was mapped without write
 1536 access, or any access to memory mapped PROT_NONE, results in a SIGSEGV signal. SIGSEGV
 1537 is used in this context to indicate a mapping error. References to unmapped addresses result in
 1538 a SIGSEGV signal.

1539 **2.7.4 Scheduling Policies**

1540 RT The scheduling semantics described in this specification are defined in terms of a conceptual
 1541 model that contains a set of thread lists. No implementation structures are necessarily implied
 1542 by the use of this conceptual model. It is assumed that no time elapses during operations
 1543 described using this model, and therefore no simultaneous operations are possible. This model
 1544 discusses only processor scheduling for runnable threads, but it should be noted that greatly
 1545 enhanced predictability of realtime applications will result if the sequencing of other resources
 1546 takes processor scheduling policy into account.

1547 There is, conceptually, one thread list for each priority. Any runnable thread may be on any
 1548 thread list. Multiple scheduling policies are provided. Each non-empty thread list is ordered,
 1549 contains a head as one end of its order, and a tail as the other. The purpose of a scheduling
 1550 policy is to define the allowable operations on this set of lists (for example, moving threads
 1551 between and within lists).

1552 Each process is controlled by an associated scheduling policy and priority. These parameters
 1553 may be specified by explicit application execution of the *sched_setscheduler()* or *sched_setparam()*
 1554 functions.

1555 Each thread is controlled by an associated scheduling policy and priority. These parameters
 1556 may be specified by explicit application execution of the *pthread_setschedparam()* function.

1557 Associated with each policy is a priority range. Each policy definition specifies the minimum
 1558 priority range for that policy. The priority ranges for each policy may or may not overlap the
 1559 priority ranges of other policies.

1560 A conforming implementation selects the thread that is defined as being at the head of the
 1561 highest priority non-empty thread list to become a running thread, regardless of its associated
 1562 policy. This thread is then removed from its thread list.

1563 Three scheduling policies are specifically required. Other implementation-dependent
 1564 scheduling policies may be defined. The following symbols are defined in the header **<sched.h>**:

	Symbol	Description
1567	SCHED_FIFO	First in-first out (FIFO) scheduling policy.
1568	SCHED_RR	Round robin scheduling policy.
1569	SCHED_OTHER	Another scheduling policy.

1570 The values of these symbols will be distinct.

1571	SCHED_FIFO	
1572	Conforming implementations include a scheduling policy called the FIFO scheduling policy.	
1573	Threads scheduled under this policy are chosen from a thread list that is ordered by the time its	
1574	threads have been on the list without being executed; generally, the head of the list is the thread	
1575	that has been on the list the longest time, and the tail is the thread that has been on the list the	
1576	shortest time.	
1577	Under the SCHED_FIFO policy, the modification of the definitional thread lists is as follows:	
1578	1. When a running thread becomes a preempted thread, it becomes the head of the thread list	
1579	for its priority.	
1580	2. When a blocked thread becomes a runnable thread, it becomes the tail of the thread list for	
1581	its priority.	
1582	3. When a running thread calls the <i>sched_setscheduler()</i> function, the process specified in the	
1583	function call is modified to the specified policy and the priority specified by the <i>param</i>	
1584	argument.	
1585	4. When a running thread calls the <i>sched_setparam()</i> function, the priority of the process	
1586	specified in the function call is modified to the priority specified by the <i>param</i> argument.	
1587	5. When a running thread calls the <i>pthread_schedsetparam()</i> function, the thread specified in	
1588	the function call is modified to the specified policy and the priority specified by the <i>param</i>	
1589	argument.	
1590	6. If a thread whose policy or priority has been modified is a running thread or is runnable, it	
1591	then becomes the tail of the thread list for its new priority.	
1592	7. When a running thread issues the <i>sched_yield()</i> function, the thread becomes the tail of the	
1593	thread list for its priority.	
1594	8. At no other time will the position of a thread with this scheduling policy within the thread	
1595	lists be affected.	
1596	For this policy, valid priorities shall be within the range returned by the function	
1597	<i>sched_get_priority_max()</i> and <i>sched_get_priority_min()</i> when SCHED_FIFO is provided as the	
1598	parameter. Conforming implementations provide a priority range of at least 32 priorities for this	
1599	policy.	
1600	SCHED_RR	
1601	Conforming implementations include a scheduling policy called the round robin scheduling	
1602	policy. This policy is identical to the SCHED_FIFO policy with the additional condition that	
1603	when the implementation detects that a running thread has been executing as a running thread	
1604	for a time period of the length returned by the function <i>sched_rr_get_interval()</i> or longer, the	
1605	thread becomes the tail of its thread list and the head of that thread list is removed and made a	
1606	running thread.	
1607	The effect of this policy is to ensure that if there are multiple SCHED_RR threads at the same	
1608	priority, one of them will not monopolise the processor. An application should not rely only on	
1609	the use of SCHED_RR to ensure application progress among multiple threads if the application	
1610	includes threads using the SCHED_FIFO policy at the same or higher priority levels or	
1611	SCHED_RR threads at a higher priority level.	
1612	A thread under this policy that is preempted and subsequently resumes execution as a running	
1613	thread completes the unexpired portion of its round-robin-interval time period.	

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

SCHED_OTHER

Conforming implementations 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 are executing under `SCHED_FIFO` or `SCHED_RR` is implementation-dependent.

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 uses only priorities within the range returned by the functions `sched_get_priority_max()` and `sched_get_priority_min()` when `SCHED_OTHER` is provided as the parameter.

2.7.5 Clocks and Timers

The header file `<time.h>` 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
<code>time_t</code>	<code>tv_sec</code>	Seconds
<code>long</code>	<code>tv_nsec</code>	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 (1000 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
<code>struct timespec</code>	<code>it_interval</code>	Timer period
<code>struct timespec</code>	<code>it_value</code>	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 is 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 will be disarmed after its next expiration; that is, a one-shot timer is specified.

1652	Timer Event Notification Control Block	
1653	Per-process timers may be created that notify the process of timer expirations by queuing a realtime extended signal. The sigevent structure, defined in <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.	
1654		
1655		
1656		
1657	Manifest Constants	
1658	The following constants are defined in <time.h> :	
1659	CLOCK_REALTIME The identifier for the systemwide realtime clock.	
1660	TIMER_ABSTIME Flag indicating time is absolute with respect to the clock associated with a timer.	
1661		
1662	The maximum allowable resolution for the CLOCK_REALTIME clock and all timers based on this clock, including the <i>nanosleep()</i> function, is represented by {_POSIX_CLOCKRES_MIN} and is defined as 20 ms (1/50 of a second). Implementations may support smaller values of resolution for the CLOCK_REALTIME clock to provide finer granularity time bases.	
1663		
1664		
1665		
1666	The minimum allowable maximum value for the CLOCK_REALTIME clock and absolute timers based on it is the same as that defined by the ISO C standard for the <i>time_t</i> type.	
1667		

2.8 Threads

This defines interfaces and functionality to support multiple flows of control, called *threads*, within a process.

Threads define system interfaces to support the source portability of applications. The key elements defining the scope are:

- a. defining a sufficient set of functionality to support multiple threads of control within a process
- b. defining a sufficient set of functionality to support the realtime application domain
- c. defining sufficient performance constraints and performance related functions to allow a realtime application to achieve deterministic response from the system.

The definition of realtime used in defining the scope of this specification is:

The ability of the system to provide a required level of service in a bounded response time.

Wherever possible, the requirements of other application environments are included in the interface definition. The Threads interfaces are specifically targeted at supporting tightly coupled multitasking environments including multiprocessors and advanced language constructs.

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.
- Synchronisation primitives optimised for tightly coupled operation of multiple control flows in a common, shared address space.
- Harmonization of the threads interfaces with the existing BASE interfaces.

2.8.1 Supported Interfaces

On XSI-conformant systems, `_POSIX_THREADS`, `_POSIX_THREAD_ATTR_STACKADDR`, `_POSIX_THREAD_ATTR_STACKSIZE` and `_POSIX_THREAD_PROCESS_SHARED` are always defined. Therefore, the following threads interfaces are always supported:

POSIX Interfaces

<code>pthread_atfork()</code>	<code>pthread_detach()</code>
<code>pthread_attr_destroy()</code>	<code>pthread_equal()</code>
<code>pthread_attr_getdetachstate()</code>	<code>pthread_exit()</code>
<code>pthread_attr_getschedparam()</code>	<code>pthread_getspecific()</code>
<code>pthread_attr_getstackaddr()</code>	<code>pthread_join()</code>
<code>pthread_attr_getstacksize()</code>	<code>pthread_key_create()</code>
<code>pthread_attr_init()</code>	<code>pthread_key_delete()</code>
<code>pthread_attr_setdetachstate()</code>	<code>pthread_kill()</code>
<code>pthread_attr_setschedparam()</code>	<code>pthread_mutex_destroy()</code>
<code>pthread_attr_setstackaddr()</code>	<code>pthread_mutex_init()</code>
<code>pthread_attr_setstacksize()</code>	<code>pthread_mutex_lock()</code>
<code>pthread_cancel()</code>	<code>pthread_mutex_trylock()</code>
<code>pthread_cleanup_pop()</code>	<code>pthread_mutex_unlock()</code>
<code>pthread_cleanup_push()</code>	<code>pthread_mutexattr_destroy()</code>

1709	<i>pthread_cond_broadcast()</i>	<i>pthread_mutexattr_getpshared()</i>
1710	<i>pthread_cond_destroy()</i>	<i>pthread_mutexattr_init()</i>
1711	<i>pthread_cond_init()</i>	<i>pthread_mutexattr_setpshared()</i>
1712	<i>pthread_cond_signal()</i>	<i>pthread_once()</i>
1713	<i>pthread_cond_timedwait()</i>	<i>pthread_self()</i>
1714	<i>pthread_cond_wait()</i>	<i>pthread_setcancelstate()</i>
1715	<i>pthread_condattr_destroy()</i>	<i>pthread_setcanceltype()</i>
1716	<i>pthread_condattr_getpshared()</i>	<i>pthread_setspecific()</i>
1717	<i>pthread_condattr_init()</i>	<i>pthread_sigmask()</i>
1718	<i>pthread_condattr_setpshared()</i>	<i>pthread_testcancel()</i>
1719	<i>pthread_create()</i>	<i>sigwait()</i>

1720 X/Open Interfaces

1721	EX	<i>pthread_attr_getguardsize()</i>	<i>pthread_rwlock_trywrlock()</i>
1722		<i>pthread_attr_setguardsize()</i>	<i>pthread_rwlock_unlock()</i>
1723		<i>pthread_getconcurrency()</i>	<i>pthread_rwlock_wrlock()</i>
1724		<i>pthread_mutexattr_gettype()</i>	<i>pthread_rwlockattr_destroy()</i>
1725		<i>pthread_mutexattr_settype()</i>	<i>pthread_rwlockattr_getpshared()</i>
1726		<i>pthread_rwlock_destroy()</i>	<i>pthread_rwlockattr_init()</i>
1727		<i>pthread_rwlock_init()</i>	<i>pthread_rwlockattr_setpshared()</i>
1728		<i>pthread_rwlock_rdlock()</i>	<i>pthread_setconcurrency()</i>
1729		<i>pthread_rwlock_tryrdlock()</i>	

1731 On XSI-conformant systems, `_POSIX_THREAD_SAFE_FUNCTIONS` is always defined.
 1732 Therefore, the following interfaces are always supported:

1733	<i>asctime_r()</i>	<i>getpwnam_r()</i>
1734	<i>ctime_r()</i>	<i>getpwuid_r()</i>
1735	<i>flockfile()</i>	<i>gmtime_r()</i>
1736	<i>ftrylockfile()</i>	<i>localtime_r()</i>
1737	<i>funlockfile()</i>	<i>putc_unlocked()</i>
1738	<i>getc_unlocked()</i>	<i>putchar_unlocked()</i>
1739	<i>getchar_unlocked()</i>	<i>rand_r()</i>
1740	<i>getgrgid_r()</i>	<i>readdir_r()</i>
1741	<i>getgrnam_r()</i>	<i>strtok_r()</i>

1742 The following threads interfaces are only supported on XSI-conformant systems if the Realtime
 1743 Threads Feature Group is supported (see Section 1.3.3 on page 4):

1744	RTT	<i>pthread_attr_getinheritsched()</i>	<i>pthread_mutex_getprioceiling()</i>
1745		<i>pthread_attr_getschedpolicy()</i>	<i>pthread_mutex_setprioceiling()</i>
1746		<i>pthread_attr_getscope()</i>	<i>pthread_mutexattr_getprioceiling()</i>
1747		<i>pthread_attr_setinheritsched()</i>	<i>pthread_mutexattr_getprotocol()</i>
1748		<i>pthread_attr_setschedpolicy()</i>	<i>pthread_mutexattr_setprioceiling()</i>
1749		<i>pthread_attr_setscope()</i>	<i>pthread_mutexattr_setprotocol()</i>
1750		<i>pthread_getschedparam()</i>	<i>pthread_setschedparam()</i>

1752 **2.8.2 Thread-safety**

1753 All interfaces defined by this specification will be thread-safe, except that the following
1754 interfaces need not be thread-safe:

1755 **POSIX Interfaces**

1756	<i>asctime()</i>	<i>getgrgid()</i>	<i>getpwnam()</i>	<i>putc_unlocked()</i>	<i>strtok()</i>
1757	<i>ctime()</i>	<i>getgrnam()</i>	<i>getpwuid()</i>	<i>putchar_unlocked()</i>	<i>ttyname()</i>
1758	<i>getc_unlocked()</i>	<i>getlogin()</i>	<i>gmtime()</i>	<i>rand()</i>	
1759	<i>getchar_unlocked()</i>	<i>getopt()</i>	<i>localtime()</i>	<i>readdir()</i>	

1760 **X/Open Interfaces**

1761	EX	<i>basename()</i>	<i>dbm_open()</i>	<i>fcvt()</i>	<i>getutxline()</i>	<i>pututxline()</i>
1762		<i>catgets()</i>	<i>dbm_store()</i>	<i>gamma()</i>	<i>getw()</i>	<i>setgrent()</i>
1763		<i>dbm_clearerr()</i>	<i>dirname()</i>	<i>gcvt()</i>	<i>l64a()</i>	<i>setkey()</i>
1764		<i>dbm_close()</i>	<i>drand48()</i>	<i>getdate()</i>	<i>lgamma()</i>	<i>setpwent()</i>
1765		<i>dbm_delete()</i>	<i>ecvt()</i>	<i>getenv()</i>	<i>lrand48()</i>	<i>setutxent()</i>
1766		<i>dbm_error()</i>	<i>encrypt()</i>	<i>getgrent()</i>	<i>mrnd48()</i>	<i>strerror()</i>
1767		<i>dbm_fetch()</i>	<i>endgrent()</i>	<i>getpwent()</i>	<i>nl_langinfo()</i>	
1768		<i>dbm_firstkey()</i>	<i>endpwent()</i>	<i>getutxent()</i>	<i>ptsname()</i>	
1769		<i>dbm_nextkey()</i>	<i>endutxent()</i>	<i>getutxid()</i>	<i>putenv()</i>	

1770

1771 The interfaces *ctermid()* and *tmpnam()* need not be thread-safe if passed a NULL argument.

1772 EX The interfaces in the Legacy Feature Group need not be thread-safe.

1773 Implementations will provide internal synchronisation as necessary in order to satisfy this
1774 requirement.

1775 **2.8.3 Thread Implementation Models**

1776 EX There are various thread implementation models. At one end of the spectrum is the “library-
1777 thread model”. In such a model, the threads of a process are not visible to the operating system
1778 kernel, and the threads are not kernel scheduled entities. The process is the only kernel
1779 scheduled entity. The process is scheduled onto the processor by the kernel according to the
1780 scheduling attributes of the process. The threads are scheduled onto the single kernel scheduled
1781 entity (the process) by the run-time library according to the scheduling attributes of the threads.
1782 A problem with this model is that it constrains concurrency. Since there is only one kernel
1783 scheduled entity (namely, the process), only one thread per process can execute at a time. If the
1784 thread that is executing blocks on I/O, then the whole process blocks.

1785 At the other end of the spectrum is the “kernel-thread model”. In this model, all threads are
1786 visible to the operating system kernel. Thus, all threads are kernel scheduled entities, and all
1787 threads can concurrently execute. The threads are scheduled onto processors by the kernel
1788 according to the scheduling attributes of the threads. The drawback to this model is that the
1789 creation and management of the threads entails operating system calls, as opposed to subroutine
1790 calls, which makes kernel threads heavier weight than library threads.

1791 Hybrids of these two models are common. A hybrid model offers the speed of library threads
1792 and the concurrency of kernel threads. In hybrid models, a process has some (relatively small)
1793 number of kernel scheduled entities associated with it. It also has a potentially much larger

1794 number of library threads associated with it. Some library threads may be bound to kernel
 1795 scheduled entities, while the other library threads are multiplexed onto the remaining kernel
 1796 scheduled entities. There are two levels of thread scheduling:

1797 • The run-time library manages the scheduling of (unbound) library threads onto kernel
 1798 scheduled entities.

1799 • The kernel manages the scheduling of kernel scheduled entities onto processors.

1800 For this reason, a hybrid model is referred to as a “two-level threads scheduling model”. In this
 1801 model, the process can have multiple concurrently executing threads; specifically, it can have as
 1802 many concurrently executing threads as it has kernel scheduled entities.

1803 2.8.4 Thread Mutexes

1804 A thread that has blocked will not prevent any unblocked thread that is eligible to use the same
 1805 processing resources from eventually making forward progress in its execution. Eligibility for
 1806 processing resources is determined by the scheduling policy.

1807 A thread becomes the owner of a mutex, *m*, when either:

- 1808 1. it returns successfully from `pthread_mutex_lock()` with *m* as the *mutex* argument, or
- 1809 2. it returns successfully from `pthread_mutex_trylock()` with *m* as the *mutex* argument, or
- 1810 3. it returns (successfully or not) from `pthread_cond_wait()` with *m* as the *mutex* argument
 1811 (except as explicitly indicated otherwise for certain errors), or
- 1812 4. it returns (successfully or not) from `pthread_cond_timedwait()` with *m* as the *mutex*
 1813 argument (except as explicitly indicated otherwise for certain errors).

1814 The thread remains the owner of *m* until it either:

- 1815 1. executes `pthread_mutex_unlock()` with *m* as the *mutex* argument, or
- 1816 2. blocks in a call to `pthread_cond_wait()` with *m* as the *mutex* argument, or
- 1817 3. blocks in a call to `pthread_cond_timedwait()` with *m* as the *mutex* argument.

1818 The implementation behaves as if at all times there is at most one owner of any mutex.

1819 A thread that becomes the owner of a mutex is said to have *acquired* the mutex and the mutex is
 1820 said to have become *locked*; when a thread gives up ownership of a mutex it is said to have
 1821 *released* the mutex and the mutex is said to have become *unlocked*.

1822 2.8.5 Thread Scheduling Attributes

1823 RTT In support of the scheduling interface, threads have attributes which are accessed through the
 1824 `pthread_attr_t` thread creation attributes object.

1825 The *contentionscope* attribute defines the scheduling contention scope of the thread to be either
 1826 `PTHREAD_SCOPE_PROCESS` or `PTHREAD_SCOPE_SYSTEM`.

1827 The *inheritsched* attribute specifies whether a newly created thread is to inherit the scheduling
 1828 attributes of the creating thread or to have its scheduling values set according to the other
 1829 scheduling attributes in the `pthread_attr_t` object.

1830 The *schedpolicy* attribute defines the scheduling policy for the thread. The *schedparam* attribute
 1831 defines the scheduling parameters for the thread. The interaction of threads having different
 1832 policies within a process is described as part of the definition of those policies.

If the `_POSIX_THREAD_PRIORITY_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*.

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

2.8.6 Thread Scheduling Contention Scope

The scheduling contention scope of a thread defines the set of threads with which the thread must compete for use of the processing resources. The scheduling operation will select 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-dependent 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 support the `PTHREAD_SCOPE_PROCESS` scheduling contention scope, the `PTHREAD_SCOPE_SYSTEM` scheduling contention scope, or both.

2.8.7 Scheduling Allocation Domain

Implementations 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 multi-processors 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 must contend 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.

1880 The choice of scheduling allocation domain size and the level of application control over
 1881 scheduling allocation domains is implementation-dependent. Conforming implementations
 1882 may change the size of scheduling allocation domains and the binding of threads to scheduling
 1883 allocation domains at any time.

1884 For application threads with scheduling allocation domains of size equal to one, the scheduling
 1885 rules defined for SCHED_FIFO and SCHED_RR will be used. All threads with system
 1886 scheduling contention scope, regardless of the processes in which they reside, compete for the
 1887 processor according to their priorities. Threads with process scheduling contention scope
 1888 compete only with other threads with process scheduling contention scope within their process.

1889 For application threads with scheduling allocation domains of size greater than one, the rules
 1890 defined for SCHED_FIFO and SCHED_RR are used in an implementation-dependent manner.
 1891 Each thread with system scheduling contention scope competes for the processors in its
 1892 scheduling allocation domain in an implementation-dependent manner according to its priority.
 1893 Threads with process scheduling contention scope are scheduled relative to other threads within
 1894 the same scheduling contention scope in the process.

1895 **2.8.8 Thread Cancellation**

1896 The thread cancellation mechanism allows a thread to terminate the execution of any other
 1897 thread in the process in a controlled manner. The target thread (that is, the one that is being
 1898 canceled) is allowed to hold cancellation requests pending in a number of ways and to perform
 1899 application-specific cleanup processing when the notice of cancellation is acted upon.

1900 Cancellation is controlled by the cancellation control interfaces. Each thread maintains its own
 1901 cancelability state. Cancellation may only occur at cancellation points or when the thread is
 1902 asynchronously cancelable.

1903 The thread cancellation mechanism described in this section depends upon programs having set
 1904 *deferred cancelability* state, which is specified as the default. Applications must also carefully
 1905 follow static lexical scoping rules in their execution behaviour. For instance, use of *setjmp()*,
 1906 *return*, *goto*, and so on, to leave user-defined cancellation scopes without doing the
 1907 necessary scope pop operation will result in undefined behaviour.

1908 Use of asynchronous cancelability while holding resources which potentially need to be released
 1909 may result in resource loss. Similarly, cancellation scopes may only be safely manipulated
 1910 (pushed and popped) when the thread is in the *deferred* or *disabled* cancelability states.

1911 **2.8.8.1 Cancelability States**

1912 The cancelability state of a thread determines the action taken upon receipt of a cancellation
 1913 request. The thread may control cancellation in a number of ways.

1914 Each thread maintains its own cancelability state, which may be encoded in two bits:

1915 *Cancelability Enable*

1916 When cancelability is PTHREAD_CANCEL_DISABLE, cancellation requests against the
 1917 target thread are held pending. By default, cancelability is set to
 1918 PTHREAD_CANCEL_ENABLE.

1919 *Cancelability Type*

1920 When cancelability is enabled and the cancelability type is
 1921 PTHREAD_CANCEL_ASYNCHRONOUS, new or pending cancellation requests may be
 1922 acted upon at any time. When cancelability is enabled and the cancelability type is
 1923 PTHREAD_CANCEL_DEFERRED, cancellation requests are held pending until a
 1924 cancellation point (see below) is reached. If cancelability is disabled, the setting of the

1925 cancelability type has no immediate effect as all cancellation requests are held pending,
 1926 however, once cancelability is enabled again the new type will be in effect. The
 1927 cancelability type is PTHREAD_CANCEL_DEFERRED in all newly created threads
 1928 including the thread in which *main()* was first invoked.

1929 2.8.8.2 Cancellation Points

1930 Cancellation points occur when a thread is executing the following functions:

1931			
1932	<i>aio_suspend()</i>	<i>pause()</i>	<i>sigsuspend()</i>
1933	<i>close()</i>	<i>poll()</i>	<i>sigtimedwait()</i>
1934	<i>creat()</i>	<i>pread()</i>	<i>sigwait()</i>
1935	<i>fcntl()</i> ¹	<i>pthread_cond_timedwait()</i>	<i>sigwaitinfo()</i>
1936	<i>fsync()</i>	<i>pthread_cond_wait()</i>	<i>sleep()</i>
1937	<i>getmsg()</i>	<i>pthread_join()</i>	<i>system()</i>
1938	<i>getpmsg()</i>	<i>pthread_testcancel()</i>	<i>tcdrain()</i>
1939	<i>lockf()</i>	<i>putmsg()</i>	<i>usleep()</i>
1940	<i>mq_receive()</i>	<i>putpmsg()</i>	<i>wait()</i>
1941	<i>mq_send()</i>	<i>pwrite()</i>	<i>wait3()</i>
1942	<i>msgrcv()</i>	<i>read()</i>	<i>waitid()</i>
1943	<i>msgsnd()</i>	<i>readv()</i>	<i>waitpid()</i>
1944	<i>msync()</i>	<i>select()</i>	<i>write()</i>
1945	<i>nanosleep()</i>	<i>sem_wait()</i>	<i>writew()</i>
1946	<i>open()</i>	<i>sigpause()</i>	

1947 _____

1948 1. When the *cmd* argument is F_SETLK.

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

<i>catclose()</i>	<i>fwprintf()</i>	<i>popen()</i>
<i>catgets()</i>	<i>fwrite()</i>	<i>printf()</i>
<i>catopen()</i>	<i>fwscanf()</i>	<i>putc()</i>
<i>closedir()</i>	<i>getc()</i>	<i>putc_unlocked()</i>
<i>closelog()</i>	<i>getc_unlocked()</i>	<i>putchar()</i>
<i>ctermid()</i>	<i>getchar()</i>	<i>putchar_unlocked()</i>
<i>dbm_close()</i>	<i>getchar_unlocked()</i>	<i>puts()</i>
<i>dbm_delete()</i>	<i>getcwd()</i>	<i>pututxline()</i>
<i>dbm_fetch()</i>	<i>getdate()</i>	<i>putw()</i>
<i>dbm_nextkey()</i>	<i>getgrent()</i>	<i>putwc()</i>
<i>dbm_open()</i>	<i>getgrgid()</i>	<i>putwchar()</i>
<i>dbm_store()</i>	<i>getgrgid_r()</i>	<i>readdir()</i>
<i>dlclose()</i>	<i>getgrnam()</i>	<i>readdir_r()</i>
<i>dlopen()</i>	<i>getgrnam_r()</i>	<i>remove()</i>
<i>endgrent()</i>	<i>getlogin()</i>	<i>rename()</i>
<i>endpwent()</i>	<i>getlogin_r()</i>	<i>rewind()</i>
<i>endutxent()</i>	<i>getpwent()</i>	<i>rewinddir()</i>
<i>fclose()</i>	<i>getpwnam()</i>	<i>scanf()</i>
<i>fcntl()</i> ²	<i>getpwnam_r()</i>	<i>seekdir()</i>
<i>fflush()</i>	<i>getpwuid()</i>	<i>semop()</i>
<i>fgetc()</i>	<i>getpwuid_r()</i>	<i>setgrent()</i>
<i>fgetpos()</i>	<i>gets()</i>	<i>setpwent()</i>
<i>fgets()</i>	<i>getutxent()</i>	<i>setutxent()</i>
<i>fgetwc()</i>	<i>getutxid()</i>	<i>strerror()</i>
<i>fgetws()</i>	<i>getutxline()</i>	<i>syslog()</i>
<i>fopen()</i>	<i>getw()</i>	<i>tmpfile()</i>
<i>fprintf()</i>	<i>getwc()</i>	<i>tmpnam()</i>
<i>fputc()</i>	<i>getwchar()</i>	<i>ttynam()</i>
<i>fputs()</i>	<i>getwd()</i>	<i>ttynam_r()</i>
<i>fputwc()</i>	<i>glob()</i>	<i>ungetc()</i>
<i>fputws()</i>	<i>iconv_close()</i>	<i>ungetwc()</i>
<i>fread()</i>	<i>iconv_open()</i>	<i>unlink()</i>
<i>freopen()</i>	<i>ioctl()</i>	<i>vfprintf()</i>
<i>fscanf()</i>	<i>lseek()</i>	<i>vwprintf()</i>
<i>fseek()</i>	<i>mkstemp()</i>	<i>vprintf()</i>
<i>fseeko()</i>	<i>nftw()</i>	<i>vwprintf()</i>
<i>fsetpos()</i>	<i>opendir()</i>	<i>wprintf()</i>
<i>ftell()</i>	<i>openlog()</i>	<i>wscanf()</i>
<i>ftello()</i>	<i>pclose()</i>	
<i>ftw()</i>	<i>perror()</i>	

An implementation will not introduce cancellation points into any other functions specified in this specification.

The side effects of acting upon a cancellation request while suspended during a call of a function is the same as the side effects that may be seen in a single-threaded program when a call to a

² For any value of the *cmd* argument.

1997		function is interrupted by a signal and the given function returns [EINTR]. Any such side effects
1998		occur before any cancellation cleanup handlers are called.
1999		Whenever a thread has cancelability enabled and a cancellation request has been made with that
2000		thread as the target and the thread calls <i>pthread_testcancel()</i> , then the cancellation request is
2001		acted upon before <i>pthread_testcancel()</i> returns. If a thread has cancelability enabled and the
2002		thread has an asynchronous cancellation request pending and the thread is suspended at a
2003		cancellation point waiting for an event to occur, then the cancellation request will be acted upon.
2004		However, if the thread is suspended at a cancellation point and the event that it is waiting for
2005		occurs before the cancellation request is acted upon, it is unspecified whether the cancellation
2006		request is acted upon or whether the request remains pending and the thread resumes normal
2007		execution.
2008	2.8.8.3	Thread Cancellation Cleanup Handlers
2009		Each thread maintains a list of cancellation cleanup handlers. The programmer uses the
2010		functions <i>pthread_cleanup_push()</i> and <i>pthread_cleanup_pop()</i> to place routines on and remove
2011		routines from this list.
2012		When a cancellation request is acted upon, the routines in the list are invoked one by one in
2013		LIFO sequence; that is, the last routine pushed onto the list (Last In) is the first to be invoked
2014		(First Out). The thread invokes the cancellation cleanup handler with cancellation disabled until
2015		the last cancellation cleanup handler returns. When the cancellation cleanup handler for a scope
2016		is invoked, the storage for that scope remains valid. If the last cancellation cleanup handler
2017		returns, thread execution is terminated and a status of PTHREAD_CANCELED is made
2018		available to any threads joining with the target. The symbolic constant PTHREAD_CANCELED
2019		expands to a constant expression of type (void*) whose value matches no pointer to an object in
2020		memory nor the value NULL.
2021		The cancellation cleanup handlers are also invoked when the thread calls <i>pthread_exit()</i> .
2022		A side effect of acting upon a cancellation request while in a condition variable wait is that the
2023		mutex is reacquired before calling the first cancellation cleanup handler. In addition, the thread
2024		is no longer considered to be waiting for the condition and the thread will not have consumed
2025		any pending condition signals on the condition.
2026		A cancellation cleanup handler cannot exit via <i>longjmp()</i> or <i>siglongjmp()</i> .
2027	2.8.8.4	Async-Cancel Safety
2028		The <i>pthread_cancel()</i> , <i>pthread_setcancelstate()</i> and <i>pthread_setcanceltype()</i> functions are defined to
2029		be async-cancel safe.
2030		No other functions in this specification are required to be async-cancel safe.
2031	2.8.9	Thread Read-Write Locks
2032	EX	Multiple readers, single writer (read-write) locks allow many threads to have simultaneous
2033		read-only access to data while allowing only one thread to have write access at any given time.
2034		They are typically used to protect data that is read-only more frequently than it is changed.
2035		Read-write locks can be used to synchronise threads in the current process and other processes if
2036		they are allocated in memory that is writable and shared among the cooperating processes and
2037		have been initialised for this behaviour.

2.9 Data Types

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

Defined Type	Description
cc_t	Type used for terminal special characters.
clock_t	Arithmetic type used for processor times.
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 <i>div()</i> function.
FILE	A structure containing information about a file.
glob_t	Structure type used in pathname 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 a gid_t .
ino_t	Arithmetic type used for file serial numbers.
key_t	Arithmetic type used for interprocess communication.
ldiv_t	Structure type returned by <i>ldiv()</i> function.
mode_t	Arithmetic type used for file attributes.
mqd_t	Used for message queue descriptors.
nfds_t	Integral 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 initialisation.
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 integral 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	Integral type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.

2087			
2088			
2089		Defined Type	Description
2090		sigset_t	Integral or structure type of an object used to represent sets of signals.
2091		size_t	Unsigned integral type used for size of objects.
2092		speed_t	Type used for terminal baud rates.
2093	EX	ssize_t	Arithmetic type used for a count of bytes or an error indication.
2094		suseconds_t	A signed arithmetic type used for time in microseconds.
2095		tcflag_t	Type used for terminal modes.
2096	RT	time_t	Arithmetic type used for time in seconds.
2097		timer_t	Used for timer ID returned by <i>timer_create()</i> .
2098	EX	uid_t	Arithmetic type used for user IDs.
2099		useconds_t	Integral type used for time in microseconds.
2100		va_list	Type used for traversing variable argument lists.
2101		wchar_t	Integral type whose range of values can represent distinct codes for all members of the largest extended character set specified by the supported locales.
2102			
2103		wctype_t	Scalar type which represents a character class descriptor.
2104		wint_t	An integral type capable of storing any valid value of <i>wchar_t</i> , or WEOF .
2105			
2106		wordexp_t	Structure type used in word expansion.
2107			

System Interfaces

2108

2109

2110

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

2111 NAME

2112 a64l, l64a — convert between a 32-bit integer and a radix-64 ASCII string

2113 SYNOPSIS

2114 EX `#include <stdlib.h>`2115 `long a64l(const char *s);`2116 `char *l64a(long value);`

2117

2118 DESCRIPTION

2119 These functions are used to maintain numbers stored in radix-64 ASCII characters. This is a
 2120 notation by which 32-bit integers can be represented by up to six characters; each character
 2121 represents a digit in radix-64 notation. If the type **long** contains more than 32 bits, only the low-
 2122 order 32 bits are used for these operations.

2123 The characters used to represent ‘digits’ are ‘.’ for 0, ‘/’ for 1, ‘0’ through ‘9’ for 2–11,
 2124 ‘A’ through ‘Z’ for 12–37, and ‘a’ through ‘z’ for 38–63.

2125 The *a64l()* function takes a pointer to a radix-64 representation, in which the first digit is the
 2126 least significant, and returns a corresponding **long** value. If the string pointed to by *s* contains
 2127 more than six characters, *a64l()* uses the first six. If the first six characters of the string contain a
 2128 null terminator, *a64l()* uses only characters preceding the null terminator. The *a64l()* function
 2129 scans the character string from left to right with the least significant digit on the left, decoding
 2130 each character as a 6-bit radix-64 number. If the type **long** contains more than 32 bits, the
 2131 resulting value is sign-extended. The behaviour of *a64l()* is unspecified if *s* is a null pointer or
 2132 the string pointed to by *s* was not generated by a previous call to *l64a()*.

2133 The *l64a()* function takes a **long** argument and returns a pointer to the corresponding radix-64
 2134 representation. The behaviour of *l64a()* is unspecified if *value* is negative.

2135 The value returned by *l64a()* may be a pointer into a static buffer. Subsequent calls to *l64a()*
 2136 may overwrite the buffer.

2137 The *l64a()* interface need not be reentrant. An interface that is not required to be reentrant is not
 2138 required to be thread-safe.

2139 RETURN VALUE

2140 On successful completion, *a64l()* returns the **long** value resulting from conversion of the input
 2141 string. If a string pointed to by *s* is an empty string, *a64l()* returns 0L.

2142 The *l64a()* function returns a pointer to the radix-64 representation. If *value* is 0L, *l64a()* returns
 2143 a pointer to an empty string.

2144 ERRORS

2145 No errors are defined.

2146 EXAMPLES

2147 None.

2148 APPLICATION USAGE

2149 If the type **long** contains more than 32 bits, the result of *a64l(l64a(x))* is *x* in the low-order 32 bits.

2150 FUTURE DIRECTIONS

2151 None.

2152 SEE ALSO

2153 *strtoul()*, <stdlib.h>.

2154 **CHANGE HISTORY**

2155 First released in Issue 4, Version 2.

2156 **Issue 5**

2157 Moved from X/OPEN UNIX extension to BASE.

2158 Normative text previously in the APPLICATION USAGE section is moved to the
2159 DESCRIPTION.

2160 A note indicating that these interfaces need not be reentrant is added to the DESCRIPTION.

2161 **NAME**

2162 abort — generate an abnormal process abort

2163 **SYNOPSIS**

2164 #include <stdlib.h>

2165 void abort(void);

2166 **DESCRIPTION**

2167 The *abort()* function causes abnormal process termination to occur, unless the signal SIGABRT is
 2168 being caught and the signal handler does not return. The abnormal termination processing
 2169 includes at least the effect of *fclose()* on all open streams, and message catalogue descriptors, and
 2170 the default actions defined for SIGABRT. The SIGABRT signal is sent to the calling process as if
 2171 by means of *raise()* with the argument SIGABRT.

2172 The status made available to *wait()* or *waitpid()* by *abort()* will be that of a process terminated by
 2173 the SIGABRT signal. The *abort()* function will override blocking or ignoring the SIGABRT
 2174 signal.

2175 **RETURN VALUE**2176 The *abort()* function does not return.2177 **ERRORS**

2178 No errors are defined.

2179 **EXAMPLES**

2180 None.

2181 **APPLICATION USAGE**

2182 Catching the signal is intended to provide the application writer with a portable means to abort
 2183 processing, free from possible interference from any implementation-provided library functions.
 2184 If SIGABRT is neither caught nor ignored, and the current directory is writable, a core dump
 2185 may be produced.

2186 **FUTURE DIRECTIONS**

2187 None.

2188 **SEE ALSO**2189 *exit()*, *kill()*, *raise()*, *signal()*,2190 **CHANGE HISTORY**

2191 First released in Issue 1.

2192 Derived from Issue 1 of the SVID.

2193 **Issue 4**

2194 The following changes are incorporated in this issue for alignment with the ISO C standard and
 2195 the ISO POSIX-1 standard:

- 2196 • The argument list is explicitly defined as **void**.
- 2197 • The DESCRIPTION is revised to identify the correct order in which operations occur. It also
 2198 identifies:
 - 2199 — how the calling process is signalled
 - 2200 — how status information is made available to the host environment
 - 2201 — that *abort()* will override blocking or ignoring of the SIGABRT signal.

- 2202 Another change is incorporated as follows:
- 2203 • The APPLICATION USAGE section is replaced.

|

2204 **NAME**

2205 abs — return an integer absolute value

2206 **SYNOPSIS**

2207 #include <stdlib.h>

2208 int abs(int i);

2209 **DESCRIPTION**

2210 The *abs()* function computes the absolute value of its integer operand, *i*. If the result cannot be
2211 represented, the behaviour is undefined.

2212 **RETURN VALUE**

2213 The *abs()* function returns the absolute value of its integer operand.

2214 **ERRORS**

2215 No errors are defined.

2216 **EXAMPLES**

2217 None.

2218 **APPLICATION USAGE**

2219 In two's-complement representation, the absolute value of the negative integer with largest
2220 magnitude {INT_MIN} might not be representable.

2221 **FUTURE DIRECTIONS**

2222 None.

2223 **SEE ALSO**

2224 *fabs()*, *labs()*, <stdlib.h>.

2225 **CHANGE HISTORY**

2226 First released in Issue 1.

2227 Derived from Issue 1 of the SVID.

2228 **Issue 4**

2229 The following change is incorporated in this issue:

- 2230 • In the APPLICATION USAGE section, the phrase “{INT_MIN} is undefined” is replaced
2231 with “{INT_MIN} might not be representable”.

2232 **NAME**

2233 access — determine accessibility of a file

2234 **SYNOPSIS**

2235 #include <unistd.h>

2236 int access(const char *path, int amode);

2237 **DESCRIPTION**

2238 The `access()` function checks the file named by the pathname pointed to by the `path` argument for accessibility according to the bit pattern contained in `amode`, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID.

2241 The value of `amode` is either the bitwise inclusive OR of the access permissions to be checked (R_OK, W_OK, X_OK) or the existence test, F_OK.

2243 If any access permissions are to be checked, each will be checked individually, as described in the **XBD specification, Chapter 2, Definitions**. If the process has appropriate privileges, an implementation may indicate success for X_OK even if none of the execute file permission bits are set.

2247 **RETURN VALUE**

2248 If the requested access is permitted, `access()` succeeds and returns 0. Otherwise, -1 is returned and `errno` is set to indicate the error.

2250 **ERRORS**2251 The `access()` function will fail if:

2252 [EACCES] Permission bits of the file mode do not permit the requested access, or search permission is denied on a component of the path prefix.

2254 EX [ELOOP] Too many symbolic links were encountered in resolving `path`.

2255 FIPS [ENAMETOOLONG]

2256 The length of the `path` argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

2258 [ENOENT] A component of `path` does not name an existing file or `path` is an empty string.

2259 [ENOTDIR] A component of the path prefix is not a directory.

2260 [EROFS] Write access is requested for a file on a read-only file system.

2261 The `access()` function may fail if:

2262 [EINVAL] The value of the `amode` argument is invalid.

2263 EX [ENAMETOOLONG]

2264 Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

2266 EX [ETXTBSY] Write access is requested for a pure procedure (shared text) file that is being executed.

2268 **EXAMPLES**

2269 None.

2270 **APPLICATION USAGE**

2271 Additional values of `amode` other than the set defined in the description may be valid, for example, if a system has extended access controls.

2273 **FUTURE DIRECTIONS**

2274 None.

2275 **SEE ALSO**2276 `chmod()`, `stat()`, `<unistd.h>`.2277 **CHANGE HISTORY**

2278 First released in Issue 1.

2279 Derived from Issue 1 of the SVID.

2280 **Issue 4**

2281 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 2282
- The type of argument *path* is changed from **char *** to **const char ***.

2283 The following change is incorporated for alignment with the FIPS requirements:

- 2284
- In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a
2285 pathname component is larger than {NAME_MAX} is now defined as mandatory and marked
2286 as an extension.

2287 **Issue 4, Version 2**

2288 The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- 2289
- It states that [ELOOP] will be returned if too many symbolic links are encountered during
2290 pathname resolution.
 - A second [ENAMETOOLONG] condition is defined that may report excessive length of an
2291 intermediate result of pathname resolution of a symbolic link.
2292

2293 **NAME**

2294 acos — arc cosine function

2295 **SYNOPSIS**

2296 #include <math.h>

2297 double acos(double x);

2298 **DESCRIPTION**2299 The *acos()* function computes the principal value of the arc cosine of *x*. The value of *x* should be
2300 in the range $[-1,1]$.2301 An application wishing to check for error situations should set *errno* to 0 before calling *acos()*. If
2302 *errno* is non-zero on return, or the value NaN is returned, an error has occurred.2303 **RETURN VALUE**2304 Upon successful completion, *acos()* returns the arc cosine of *x*, in the range $[0, \pi]$ radians. If the
2305 EX value of *x* is not in the range $[-1,1]$, and is not $\pm\text{Inf}$ or NaN, either 0.0 or NaN is returned and
2306 *errno* is set to [EDOM].2307 EX If *x* is NaN, NaN is returned and *errno* may be set to [EDOM]. If *x* is $\pm\text{Inf}$, either 0.0 is returned
2308 and *errno* is set to [EDOM], or NaN is returned and *errno* may be set to [EDOM].2309 **ERRORS**2310 The *acos()* function will fail if:2311 EX [EDOM] The value *x* is not $\pm\text{Inf}$ or NaN and is not in the range $[-1,1]$.2312 The *acos()* function may fail if:2313 EX [EDOM] The value *x* is $\pm\text{Inf}$ or NaN.

2314 EX No other errors will occur.

2315 **EXAMPLES**

2316 None.

2317 **APPLICATION USAGE**

2318 None.

2319 **FUTURE DIRECTIONS**

2320 None.

2321 **SEE ALSO**2322 *cos()*, *isnan()*, <math.h>.2323 **CHANGE HISTORY**

2324 First released in Issue 1.

2325 Derived from Issue 1 of the SVID.

2326 **Issue 4**

2327 The following changes are incorporated in this issue:

- 2328 • Removed references to *matherr()*.
- 2329 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
- 2330 the ISO C standard and to rationalise error handling in the mathematics functions.
- 2331 • The return value specified for [EDOM] is marked as an extension.

2332 **Issue 5**

2333 The DESCRIPTION is updated to indicate how an application should check for an error. This
2334 text was previously published in the APPLICATION USAGE section.

2335 **NAME**

2336 acosh, asinh, atanh — inverse hyperbolic functions

2337 **SYNOPSIS**

2338 EX #include <math.h>

2339 double acosh(double x);

2340 double asinh(double x);

2341 double atanh(double x);

2342

2343 **DESCRIPTION**2344 The *acosh()*, *asinh()* and *atanh()* functions compute the inverse hyperbolic cosine, sine, and
2345 tangent of their argument, respectively.2346 **RETURN VALUE**2347 The *acosh()*, *asinh()* and *atanh()* functions return the inverse hyperbolic cosine, sine, and tangent
2348 of their argument, respectively.2349 The *acosh()* function returns an implementation-dependent value (NaN or equivalent if
2350 available) and sets *errno* to [EDOM] when its argument is less than 1.0.2351 The *atanh()* function returns an implementation-dependent value (NaN or equivalent if
2352 available) and sets *errno* to [EDOM] when its argument has absolute value greater than 1.0.2353 If *x* is NaN, the *asinh()*, *acosh()* and *atanh()* functions return NaN and may set *errno* to [EDOM].2354 **ERRORS**2355 The *acosh()* function will fail if:2356 [EDOM] The *x* argument is less than 1.0.2357 The *atanh()* function will fail if:2358 [EDOM] The *x* argument has an absolute value greater than 1.0.2359 The *atanh()* function will fail if:2360 [ERANGE] The *x* argument has an absolute value equal to 1.02361 The *asinh()*, *acosh()* and *atanh()* functions may fail if:2362 [EDOM] The value of *x* is NaN.2363 **EXAMPLES**

2364 None.

2365 **APPLICATION USAGE**

2366 None.

2367 **FUTURE DIRECTIONS**

2368 None.

2369 **SEE ALSO**2370 *cosh()*, *sinh()*, *tanh()*, <math.h>.2371 **CHANGE HISTORY**

2372 First released in Issue 4, Version 2.

2373 **Issue 5**

2374 Moved from X/OPEN UNIX extension to BASE.

2375 **NAME**2376 advance — pattern match given a compiled regular expression (**LEGACY**)2377 **SYNOPSIS**2378 EX `#include <regex.h>`2379 `int advance(const char *string, const char *expbuf);`

2380

2381 **DESCRIPTION**2382 Refer to *regex()*.2383 **CHANGE HISTORY**

2384 First released in Issue 2.

2385 Derived from Issue 2 of the SVID.

2386 **Issue 4**

2387 The following changes are incorporated in this issue:

- 2388 • The **<regex.h>** header is added to the SYNOPSIS section.
- 2389 • The type of arguments *string* and *expbuf* are changed from **char *** to **const char ***.
- 2390 • The interface is marked TO BE WITHDRAWN, because improved functionality is now
- 2391 provided by interfaces introduced for alignment with the ISO POSIX-2 standard.

2392 **Issue 5**

2393 Marked LEGACY.

2394 **NAME**2395 aio_cancel — cancel an asynchronous I/O request (**REALTIME**)2396 **SYNOPSIS**2397 RT `#include <aio.h>`2398 `int aio_cancel(int fildes, struct aiocb *aiocbp);`

2399

2400 **DESCRIPTION**

2401 The *aio_cancel()* function attempts to cancel one or more asynchronous I/O requests currently
 2402 outstanding against file descriptor *fildes*. The *aiocbp* argument points to the asynchronous I/O
 2403 control block for a particular request to be canceled. If *aiocbp* is NULL, then all outstanding
 2404 cancelable asynchronous I/O requests against *fildes* are canceled.

2405 Normal asynchronous notification occurs for asynchronous I/O operations that are successfully
 2406 canceled. If there are requests that cannot be canceled, then the normal asynchronous
 2407 completion process takes place for those requests when they are completed.

2408 For requested operations that are successfully canceled, the associated error status is set to
 2409 [ECANCELED] and the return status is -1. For requested operations that are not successfully
 2410 canceled, the *aiocbp* is not modified by *aio_cancel()*.

2411 If *aiocbp* is not NULL, then if *fildes* does not have the same value as the file descriptor with which
 2412 the asynchronous operation was initiated, unspecified results occur.

2413 Which operations are cancelable is implementation-dependent.

2414 **RETURN VALUE**

2415 The *aio_cancel()* function returns the value AIO_CANCELED to the calling process if the
 2416 requested operation(s) were canceled. The value AIO_NOTCANCELED is returned if at least
 2417 one of the requested operation(s) cannot be canceled because it is in progress. In this case, the
 2418 state of the other operations, if any, referenced in the call to *aio_cancel()* is not indicated by the
 2419 return value of *aio_cancel()*. The application may determine the state of affairs for these
 2420 operations by using *aio_error()*. The value AIO_ALLDONE is returned if all of the operations
 2421 have already completed. Otherwise, the function returns -1 and sets *errno* to indicate the error.

2422 **ERRORS**

2423 The *aio_cancel()* function will fail if:

2424 [EBADF] The *fildes* argument is not a valid file descriptor.

2425 [ENOSYS] The *aio_cancel()* function is not supported by this implementation.

2426 **EXAMPLES**

2427 None.

2428 **APPLICATION USAGE**

2429 None.

2430 **FUTURE DIRECTIONS**

2431 None.

2432 **SEE ALSO**

2433 *aio_read()*, *aio_write()*.

2434 **CHANGE HISTORY**

2435 First released in Issue 5.

2436 Included for alignment with the POSIX Realtime Extension.

2437 **NAME**2438 aio_error — retrieve errors status for an asynchronous I/O operation (**REALTIME**)2439 **SYNOPSIS**2440 RT `#include <aio.h>`2441 `int aio_error(const struct aiocb *aiocbp);`

2442

2443 **DESCRIPTION**

2444 The *aio_error()* function returns the error status associated with the **aiocb** structure referenced
 2445 by the *aiocbp* argument. The error status for an asynchronous I/O operation is the *errno* value
 2446 that would be set by the corresponding *read()*, *write()*, or *fsync()* operation. If the operation has
 2447 not yet completed, then the error status will be equal to *EINPROGRESS*.

2448 **RETURN VALUE**

2449 If the asynchronous I/O operation has completed successfully, then 0 is returned. If the
 2450 asynchronous operation has completed unsuccessfully, then the error status, as described for
 2451 *read()*, *write()*, and *fsync()*, is returned. If the asynchronous I/O operation has not yet
 2452 completed, then *EINPROGRESS* is returned.

2453 **ERRORS**2454 The *aio_error()* function will fail if:2455 *[ENOSYS]* The *aio_error()* function is not supported by this implementation.2456 The *aio_error()* function may fail if:

2457 *[EINVAL]* The *aiocbp* argument does not refer to an asynchronous operation whose
 2458 return status has not yet been retrieved.

2459 **EXAMPLES**

2460 None.

2461 **APPLICATION USAGE**

2462 None.

2463 **FUTURE DIRECTIONS**

2464 None.

2465 **SEE ALSO**

2466 *aio_read()*, *aio_write()*, *aio_fsync()*, *lio_listio()*, *aio_return()*, *aio_cancel()*, *read()*, *lseek()*, *close()*,
 2467 *_exit()*, *exec*, *fork()*.

2468 **CHANGE HISTORY**

2469 First released in Issue 5.

2470 Included for alignment with the POSIX Realtime Extension.

2471 **NAME**2472 aio_fsync — asynchronous file synchronisation (**REALTIME**)2473 **SYNOPSIS**2474 RT `#include <aio.h>`2475 `int aio_fsync(int op, struct aiocb *aiocbp);`

2476

2477 **DESCRIPTION**

2478 The *aio_fsync()* function asynchronously forces all I/O operations associated with the file
 2479 indicated by the file descriptor *aio_fildes* member of the **aiocb** structure referenced by the *aiocbp*
 2480 argument and queued at the time of the call to *aio_fsync()* to the synchronised I/O completion
 2481 state. The function call returns when the synchronisation request has been initiated or queued to
 2482 the file or device (even when the data cannot be synchronised immediately).

2483 If *op* is O_DSYNC, all currently queued I/O operations are completed as if by a call to
 2484 *fdatasync()*; that is, as defined for synchronised I/O data integrity completion. If *op* is O_SYNC,
 2485 all currently queued I/O operations are completed as if by a call to *fsync()*; that is, as defined for
 2486 synchronised I/O file integrity completion. If the *aio_fsync()* function fails, or if the operation
 2487 queued by *aio_fsync()* fails, then, as for *fsync()* and *fdatasync()*, outstanding I/O operations are
 2488 not guaranteed to have been completed.

2489 If *aio_fsync()* succeeds, then it is only the I/O that was queued at the time of the call to
 2490 *aio_fsync()* that is guaranteed to be forced to the relevant completion state. The completion of
 2491 subsequent I/O on the file descriptor is not guaranteed to be completed in a synchronised
 2492 fashion.

2493 The *aiocbp* argument refers to an asynchronous I/O control block. The *aiocbp* value may be used
 2494 as an argument to *aio_error()* and *aio_return()* in order to determine the error status and return
 2495 status, respectively, of the asynchronous operation while it is proceeding. When the request is
 2496 queued, the error status for the operation is EINPROGRESS. When all data has been
 2497 successfully transferred, the error status will be reset to reflect the success or failure of the
 2498 operation. If the operation does not complete successfully, the error status for the operation will
 2499 be set to indicate the error. The *aio_sigevent* member determines the asynchronous notification
 2500 to occur as specified in **Signal Generation and Delivery** on page 808 when all operations have
 2501 achieved synchronised I/O completion. All other members of the structure referenced by *aiocbp*
 2502 are ignored. If the control block referenced by *aiocbp* becomes an illegal address prior to
 2503 asynchronous I/O completion, then the behaviour is undefined.

2504 If the *aio_fsync()* function fails or the *aiocbp* indicates an error condition, data is not guaranteed
 2505 to have been successfully transferred.

2506 If *aiocbp* is NULL, then no status is returned in *aiocbp*, and no signal is generated upon
 2507 completion of the operation.

2508 **RETURN VALUE**

2509 The *aio_fsync()* function returns the value 0 to the calling process if the I/O operation is
 2510 successfully queued; otherwise, the function returns the value -1 and sets *errno* to indicate the
 2511 error.

2512 **ERRORS**2513 The *aio_fsync()* function will fail if:

2514 [EAGAIN] The requested asynchronous operation was not queued due to temporary
 2515 resource limitations.

2516	[EBADF]	The <i>aio_fildes</i> member of the aiocb structure referenced by the <i>aiocb</i> argument	
2517		is not a valid file descriptor open for writing.	
2518	[EINVAL]	This implementation does not support synchronised I/O for this file.	
2519	[EINVAL]	A value of <i>op</i> other than O_DSYNC or O_SYNC was specified.	
2520	[ENOSYS]	The <i>aio_fsync()</i> function is not supported by this implementation.	
2521		In the event that any of the queued I/O operations fail, <i>aio_fsync()</i> returns the error condition	
2522		defined for <i>read()</i> and <i>write()</i> . The error will be returned in the error status for the asynchronous	
2523		<i>fsync()</i> operation, which can be retrieved using <i>aio_error()</i> .	
2524	EXAMPLES		
2525		None.	
2526	APPLICATION USAGE		
2527		None.	
2528	FUTURE DIRECTIONS		
2529		None.	
2530	SEE ALSO		
2531		<i>fcntl()</i> , <i>fdatasync()</i> , <i>fsync()</i> , <i>open()</i> , <i>read()</i> , <i>write()</i> .	
2532	CHANGE HISTORY		
2533		First released in Issue 5.	
2534		Included for alignment with the POSIX Realtime Extension.	

2535 **NAME**2536 aio_read — asynchronous read from a file (**REALTIME**)2537 **SYNOPSIS**

2538 RT #include <aio.h>

2539 int aio_read(struct aiocb *aiocbp);

2540

2541 **DESCRIPTION**

2542 The *aio_read()* function allows the calling process to read *aiocbp->aio_nbytes* from the file
 2543 associated with *aiocbp->aio_fildes* into the buffer pointed to by *aiocbp->aio_buf*. The function call
 2544 returns when the read request has been initiated or queued to the file or device (even when the
 2545 data cannot be delivered immediately). If **_POSIX_PRIORITIZED_IO** is defined and prioritized
 2546 I/O is supported for this file, then the asynchronous operation is submitted at a priority equal to
 2547 the scheduling priority of the process minus *aiocbp->aio_reqprio*. The *aiocbp* value may be used as
 2548 an argument to *aio_error()* and *aio_return()* in order to determine the error status and return
 2549 status, respectively, of the asynchronous operation while it is proceeding. If an error condition is
 2550 encountered during queuing, the function call returns without having initiated or queued the
 2551 request. The requested operation takes place at the absolute position in the file as given by
 2552 *aio_offset*, as if *lseek()* were called immediately prior to the operation with an *offset* equal to
 2553 *aio_offset* and a *whence* equal to **SEEK_SET**. After a successful call to enqueue an asynchronous
 2554 I/O operation, the value of the file offset for the file is unspecified.

2555 The *aiocbp->aio_lio_opcode* field is ignored by *aio_read()*.

2556 The *aiocbp* argument points to an **aiocb** structure. If the buffer pointed to by *aiocbp->aio_buf* or
 2557 the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O
 2558 completion, then the behaviour is undefined.

2559 Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

2560 If **_POSIX_SYNCHRONIZED_IO** is defined and synchronised I/O is enabled on the file
 2561 associated with *aiocbp->aio_fildes*, the behaviour of this function is according to the definitions of
 2562 synchronised I/O data integrity completion and synchronised I/O file integrity completion.

2563 For any system action that changes the process memory space while an asynchronous I/O is
 2564 outstanding to the address range being changed, the result of that action is undefined.

2565 EX For regular files, no data transfer will occur past the offset maximum established in the open file
 2566 description associated with *aiocbp->aio_fildes*.

2567 **RETURN VALUE**

2568 The *aio_read()* function returns the value zero to the calling process if the I/O operation is
 2569 successfully queued; otherwise, the function returns the value **-1** and sets *errno* to indicate the
 2570 error.

2571 **ERRORS**2572 The *aio_read()* function will fail if:

2573 [EAGAIN] The requested asynchronous I/O operation was not queued due to system
 2574 resource limitations.

2575 [ENOSYS] The *aio_read()* function is not supported by this implementation.

2576 Each of the following conditions may be detected synchronously at the time of the call to
 2577 *aio_read()*, or asynchronously. If any of the conditions below are detected synchronously, the
 2578 *aio_read()* function returns **-1** and sets *errno* to the corresponding value. If any of the conditions
 2579 below are detected asynchronously, the return status of the asynchronous operation is set to **-1**,

2580		and the error status of the asynchronous operation will be set to the corresponding value.	
2581	[EBADF]	The <i>aiocbp->aio_fildes</i> argument is not a valid file descriptor open for reading.	
2582	[EINVAL]	The file offset value implied by <i>aiocbp->aio_offset</i> would be invalid, <i>aiocbp->aio_reqprio</i> is not a valid value, or <i>aiocbp->aio_nbytes</i> is an invalid value.	
2584		In the case that the <i>aio_read()</i> successfully queues the I/O operation but the operation is	
2585		subsequently canceled or encounters an error, the return status of the asynchronous operation is	
2586		one of the values normally returned by the <i>read()</i> function call. In addition, the error status of	
2587		the asynchronous operation will be set to one of the error statuses normally set by the <i>read()</i>	
2588		function call, or one of the following values:	
2589	[EBADF]	The <i>aiocbp->aio_fildes</i> argument is not a valid file descriptor open for reading.	
2590	[ECANCELED]	The requested I/O was canceled before the I/O completed due to an explicit	
2591		<i>aio_cancel()</i> request.	
2592	[EINVAL]	The file offset value implied by <i>aiocbp->aio_offset</i> would be invalid.	
2593	EX	The following condition may be detected synchronously or asynchronously:	
2594	[EOVERFLOW]	The file is a regular file, <i>aiocbp->aio_nbytes</i> is greater than 0 and the starting	
2595		offset in <i>aiocbp->aio_offset</i> is before the end-of-file and is at or beyond the offset	
2596		maximum in the open file description associated with <i>aiocbp->aio_fildes</i> .	
2597	EXAMPLES		
2598		None.	
2599	APPLICATION USAGE		
2600		None.	
2601	FUTURE DIRECTIONS		
2602		None.	
2603	SEE ALSO		
2604		<i>aio_cancel()</i> , <i>aio_error()</i> , <i>lio_listio()</i> , <i>aio_return()</i> , <i>aio_write()</i> , <i>close()</i> , <i>_exit()</i> , <i>exec</i> , <i>fork()</i> , <i>lseek()</i> ,	
2605		<i>read()</i> .	
2606	CHANGE HISTORY		
2607		First released in Issue 5.	
2608		Included for alignment with the POSIX Realtime Extension. Large File Summit extensions	
2609		added.	

2610 **NAME**2611 aio_return — retrieve return status of an asynchronous I/O operation (**REALTIME**)2612 **SYNOPSIS**2613 RT `#include <aio.h>`2614 `ssize_t aio_return(struct aiocb *aiocbp);`

2615

2616 **DESCRIPTION**

2617 The *aio_return()* function returns the return status associated with the **aiocb** structure referenced
 2618 by the *aiocbp* argument. The return status for an asynchronous I/O operation is the value that
 2619 would be returned by the corresponding *read()*, *write()*, or *fsync()* function call. If the error
 2620 status for the operation is equal to EINPROGRESS, then the return status for the operation is
 2621 undefined. The *aio_return()* function may be called exactly once to retrieve the return status of a
 2622 given asynchronous operation; thereafter, if the same **aiocb** structure is used in a call to
 2623 *aio_return()* or *aio_error()*, an error may be returned. When the **aiocb** structure referred to by
 2624 *aiocbp* is used to submit another asynchronous operation, then *aio_return()* may be successfully
 2625 used to retrieve the return status of that operation.

2626 **RETURN VALUE**

2627 If the asynchronous I/O operation has completed, then the return status, as described for *read()*,
 2628 *write()*, and *fsync()*, is returned. If the asynchronous I/O operation has not yet completed, the
 2629 results of *aio_return()* are undefined.

2630 **ERRORS**2631 The *aio_return()* function will fail if:

2632 [EINVAL] The *aiocbp* argument does not refer to an asynchronous operation whose
 2633 return status has not yet been retrieved.

2634 [ENOSYS] The *aio_return()* function is not supported by this implementation.

2635 **EXAMPLES**

2636 None.

2637 **APPLICATION USAGE**

2638 None.

2639 **FUTURE DIRECTIONS**

2640 None.

2641 **SEE ALSO**

2642 *aio_cancel()*, *aio_error()*, *aio_fsync()*, *aio_read()*, *aio_write()*, *close()*, *_exit()*, *exec*, *fork()*, *lio_listio()*,
 2643 *lseek()*, *read()*.

2644 **CHANGE HISTORY**

2645 First released in Issue 5.

2646 Included for alignment with the POSIX Realtime Extension.

2647 **NAME**2648 aio_suspend — wait for an asynchronous I/O request (**REALTIME**)2649 **SYNOPSIS**2650 RT `#include <aio.h>`

```
2651 int aio_suspend(const struct aiocb * const list[], int nent,
2652                const struct timespec *timeout);
2653
```

2654 **DESCRIPTION**

2655 The *aio_suspend()* function suspends the calling thread until at least one of the asynchronous
 2656 I/O operations referenced by the *list* argument has completed, until a signal interrupts the
 2657 function, or, if *timeout* is not NULL, until the time interval specified by *timeout* has passed. If any
 2658 of the **aiocb** structures in the list correspond to completed asynchronous I/O operations (that is,
 2659 the error status for the operation is not equal to EINPROGRESS) at the time of the call, the
 2660 function returns without suspending the calling thread. The *list* argument is an array of pointers
 2661 to asynchronous I/O control blocks. The *nent* argument indicates the number of elements in the
 2662 array. Each **aiocb** structure pointed to will have been used in initiating an asynchronous I/O
 2663 request via *aio_read()*, *aio_write()*, or *lio_listio()*. This array may contain NULL pointers, which
 2664 are ignored. If this array contains pointers that refer to **aiocb** structures that have not been used
 2665 in submitting asynchronous I/O, the effect is undefined.

2666 If the time interval indicated in the **timespec** structure pointed to by *timeout* passes before any of
 2667 the I/O operations referenced by *list* are completed, then *aio_suspend()* returns with an error.

2668 **RETURN VALUE**

2669 If the *aio_suspend()* function returns after one or more asynchronous I/O operations have
 2670 completed, the function returns zero. Otherwise, the function returns a value of -1 and sets
 2671 *errno* to indicate the error.

2672 The application may determine which asynchronous I/O completed by scanning the associated
 2673 error and return status using *aio_error()* and *aio_return()*, respectively.

2674 **ERRORS**2675 The *aio_suspend()* function will fail if:

2676 [EAGAIN] No asynchronous I/O indicated in the list referenced by *list* completed in the
 2677 time interval indicated by *timeout*.

2678 [EINTR] A signal interrupted the *aio_suspend()* function. Note that, since each
 2679 asynchronous I/O operation may possibly provoke a signal when it
 2680 completes, this error return may be caused by the completion of one (or more)
 2681 of the very I/O operations being awaited.

2682 [ENOSYS] The *aio_suspend()* function is not supported by this implementation.

2683 **EXAMPLES**

2684 None.

2685 **APPLICATION USAGE**

2686 None.

2687 **FUTURE DIRECTIONS**

2688 None.

2689 **SEE ALSO**2690 *aio_read()*, *aio_write()*, *lio_listio()*.

2691	CHANGE HISTORY	
2692	First released in Issue 5.	
2693	Included for alignment with the POSIX Realtime Extension.	

2694 NAME

2695 aio_write — asynchronous write to a file (**REALTIME**)

2696 SYNOPSIS

2697 RT `#include <aio.h>`2698 `int aio_write(struct aiocb *aiocbp);`

2699

2700 DESCRIPTION

2701 The *aio_write()* function allows the calling process to write *aiocbp->aio_nbytes* to the file
 2702 associated with *aiocbp->aio_fildes* from the buffer pointed to by *aiocbp->aio_buf*. The function call
 2703 returns when the write request has been initiated or, at a minimum, queued to the file or device.
 2704 If **_POSIX_PRIORITIZED_IO** is defined and prioritized I/O is supported for this file, then the
 2705 asynchronous operation is submitted at a priority equal to the scheduling priority of the process
 2706 minus *aiocbp->aio_reqprio*. The *aiocbp* may be used as an argument to *aio_error()* and *aio_return()*
 2707 in order to determine the error status and return status, respectively, of the asynchronous
 2708 operation while it is proceeding.

2709 The *aiocbp* argument points to an **aiocb** structure. If the buffer pointed to by *aiocbp->aio_buf* or
 2710 the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O
 2711 completion, then the behaviour is undefined.

2712 If **O_APPEND** is not set for the file descriptor *aio_fildes*, then the requested operation takes place
 2713 at the absolute position in the file as given by *aio_offset*, as if *lseek()* were called immediately
 2714 prior to the operation with an *offset* equal to *aio_offset* and a *whence* equal to **SEEK_SET**. If
 2715 **O_APPEND** is set for the file descriptor, write operations append to the file in the same order as
 2716 the calls were made. After a successful call to enqueue an asynchronous I/O operation, the
 2717 value of the file offset for the file is unspecified.

2718 The *aiocbp->aio_lio_opcode* field is ignored by *aio_write()*.

2719 Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

2720 If **_POSIX_SYNCHRONIZED_IO** is defined and synchronised I/O is enabled on the file
 2721 associated with *aiocbp->aio_fildes*, the behaviour of this function shall be according to the
 2722 definitions of synchronised I/O data integrity completion and synchronised I/O file integrity
 2723 completion.

2724 For any system action that changes the process memory space while an asynchronous I/O is
 2725 outstanding to the address range being changed, the result of that action is undefined.

2726 EX For regular files, no data transfer will occur past the offset maximum established in the open file
 2727 description associated with *aiocbp->aio_fildes*.

2728 RETURN VALUE

2729 The *aio_write()* function returns the value zero to the calling process if the I/O operation is
 2730 successfully queued; otherwise, the function returns the value -1 and sets *errno* to indicate the
 2731 error.

2732 ERRORS

2733 The *aio_write()* function will fail if:

2734 [EAGAIN] The requested asynchronous I/O operation was not queued due to system
 2735 resource limitations.

2736 [ENOSYS] The *aio_write()* function is not supported by this implementation.

2737 Each of the following conditions may be detected synchronously at the time of the call to
 2738 *aio_write()*, or asynchronously. If any of the conditions below are detected synchronously, the

2739 *aio_write()* function returns `-1` and sets *errno* to the corresponding value. If any of the conditions
 2740 below are detected asynchronously, the return status of the asynchronous operation is set to `-1`,
 2741 and the error status of the asynchronous operation will be set to the corresponding value.

2742 [EBADF] The *aioctx->aio_fildes* argument is not a valid file descriptor open for writing.

2743 [EINVAL] The file offset value implied by *aioctx->aio_offset* would be invalid, *aioctx->aio_reqprio*
 2744 is not a valid value, or *aioctx->aio_nbytes* is an invalid value.

2745 In the case that the *aio_write()* successfully queues the I/O operation, the return status of the
 2746 asynchronous operation will be one of the values normally returned by the *write()* function call.
 2747 If the operation is successfully queued but is subsequently canceled or encounters an error, the
 2748 error status for the asynchronous operation contains one of the values normally set by the
 2749 *write()* function call, or one of the following:

2750 [EBADF] The *aioctx->aio_fildes* argument is not a valid file descriptor open for writing.

2751 [EINVAL] The file offset value implied by *aioctx->aio_offset* would be invalid.

2752 [ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit
 2753 *aio_cancel()* request.

2754 EX The following condition may be detected synchronously or asynchronously:

2755 [EFBIG] The file is a regular file, *aioctx->aio_nbytes* is greater than 0 and the starting
 2756 offset in *aioctx->aio_offset* is at or beyond the offset maximum in the open file
 2757 description associated with *aioctx->aio_fildes*.

2758 EXAMPLES

2759 None.

2760 APPLICATION USAGE

2761 None.

2762 FUTURE DIRECTIONS

2763 None.

2764 SEE ALSO

2765 *aio_cancel()*, *aio_error()*, *aio_read()*, *aio_return()*, *lio_listio()*, *close()*, *_exit()*, *exec*, *fork()*, *lseek()*,
 2766 *write()*.

2767 CHANGE HISTORY

2768 First released in Issue 5.

2769 Included for alignment with the POSIX Realtime Extension. Large File Summit extensions
 2770 added.

2771 **NAME**

2772 alarm — schedule an alarm signal

2773 **SYNOPSIS**

2774 #include <unistd.h>

2775 unsigned int alarm(unsigned int *seconds*);2776 **DESCRIPTION**

2777 The *alarm()* function causes the system to generate a SIGALRM signal for the process after the
 2778 number of real-time seconds specified by *seconds* have elapsed. Processor scheduling delays
 2779 may prevent the process from handling the signal as soon as it is generated.

2780 If *seconds* is 0, a pending alarm request, if any, is cancelled.

2781 Alarm requests are not stacked; only one SIGALRM generation can be scheduled in this manner;
 2782 if the SIGALRM signal has not yet been generated, the call will result in rescheduling the time at
 2783 which the SIGALRM signal will be generated.

2784 **EX** Interactions between *alarm()* and any of *setitimer()*, *ualarm()* or *usleep()* are unspecified.

2785 **RETURN VALUE**

2786 If there is a previous *alarm()* request with time remaining, *alarm()* returns a non-zero value that
 2787 is the number of seconds until the previous request would have generated a SIGALRM signal.
 2788 Otherwise, *alarm()* returns 0.

2789 **ERRORS**

2790 The *alarm()* function is always successful, and no return value is reserved to indicate an error.

2791 **EXAMPLES**

2792 None.

2793 **APPLICATION USAGE**

2794 The *fork()* function clears pending alarms in the child process. A new process image created by
 2795 one of the *exec* functions inherits the time left to an alarm signal in the old process' image.

2796 **FUTURE DIRECTIONS**

2797 None.

2798 **SEE ALSO**

2799 *exec*, *fork()*, *getitimer()*, *pause()*, *sigaction()*, *ualarm()*, *usleep()*, <signal.h>, <unistd.h>.

2800 **CHANGE HISTORY**

2801 First released in Issue 1.

2802 Derived from Issue 1 of the SVID.

2803 **Issue 4**

2804 The following change is incorporated in this issue:

- 2805 • The header <unistd.h> is included in the SYNOPSIS section.

2806 **Issue 4, Version 2**

2807 The DESCRIPTION is updated to indicate that interactions with the *setitimer()*, *ualarm()* and
 2808 *usleep()* functions are unspecified.

2809 **NAME**

2810 asctime, asctime_r — convert date and time to a string

2811 **SYNOPSIS**

2812 #include <time.h>

2813 char *asctime(const struct tm *timeptr);

2814 char *asctime_r(const struct tm *tm, char *buf);

2815 **DESCRIPTION**2816 The *asctime()* function converts the broken-down time in the structure pointed to by *timeptr* into
2817 a string in the form:

2818 Sun Sep 16 01:03:52 1973\n\0

2819 using the equivalent of the following algorithm:

```

2820     char *asctime(const struct tm *timeptr)
2821     {
2822         static char wday_name[7][3] = {
2823             "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
2824         };
2825         static char mon_name[12][3] = {
2826             "Jan", "Feb", "Mar", "Apr", "May", "Jun",
2827             "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
2828         };
2829         static char result[26];
2830
2831         sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
2832             wday_name[timeptr->tm_wday],
2833             mon_name[timeptr->tm_mon],
2834             timeptr->tm_mday, timeptr->tm_hour,
2835             timeptr->tm_min, timeptr->tm_sec,
2836             1900 + timeptr->tm_year);
2837         return result;
2838     }

```

2838 The **tm** structure is defined in the <time.h> header.

2839 The *asctime()*, *ctime()*, *gmtime()* and *localtime()* functions return values in one of two static
2840 objects: a broken-down time structure and an array of type **char**. Execution of any of the
2841 functions may overwrite the information returned in either of these objects by any of the other
2842 functions.

2843 The *asctime()* interface need not be reentrant.

2844 The *asctime_r()* function converts the broken-down time in the structure pointed to by *tm* into a
2845 string that is placed in the user-supplied buffer pointed to by *buf* (which contains at least 26
2846 bytes) and then returns *buf*.

2847 **RETURN VALUE**2848 Upon successful completion, *asctime()* returns a pointer to the string.

2849 Upon successful completion, *asctime_r()* returns a pointer to a character string containing the
2850 date and time. This string is pointed to by the argument *buf*. If the function is unsuccessful, it
2851 returns NULL.

2852 **ERRORS**

2853 No errors are defined.

2854 **EXAMPLES**

2855 None.

2856 **APPLICATION USAGE**

2857 Values for the broken-down time structure can be obtained by calling *gmtime()* or *localtime()*.
 2858 This interface is included for compatibility with older implementations, and does not support
 2859 localised date and time formats. Applications should use *strftime()* to achieve maximum
 2860 portability.

2861 **FUTURE DIRECTIONS**

2862 None.

2863 **SEE ALSO**

2864 *clock()*, *ctime()*, *difftime()*, *gmtime()*, *localtime()*, *mktime()*, *strftime()*, *strptime()*, *time()*, *utime()*,
 2865 <time.h>.

2866 **CHANGE HISTORY**

2867 First released in Issue 1.

2868 Derived from Issue 1 of the SVID.

2869 **Issue 4**

2870 The following change is incorporated for alignment with the ISO C standard:

- 2871 • The type of argument *timeptr* is changed from **struct tm*** to **const struct tm***.

2872 Other changes are incorporated as follows:

- 2873 • The location of the **tm** structure is now defined.
- 2874 • The APPLICATION USAGE section is expanded to describe the time-handling functions
 2875 generally and to refer users to *strftime()*, which is a locale-dependent time-handling function.

2876 **Issue 5**

2877 Normative text previously in the APPLICATION USAGE section is moved to the
 2878 DESCRIPTION.

2879 The *asctime_r()* function is included for alignment with the POSIX Threads Extension.

2880 A note indicating that the *asctime()* interface need not be reentrant is added to the
 2881 DESCRIPTION.

2882 **NAME**2883 `asin` — arc sine function2884 **SYNOPSIS**2885 `#include <math.h>`2886 `double asin(double x);`2887 **DESCRIPTION**2888 The `asin()` function computes the principal value of the arc sine of x . The value of x should be in the range $[-1,1]$.2890 An application wishing to check for error situations should set `errno` to 0, then call `asin()`. If `errno` is non-zero on return, or the return value is NaN, an error has occurred.2892 **RETURN VALUE**2893 Upon successful completion, `asin()` returns the arc sine of x , in the range $[-\pi/2, \pi/2]$ radians. If the value of x is not in the range $[-1,1]$, and is not $\pm\text{Inf}$ or NaN, either 0.0 or NaN is returned and `errno` is set to [EDOM].2896 EX If x is NaN, NaN is returned and `errno` may be set to [EDOM].2897 EX If x is $\pm\text{Inf}$, either 0.0 is returned and `errno` is set to [EDOM] or NaN is returned and `errno` may be set to [EDOM].2899 If the result underflows, 0.0 is returned and `errno` may be set to [ERANGE].2900 **ERRORS**2901 The `asin()` function will fail if:2902 EX [EDOM] The value x is not $\pm\text{Inf}$ or NaN and is not in the range $[-1,1]$.2903 The `asin()` function may fail if:2904 EX [EDOM] The value of x is $\pm\text{Inf}$ or NaN.

2905 [ERANGE] The result underflows.

2906 EX No other errors will occur.

2907 **EXAMPLES**

2908 None.

2909 **APPLICATION USAGE**

2910 None.

2911 **FUTURE DIRECTIONS**

2912 None.

2913 **SEE ALSO**2914 `isnan()`, `sin()`, `<math.h>`.2915 **CHANGE HISTORY**

2916 First released in Issue 1.

2917 Derived from Issue 1 of the SVID.

2918 **Issue 4**

2919 The following changes are incorporated in this issue:

- 2920 • Removed references to *matherr()*.
- 2921 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
2922 the ISO C standard and to rationalise error handling in the mathematics functions.
- 2923 • The return value specified for [EDOM] is marked as an extension.

2924 **Issue 5**

2925 The DESCRIPTION is updated to indicate how an application should check for an error. This
2926 text was previously published in the APPLICATION USAGE section.

2927 **NAME**

2928 asinh — hyperbolic arc sine

2929 **SYNOPSIS**

2930 EX #include <math.h>

2931 double asinh(double x);

2932

2933 **DESCRIPTION**2934 Refer to *acosh()*.2935 **CHANGE HISTORY**

2936 First released in Issue 4, Version 2.

2937 **Issue 5**

2938 Moved from X/OPEN UNIX extension to BASE.

2939 **NAME**

2940 assert — insert program diagnostics

2941 **SYNOPSIS**

2942 #include <assert.h>

2943 void assert(int *expression*);2944 **DESCRIPTION**

2945 The *assert()* macro inserts diagnostics into programs. When it is executed, if *expression* is false
2946 (that is, compares equal to 0), *assert()* writes information about the particular call that failed
2947 (including the text of the argument, the name of the source file and the source file line number —
2948 the latter are respectively the values of the preprocessing macros `__FILE__` and `__LINE__`) on
2949 *stderr* and calls *abort()*.

2950 Forcing a definition of the name `NDEBUG`, either from the compiler command line or with the
2951 preprocessor control statement `#define NDEBUG` ahead of the `#include <assert.h>` statement,
2952 will stop assertions from being compiled into the program.

2953 **RETURN VALUE**2954 The *assert()* macro returns no value.2955 **ERRORS**

2956 No errors are defined.

2957 **EXAMPLES**

2958 None.

2959 **APPLICATION USAGE**

2960 None.

2961 **FUTURE DIRECTIONS**

2962 None.

2963 **SEE ALSO**2964 *abort()*, *stderr()*, <assert.h>.2965 **CHANGE HISTORY**

2966 First released in Issue 1.

2967 Derived from Issue 1 of the SVID.

2968 **Issue 4**

2969 The following change is incorporated in this issue:

- 2970
- The APPLICATION USAGE section is merged into the DESCRIPTION.

2971 **NAME**

2972 atan — arc tangent function

2973 **SYNOPSIS**

2974 #include <math.h>

2975 double atan(double x);

2976 **DESCRIPTION**2977 The *atan()* function computes the principal value of the arc tangent of *x*.2978 An application wishing to check for error situations should set *errno* to 0 before calling *atan()*. If
2979 *errno* is non-zero on return, or the return value is NaN, an error has occurred.2980 **RETURN VALUE**2981 Upon successful completion, *atan()* returns the arc tangent of *x* in the range $[-\pi/2, \pi/2]$ radians.2982 EX If *x* is NaN, NaN is returned and *errno* may be set to [EDOM].2983 If the result underflows, 0.0 is returned and *errno* may be set to [ERANGE].2984 **ERRORS**2985 The *atan()* function may fail if:2986 EX [EDOM] The value of *x* is NaN.

2987 [ERANGE] The result underflows.

2988 EX No other errors will occur.

2989 **EXAMPLES**

2990 None.

2991 **APPLICATION USAGE**

2992 None.

2993 **FUTURE DIRECTIONS**

2994 None.

2995 **SEE ALSO**2996 *atan2()*, *isnan()*, *tan()*, <math.h>.2997 **CHANGE HISTORY**

2998 First released in Issue 1.

2999 Derived from Issue 1 of the SVID.

3000 **Issue 4**

3001 The following changes are incorporated in this issue:

- 3002 • Removed references to *matherr()*.
- 3003 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
- 3004 the ISO C standard and to rationalise error handling in the mathematics functions.
- 3005 • The return value specified for [EDOM] is marked as an extension.

3006 **Issue 5**

3007 The DESCRIPTION is updated to indicate how an application should check for an error. This

3008 text was previously published in the APPLICATION USAGE section.

3009 **NAME**

3010 atan2 — arc tangent function

3011 **SYNOPSIS**

3012 #include <math.h>

3013 double atan2(double y, double x);

3014 **DESCRIPTION**3015 The *atan2()* function computes the principal value of the arc tangent of y/x , using the signs of
3016 both arguments to determine the quadrant of the return value.3017 An application wishing to check for error situations should set *errno* to 0 before calling *atan2()*.
3018 If *errno* is non-zero on return, or the return value is NaN, an error has occurred.3019 **RETURN VALUE**3020 Upon successful completion, *atan2()* returns the arc tangent of y/x in the range $[-\pi, \pi]$ radians. If
3021 both arguments are 0.0, an implementation-dependent value is returned and *errno* may be set to
3022 [EDOM].3023 EX If *x* or *y* is NaN, NaN is returned and *errno* may be set to [EDOM].3024 If the result underflows, 0.0 is returned and *errno* may be set to [ERANGE].3025 **ERRORS**3026 The *atan2()* function may fail if:

3027 EX [EDOM] Both arguments are 0.0 or one or more of the arguments is NaN.

3028 [ERANGE] The result underflows.

3029 EX No other errors will occur.

3030 **EXAMPLES**

3031 None.

3032 **APPLICATION USAGE**

3033 None.

3034 **FUTURE DIRECTIONS**

3035 None.

3036 **SEE ALSO**3037 *atan()*, *isnan()*, *tan()*, <math.h>.3038 **CHANGE HISTORY**

3039 First released in Issue 1.

3040 Derived from Issue 1 of the SVID.

3041 **Issue 4**

3042 The following changes are incorporated in this issue:

- 3043 • Removed references to *matherr()*.
- 3044 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
- 3045 the ISO C standard and to rationalise error handling in the mathematics functions.
- 3046 • The return value specified for [EDOM] is marked as an extension.

3047 **Issue 5**

3048 The DESCRIPTION is updated to indicate how an application should check for an error. This
3049 text was previously published in the APPLICATION USAGE section.

3050 NAME

3051 atanh — hyperbolic arc tangent

3052 SYNOPSIS

3053 EX #include <math.h>

3054 double atanh(double x);

3055

3056 DESCRIPTION

3057 Refer to *acosh()*.

3058 CHANGE HISTORY

3059 First released in Issue 4, Version 2.

3060 Issue 5

3061 Moved from X/OPEN UNIX extension to BASE.

3062 NAME

3063 atexit — register a function to run at process termination

3064 SYNOPSIS

3065 #include <stdlib.h>

3066 int atexit(void (**func*)(void));

3067 DESCRIPTION

3068 The *atexit()* function registers the function pointed to by *func* to be called without arguments. At
3069 normal process termination, functions registered by *atexit()* are called in the reverse order to that
3070 in which they were registered. Normal termination occurs either by a call to *exit()* or a return
3071 from *main()*.

3072 At least 32 functions can be registered with *atexit()*.

3073 After a successful call to any of the *exec* functions, any functions previously registered by *atexit()*
3074 are no longer registered.

3075 RETURN VALUE

3076 Upon successful completion, *atexit()* returns 0. Otherwise, it returns a non-zero value.

3077 ERRORS

3078 No errors are defined.

3079 EXAMPLES

3080 None.

3081 APPLICATION USAGE

3082 The functions registered by a call to *atexit()* must return to ensure that all registered functions
3083 are called.

3084 The application should call *sysconf()* to obtain the value of {ATEXIT_MAX}, the number of
3085 functions that can be registered. There is no way for an application to tell how many functions
3086 have already been registered with *atexit()*.

3087 FUTURE DIRECTIONS

3088 None.

3089 SEE ALSO

3090 *exit()*, *sysconf()*, <stdlib.h>.

3091 CHANGE HISTORY

3092 First released in Issue 4.

3093 Derived from the ANSI C standard.

3094 Issue 4, Version 2

3095 The APPLICATION USAGE section is updated to indicate how an application can determine the
3096 setting of {ATEXIT_MAX}, which is a constant added for X/OPEN UNIX conformance.

3097 **NAME**3098 *atof* — convert a string to double-precision number3099 **SYNOPSIS**3100 `#include <stdlib.h>`3101 `double atof(const char *str);`3102 **DESCRIPTION**3103 The call *atof(str)* is equivalent to:3104 `strtod(str, (char **)NULL),`3105 except that the handling of errors may differ. If the value cannot be represented, the behaviour
3106 is undefined.3107 **RETURN VALUE**3108 The *atof()* function returns the converted value if the value can be represented.3109 **ERRORS**

3110 No errors are defined.

3111 **EXAMPLES**

3112 None.

3113 **APPLICATION USAGE**3114 The *atof()* function is subsumed by *strtod()* but is retained because it is used extensively in
3115 existing code. If the number is not known to be in range, *strtod()* should be used because *atof()* is
3116 not required to perform any error checking.3117 **FUTURE DIRECTIONS**

3118 None.

3119 **SEE ALSO**3120 *strtod()*, `<stdlib.h>`.3121 **CHANGE HISTORY**

3122 First released in Issue 1.

3123 Derived from Issue 1 of the SVID.

3124 **Issue 4**

3125 The following change is incorporated for alignment with the ISO C standard:

- 3126
- The type of argument *str* is changed from **char *** to **const char ***.

3127 Other changes are incorporated as follows:

- 3128
- Reference to how *str* is converted is removed from the DESCRIPTION.
 - The APPLICATION USAGE section is added.

3130 **NAME**3131 *atoi* — convert a string to integer3132 **SYNOPSIS**

3133 #include <stdlib.h>

3134 int *atoi*(const char **str*);3135 **DESCRIPTION**3136 The call *atoi*(*str*) is equivalent to:3137 (int) *strtol*(*str*, (char **)NULL, 10)3138 except that the handling of errors may differ. If the value cannot be represented, the behaviour
3139 is undefined.3140 **RETURN VALUE**3141 The *atoi*() function returns the converted value if the value can be represented.3142 **ERRORS**

3143 No errors are defined.

3144 **EXAMPLES**

3145 None.

3146 **APPLICATION USAGE**3147 The *atoi*() function is subsumed by *strtol*() but is retained because it is used extensively in
3148 existing code. If the number is not known to be in range, *strtol*() should be used because *atoi*() is
3149 not required to perform any error checking.3150 **FUTURE DIRECTIONS**

3151 None.

3152 **SEE ALSO**3153 *strtol*(), <stdlib.h>.3154 **CHANGE HISTORY**

3155 First released in Issue 1.

3156 Derived from Issue 1 of the SVID.

3157 **Issue 4**

3158 The following change is incorporated for alignment with the ISO C standard:

- 3159
- The type of argument *str* is changed from **char *** to **const char ***.

3160 Other changes are incorporated as follows:

- 3161
- Reference to how *str* is converted is removed from the DESCRIPTION.
 - The APPLICATION USAGE section is added.

3163 **NAME**

3164 *atol* — convert a string to long integer

3165 **SYNOPSIS**

3166 `#include <stdlib.h>`

3167 `long int atol(const char *str);`

3168 **DESCRIPTION**

3169 The call *atol(str)* is equivalent to:

3170 `strtoul(str, (char **)NULL, 10)`

3171 except that the handling of errors may differ. If the value cannot be represented, the behaviour
3172 is undefined.

3173 **RETURN VALUE**

3174 The *atol()* function returns the converted value if the value can be represented.

3175 **ERRORS**

3176 No errors are defined.

3177 **EXAMPLES**

3178 None.

3179 **APPLICATION USAGE**

3180 The *atol()* function is subsumed by *strtoul()* but is retained because it is used extensively in
3181 existing code. If the number is not known to be in range, *strtoul()* should be used because *atol()* is
3182 not required to perform any error checking.

3183 **FUTURE DIRECTIONS**

3184 None.

3185 **SEE ALSO**

3186 *strtoul()*, `<stdlib.h>`.

3187 **CHANGE HISTORY**

3188 First released in Issue 1.

3189 Derived from Issue 1 of the SVID.

3190 **Issue 4**

3191 The following changes are incorporated for alignment with the ISO C standard:

- 3192 • The type of argument *str* is changed from **char *** to **const char ***.
- 3193 • The return type of the function is expanded to **long int**.

3194 Other changes are incorporated as follows:

- 3195 • Reference to how *str* is converted is removed from the DESCRIPTION.
- 3196 • The APPLICATION USAGE section is added.

3197 **NAME**3198 **basename** — return the last component of a pathname3199 **SYNOPSIS**

3200 EX #include <libgen.h>

3201 char *basename(char *path);

3202

3203 **DESCRIPTION**3204 The *basename()* function takes the pathname pointed to by *path* and returns a pointer to the final
3205 component of the pathname, deleting any trailing '/' characters.3206 If the string consists entirely of the '/' character, *basename()* returns a pointer to the string "/".3207 If *path* is a null pointer or points to an empty string, *basename()* returns a pointer to the string ".".3208 The *basename()* function may modify the string pointed to by *path*, and may return a pointer to
3209 static storage that may then be overwritten by a subsequent call to *basename()*.

3210 This interface need not be reentrant.

3211 **RETURN VALUE**3212 The *basename()* function returns a pointer to the final component of *path*.3213 **ERRORS**

3214 No errors are defined.

3215 **EXAMPLES**

3216

3217

3218

3219

3220

Input String	Output String
"/usr/lib"	"lib"
"/usr/"	"usr"
"/"	"/"

3221 **APPLICATION USAGE**

3222 None.

3223 **FUTURE DIRECTIONS**

3224 None.

3225 **SEE ALSO**3226 *dirname()*, <libgen.h>.3227 **CHANGE HISTORY**

3228 First released in Issue 4, Version 2.

3229 **Issue 5**

3230 Moved from X/OPEN UNIX extension to BASE.

3231 Normative text previously in the APPLICATION USAGE section is moved to the
3232 DESCRIPTION.

3233 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

3234 **NAME**

3235 bcmp — memory operations

3236 **SYNOPSIS**

3237 EX #include <strings.h>

3238 int bcmp(const void *s1, const void *s2, size_t n);

3239

3240 **DESCRIPTION**3241 The *bcmp()* function compares the first *n* bytes of the area pointed to by *s1* with the area pointed
3242 to by *s2*.3243 **RETURN VALUE**3244 The *bcmp()* function returns 0 if *s1* and *s2* are identical, non-zero otherwise. Both areas are
3245 assumed to be *n* bytes long. If the value of *n* is 0, *bcmp()* returns 0.3246 **ERRORS**

3247 No errors are defined.

3248 **EXAMPLES**

3249 None.

3250 **APPLICATION USAGE**3251 For portability to implementations conforming to earlier versions of this specification, *memcmp()*
3252 is preferred over this function.3253 **FUTURE DIRECTIONS**

3254 None.

3255 **SEE ALSO**3256 *memcmp()*, <strings.h>.3257 **CHANGE HISTORY**

3258 First released in Issue 4, Version 2.

3259 **Issue 5**

3260 Moved from X/OPEN UNIX extension to BASE.

3261 **NAME**

3262 bcopy — memory operations

3263 **SYNOPSIS**

3264 EX #include <strings.h>

3265 void bcopy(const void *s1, void *s2, size_t n);

3266

3267 **DESCRIPTION**3268 The *bcopy()* function copies *n* bytes from the area pointed to by *s1* to the area pointed to by *s2*.3269 The bytes are copied correctly even if the area pointed to by *s1* overlaps the area pointed to by
3270 *s2*.3271 **RETURN VALUE**3272 The *bcopy()* function returns no value.3273 **ERRORS**

3274 No errors are defined.

3275 **EXAMPLES**

3276 None.

3277 **APPLICATION USAGE**3278 For portability to implementations conforming to earlier versions of this specification,
3279 *memmove()* is preferred over this function.

3280 The following are approximately equivalent (note the order of the arguments):

3281 bcopy(s1,s2,n) ~ memmove(s2,s1,n)

3282 **FUTURE DIRECTIONS**

3283 None.

3284 **SEE ALSO**3285 *memmove()*, <strings.h>.3286 **CHANGE HISTORY**

3287 First released in Issue 4, Version 2.

3288 **Issue 5**

3289 Moved from X/OPEN UNIX extension to BASE.

3290 NAME

3291 brk, sbrk — change space allocation (LEGACY)

3292 SYNOPSIS

3293 EX `#include <unistd.h>`3294 `int brk(void *addr);`3295 `void *sbrk(intptr_t incr);`

3296

3297 DESCRIPTION

3298 The *brk()* and *sbrk()* functions are used to change the amount of space allocated for the calling
 3299 process. The change is made by resetting the process' break value and allocating the appropriate
 3300 amount of space. The amount of allocated space increases as the break value increases. The
 3301 newly-allocated space is set to 0. However, if the application first decrements and then
 3302 increments the break value, the contents of the reallocated space are unspecified.

3303 The *brk()* function sets the break value to *addr* and changes the allocated space accordingly.

3304 The *sbrk()* function adds *incr* bytes to the break value and changes the allocated space
 3305 accordingly. If *incr* is negative, the amount of allocated space is decreased by *incr* bytes. The
 3306 current value of the program break is returned by *sbrk(0)*.

3307 The behaviour of *brk()* and *sbrk()* is unspecified if an application also uses any other memory
 3308 functions (such as *malloc()*, *mmap()*, *free()*). Other functions may use these other memory
 3309 functions silently.

3310 It is unspecified whether the pointer returned by *sbrk()* is aligned suitably for any purpose.

3311 These interfaces need not be reentrant.

3312 RETURN VALUE

3313 Upon successful completion, *brk()* returns 0. Otherwise, it returns -1 and sets *errno* to indicate
 3314 the error.

3315 Upon successful completion, *sbrk()* returns the prior break value. Otherwise, it returns
 3316 (**void ***)-1 and sets *errno* to indicate the error.

3317 ERRORS

3318 The *brk()* and *sbrk()* functions will fail if:

3319 [ENOMEM] The requested change would allocate more space than allowed.

3320 The *brk()* and *sbrk()* functions may fail if:

3321 [EAGAIN] The total amount of system memory available for allocation to this process is
 3322 temporarily insufficient. This may occur even though the space requested
 3323 was less than the maximum data segment size.

3324 [ENOMEM] The requested change would be impossible as there is insufficient swap space
 3325 available, or would cause a memory allocation conflict.

3326 EXAMPLES

3327 None.

3328 APPLICATION USAGE

3329 The *brk()* and *sbrk()* functions have been used in specialised cases where no other memory
 3330 allocation function provided the same capability. The use of *malloc()* is now preferred because it
 3331 can be used portably with all other memory allocation functions and with any function that uses
 3332 other allocation functions.

3333 **FUTURE DIRECTIONS**

3334 None.

3335 **SEE ALSO**3336 *exec*, *malloc()*, *mmap()*, **<unistd.h>**.3337 **CHANGE HISTORY**

3338 First released in Issue 4, Version 2.

3339 **Issue 5**

3340 Moved from X/OPEN UNIX extension to BASE.

3341 Normative text previously in the APPLICATION USAGE section is moved to the
3342 DESCRIPTION.

3343 Marked LEGACY.

3344 The type of the argument to *sbrk()* is changed from **int** to **intptr_t**.

3345 A note indicating that these interfaces need not be reentrant is added to the DESCRIPTION.

3346 **NAME**

3347 bsd_signal — simplified signal facilities

3348 **SYNOPSIS**

3349 EX #include <signal.h>

3350 void (*bsd_signal(int *sig*, void (*func)(int)))(int);

3351

3352 **DESCRIPTION**3353 The *bsd_signal()* function provides a partially compatible interface for programs written to
3354 historical system interfaces (see APPLICATION USAGE below).3355 The function call *bsd_signal(sig, func)* has an effect as if implemented as:

```

3356       void (*bsd_signal(int sig, void (*func)(int)))(int)
3357       {
3358           struct sigaction act, oact;
3359           act.sa_handler = func;
3360           act.sa_flags = SA_RESTART;
3361           sigemptyset(&act.sa_mask);
3362           sigaddset(&act.sa_mask, sig);
3363           if (sigaction(sig, &act, &oact) == -1)
3364               return(SIG_ERR);
3365           return(oact.sa_handler);
3366       }

```

3367 The handler function should be declared:

3368 void handler(int *sig*);3369 where *sig* is the signal number. The behaviour is undefined if *func* is a function that takes more
3370 than one argument, or an argument of a different type.3371 **RETURN VALUE**3372 Upon successful completion, *bsd_signal()* returns the previous action for *sig*. Otherwise,
3373 SIG_ERR is returned and *errno* is set to indicate the error.3374 **ERRORS**3375 Refer to *sigaction()*.3376 **EXAMPLES**

3377 None.

3378 **APPLICATION USAGE**

3379 This function is a direct replacement for the BSD *signal()* function for simple applications that
3380 are installing a single-argument signal handler function. If a BSD signal handler function is
3381 being installed that expects more than one argument, the application has to be modified to use
3382 *sigaction()*. The *bsd_signal()* function differs from *signal()* in that the SA_RESTART flag is set
3383 and the SA_RESETHAND will be clear when *bsd_signal()* is used. The state of these flags is not
3384 specified for *signal()*.

3385 **FUTURE DIRECTIONS**

3386 None.

3387 **SEE ALSO**3388 *sigaction()*, *sigaddset()*, *sigemptyset()*, *signal()*, <signal.h>.

3389 **CHANGE HISTORY**

3390 First released in Issue 4, Version 2.

3391 **Issue 5**

3392 Moved from X/OPEN UNIX extension to BASE.

|
|

3393 **NAME**

3394 bsearch — binary search a sorted table

3395 **SYNOPSIS**

3396 #include <stdlib.h>

```
3397       void *bsearch(const void *key, const void *base, size_t nel,
3398                   size_t width, int (*compar)(const void *, const void *));
```

3399 **DESCRIPTION**

3400 The *bsearch()* function searches an array of *nel* objects, the initial element of which is pointed to
 3401 by *base*, for an element that matches the object pointed to by *key*. The size of each element in the
 3402 array is specified by *width*.

3403 The comparison function pointed to by *compar* is called with two arguments that point to the *key*
 3404 object and to an array element, in that order.

3405 The function must return an integer less than, equal to, or greater than 0 if the *key* object is
 3406 considered, respectively, to be less than, to match, or to be greater than the array element. The
 3407 array must consist of: all the elements that compare less than, all the elements that compare
 3408 equal to, and all the elements that compare greater than the *key* object, in that order.

3409 **RETURN VALUE**

3410 The *bsearch()* function returns a pointer to a matching member of the array, or a null pointer if
 3411 no match is found. If two or more members compare equal, which member is returned is
 3412 unspecified.

3413 **ERRORS**

3414 No errors are defined.

3415 **EXAMPLES**

3416 The example below searches a table containing pointers to nodes consisting of a string and its
 3417 length. The table is ordered alphabetically on the string in the node pointed to by each entry.

3418 The code fragment below reads in strings and either finds the corresponding node and prints out
 3419 the string and its length, or prints an error message.

```
3420       #include <stdio.h>
3421       #include <stdlib.h>
3422       #include <string.h>
3423       #define TABSIZE 1000
3424       struct node {                                /* these are stored in the table */
3425           char *string;
3426           int length;
3427       };
3428       struct node table[TABSIZE]; /* table to be searched */
3429       .
3430       .
3431       .
3432       {
3433           struct node *node_ptr, node;
3434           /* routine to compare 2 nodes */
3435           int node_compare(const void *, const void *);
3436           char str_space[20]; /* space to read string into */
3437           .
3438           .
3439           .
```

```

3440     node.string = str_space;
3441     while (scanf("%s", node.string) != EOF) {
3442         node_ptr = (struct node *)bsearch((void *)&node,
3443             (void *)table, TABSIZE,
3444             sizeof(struct node), node_compare);
3445         if (node_ptr != NULL) {
3446             (void)printf("string = %20s, length = %d\n",
3447                 node_ptr->string, node_ptr->length);
3448         } else {
3449             (void)printf("not found: %s\n", node.string);
3450         }
3451     }
3452 }
3453 /*
3454     This routine compares two nodes based on an
3455     alphabetical ordering of the string field.
3456 */
3457 int
3458 node_compare(const void *node1, const void *node2)
3459 {
3460     return strcoll(((const struct node *)node1)->string,
3461         ((const struct node *)node2)->string);
3462 }

```

3463 APPLICATION USAGE

3464 The pointers to the key and the element at the base of the table should be of type pointer-to-
3465 element.

3466 The comparison function need not compare every byte, so arbitrary data may be contained in
3467 the elements in addition to the values being compared.

3468 In practice, the array is usually sorted according to the comparison function.

3469 FUTURE DIRECTIONS

3470 None.

3471 SEE ALSO

3472 *bsearch()*, *lsearch()*, *qsort()*, *tsearch()*, *<stdlib.h>*.

3473 CHANGE HISTORY

3474 First released in Issue 1.

3475 Derived from Issue 1 of the SVID.

3476 Issue 4

3477 The following changes are incorporated for alignment with the ISO C standard:

- 3478 • The type of arguments *key* and *base*, and the type of arguments to *compar()*, are changed from
3479 **void*** to **const void***.

- 3480 • The requirement that the table be sorted according to *compar()* is removed from the
3481 DESCRIPTION.

3482	Other changes are incorporated as follows:	
3483	• Text indicating the need for various casts is removed from the APPLICATION USAGE	
3484	section.	
3485	• The code in the EXAMPLES section is changed to use <i>strcoll()</i> instead of <i>strcmp()</i> in	
3486	<i>node_compare()</i> .	
3487	• The return value and the contents of the array are now requirements on the application.	
3488	• The DESCRIPTION is changed to specify the order of arguments.	

3489 **NAME**

3490 btowc — single-byte to wide-character conversion

3491 **SYNOPSIS**

3492 #include <stdio.h>

3493 #include <wchar.h>

3494 wint_t btowc(int c);

3495 **DESCRIPTION**3496 The *btowc()* function determines whether *c* constitutes a valid (one-byte) character in the initial shift state.

3498 The behaviour of this function is affected by the LC_CTYPE category of the current locale.

3499 **RETURN VALUE**3500 The *btowc()* function returns WEOF if *c* has the value EOF or if (**unsigned char**) *c* does not constitute a valid (one-byte) character in the initial shift state. Otherwise, it returns the wide-character representation of that character.3503 **ERRORS**

3504 No errors are defined.

3505 **EXAMPLES**

3506 None.

3507 **APPLICATION USAGE**

3508 None.

3509 **FUTURE DIRECTIONS**

3510 None.

3511 **SEE ALSO**3512 *wctob()*, <wchar.h>.3513 **CHANGE HISTORY**

3514 First released in Issue 5.

3515 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

3516 **NAME**

3517 bzero — memory operations

3518 **SYNOPSIS**

3519 EX #include <strings.h>

3520 void bzero(void *s, size_t n);

3521

3522 **DESCRIPTION**3523 The *bzero()* function places *n* zero-valued bytes in the area pointed to by *s*.3524 **RETURN VALUE**3525 The *bzero()* function returns no value.3526 **ERRORS**

3527 No errors are defined.

3528 **EXAMPLES**

3529 None.

3530 **APPLICATION USAGE**3531 For portability to implementations conforming to earlier versions of this specification, *memset()*
3532 is preferred over this function.3533 **FUTURE DIRECTIONS**

3534 None.

3535 **SEE ALSO**3536 *memset()*, <strings.h>.3537 **CHANGE HISTORY**

3538 First released in Issue 4, Version 2.

3539 **Issue 5**

3540 Moved from X/OPEN UNIX extension to BASE.

3541 **NAME**

3542 calloc — a memory allocator

3543 **SYNOPSIS**

3544 #include <stdlib.h>

3545 void *calloc(size_t *nelem*, size_t *elsize*);3546 **DESCRIPTION**3547 The *calloc()* function allocates unused space for an array of *nelem* elements each of whose size in
3548 bytes is *elsize*. The space is initialised to all bits 0.

3549 The order and contiguity of storage allocated by successive calls to *calloc()* is unspecified. The
 3550 pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a
 3551 pointer to any type of object and then used to access such an object or an array of such objects in
 3552 the space allocated (until the space is explicitly freed or reallocated). Each such allocation will
 3553 yield a pointer to an object disjoint from any other object. The pointer returned points to the
 3554 start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer
 3555 is returned. If the size of the space requested is 0, the behaviour is implementation-dependent;
 3556 the value returned will be either a null pointer or a unique pointer.

3557 **RETURN VALUE**

3558 Upon successful completion with both *nelem* and *elsize* non-zero, *calloc()* returns a pointer to the
 3559 allocated space. If either *nelem* or *elsize* is 0, then either a null pointer or a unique pointer value
 3560 EX that can be successfully passed to *free()* is returned. Otherwise, it returns a null pointer and sets
 3561 *errno* to indicate the error.

3562 **ERRORS**3563 The *calloc()* function will fail if:

3564 EX [ENOMEM] Insufficient memory is available.

3565 **EXAMPLES**

3566 None.

3567 **APPLICATION USAGE**

3568 There is now no requirement for the implementation to support the inclusion of <malloc.h>.

3569 **FUTURE DIRECTIONS**

3570 None.

3571 **SEE ALSO**3572 *free()*, *malloc()*, *realloc()*, <stdlib.h>.3573 **CHANGE HISTORY**

3574 First released in Issue 1.

3575 Derived from Issue 1 of the SVID.

3576 **Issue 4**

3577 The following changes are incorporated in this issue for alignment with the ISO C standard:

- 3578 • The DESCRIPTION is updated to indicate (a) that the order and contiguity of storage
 3579 allocated by successive calls to this function is unspecified, (b) that each allocation yields a
 3580 pointer to an object disjoint from any other object, (c) that the returned pointer points to the
 3581 lowest byte address of the allocation, and (d) the behaviour if space is requested of zero size.
- 3582 • The RETURN VALUE section is updated to indicate what will be returned if either *nelem* or
 3583 *elsize* is 0.

3584 Other changes are incorporated as follows:

3585 • The setting of *errno* and the [ENOMEM] error are marked as extensions.

3586 • The APPLICATION USAGE section is changed to record that <**malloc.h**> need no longer be
3587 supported on XSI-conformant systems.

3588 **NAME**

3589 catclose — close a message catalogue descriptor

3590 **SYNOPSIS**

3591 EX #include <nl_types.h>

3592 int catclose(nl_catd catd);

3593

3594 **DESCRIPTION**

3595 The *catclose()* function closes the message catalogue identified by *catd*. If a file descriptor is used
3596 to implement the type **nl_catd**, that file descriptor will be closed.

3597 **RETURN VALUE**

3598 Upon successful completion, *catclose()* returns 0. Otherwise -1 is returned, and *errno* is set to
3599 indicate the error.

3600 **ERRORS**3601 The *catclose()* function may fail if:

3602 [EBADF] The catalogue descriptor is not valid.

3603 [EINTR] The *catclose()* function was interrupted by a signal.3604 **EXAMPLES**

3605 None.

3606 **APPLICATION USAGE**

3607 None.

3608 **FUTURE DIRECTIONS**

3609 None.

3610 **SEE ALSO**3611 *catgets()*, *catopen()*, <nl_types.h>.3612 **CHANGE HISTORY**

3613 First released in Issue 2.

3614 **Issue 4**

3615 The following change is incorporated in this issue:

- 3616
 - The [EBADF] and [EINTR] errors are added to the ERRORS section.

3617 **NAME**

3618 catgets — read a program message

3619 **SYNOPSIS**

3620 EX #include <nl_types.h>

3621 char *catgets(nl_catd *catd*, int *set_id*, int *msg_id*, const char **s*);

3622

3623 **DESCRIPTION**

3624 The *catgets()* function attempts to read message *msg_id*, in set *set_id*, from the message catalogue identified by *catd*. The *catd* argument is a message catalogue descriptor returned from an earlier call to *catopen()*. The *s* argument points to a default message string which will be returned by *catgets()* if it cannot retrieve the identified message.

3628 This interface need not be reentrant.

3629 **RETURN VALUE**

3630 If the identified message is retrieved successfully, *catgets()* returns a pointer to an internal buffer area containing the null-terminated message string. If the call is unsuccessful for any reason, *s* is returned and *errno* may be set to indicate the error.

3633 **ERRORS**3634 The *catgets()* function may fail if:

3635 [EBADF] The *catd* argument is not a valid message catalogue descriptor open for
3636 reading.

3637 [EINTR] The read operation was terminated due to the receipt of a signal, and no data
3638 was transferred.

3639 [EINVAL] The message catalog identified by *catd* is corrupted.3640 [ENOMSG] The message identified by *set_id* and *msg_id* is not in the message catalog.3641 **EXAMPLES**

3642 None.

3643 **APPLICATION USAGE**

3644 None.

3645 **FUTURE DIRECTIONS**

3646 None.

3647 **SEE ALSO**3648 *catclose()*, *catopen()*, <nl_types.h>.3649 **CHANGE HISTORY**

3650 First released in Issue 2.

3651 **Issue 4**

3652 The following changes are incorporated in this issue:

- 3653 • The type of argument *s* is changed from **char *** to **const char ***.
- 3654 • The [EBADF] and [EINTR] errors are added to the ERRORS section.

3655 **Issue 4, Version 2**

3656 The following changes are incorporated for X/OPEN UNIX conformance:

- 3657 • The RETURN VALUE section notes that *errno* may be set to indicate an error.

- | | | |
|------|--|--|
| 3658 | • In the ERRORS section, [EINVAL] and [ENOMSG] are added as optional errors. | |
| 3659 | Issue 5 | |
| 3660 | A note indicating that this interface need not be reentrant is added to the DESCRIPTION. | |

3661 NAME

3662 catopen — open a message catalogue

3663 SYNOPSIS

3664 EX `#include <nl_types.h>`3665 `nl_catd catopen(const char *name, int oflag);`

3666

3667 DESCRIPTION

3668 The *catopen()* function opens a message catalogue and returns a message catalogue descriptor.
 3669 The *name* argument specifies the name of the message catalogue to be opened. If *name* contains a
 3670 *" / "*, then *name* specifies a complete name for the message catalogue. Otherwise, the environment
 3671 variable *NLSPATH* is used with *name* substituted for *%N* (see the **XBD** specification, **Chapter 6**,
 3672 **Environment Variables**). If *NLSPATH* does not exist in the environment, or if a message
 3673 catalogue cannot be found in any of the components specified by *NLSPATH*, then an
 3674 implementation-dependent default path is used. This default may be affected by the setting of
 3675 *LC_MESSAGES* if the value of *oflag* is *NL_CAT_LOCALE*, or the *LANG* environment variable if
 3676 *oflag* is 0.

3677 A message catalogue descriptor remains valid in a process until that process closes it, or a
 3678 successful call to one of the *exec* functions. A change in the setting of the *LC_MESSAGES*
 3679 category may invalidate existing open catalogues.

3680 If a file descriptor is used to implement message catalogue descriptors, the *FD_CLOEXEC* flag
 3681 will be set; see *<fcntl.h>*.

3682 If the value of the *oflag* argument is 0, the *LANG* environment variable is used to locate the
 3683 catalogue without regard to the *LC_MESSAGES* category. If the *oflag* argument is
 3684 *NL_CAT_LOCALE*, the *LC_MESSAGES* category is used to locate the message catalogue (see
 3685 the **XBD** specification, **Section 6.2, Internationalisation Variables**).

3686 RETURN VALUE

3687 Upon successful completion, *catopen()* returns a message catalogue descriptor for use on
 3688 subsequent calls to *catgets()* and *catclose()*. Otherwise *catopen()* returns (*nl_catd*) -1 and sets
 3689 *errno* to indicate the error.

3690 ERRORS

3691 The *catopen()* function may fail if:

3692 [EACCES] Search permission is denied for the component of the path prefix of the
 3693 message catalogue or read permission is denied for the message catalogue.

3694 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

3695 [ENAMETOOLONG]

3696 The length of the pathname of the message catalogue exceeds {PATH_MAX},
 3697 or a pathname component is longer than {NAME_MAX}.

3698 [ENAMETOOLONG]

3699 Pathname resolution of a symbolic link produced an intermediate result
 3700 whose length exceeds {PATH_MAX}.

3701 [ENFILE] Too many files are currently open in the system.

3702 [ENOENT] The message catalogue does not exist or the *name* argument points to an
 3703 empty string.

3704 [ENOMEM] Insufficient storage space is available.

3705 [ENOTDIR] A component of the path prefix of the message catalogue is not a directory.

3706 EXAMPLES

3707 None.

3708 APPLICATION USAGE

3709 Some implementations of *catopen()* use *malloc()* to allocate space for internal buffer areas. The
3710 *catopen()* function may fail if there is insufficient storage space available to accommodate these
3711 buffers.

3712 Portable applications must assume that message catalogue descriptors are not valid after a call
3713 to one of the *exec* functions.

3714 Application writers should be aware that guidelines for the location of message catalogues have
3715 not yet been developed. Therefore they should take care to avoid conflicting with catalogues
3716 used by other applications and the standard utilities.

3717 FUTURE DIRECTIONS

3718 None.

3719 SEE ALSO

3720 *catclose()*, *catgets()*, *<fcntl.h>*, *<nl_types.h>*, the XCU specification, *gencat*.

3721 CHANGE HISTORY

3722 First released in Issue 2.

3723 Issue 4

3724 The following changes are incorporated in this issue:

- 3725 • The type of argument *name* is changed from **char *** to **const char ***.
- 3726 • The DESCRIPTION is updated (a) to indicate the longevity of message catalogue descriptors,
3727 and (b) to specify values for the *oflag* argument and the effect of LC_MESSAGES and
3728 *NLSPATH*.
- 3729 • The [EACCES], [EMFILE], [ENAMETOOLONG], [ENFILE], [ENOENT] and [ENOTDIR]
3730 errors are added to the ERRORS section.
- 3731 • The APPLICATION USAGE section is updated to indicate that (a) portable applications
3732 should not assume the continued validity of message catalogue descriptors after a call to one
3733 of the *exec* functions, and (b) message catalogues must be located with care.

3734 Issue 4, Version 2

3735 The following change is incorporated for X/OPEN UNIX conformance:

- 3736 • In the ERRORS section, an [ENAMETOOLONG] condition is defined that may report
3737 excessive length of an intermediate result of pathname resolution of a symbolic link.

3738 **NAME**

3739 cbrt — cube root function

3740 **SYNOPSIS**

3741 EX #include <math.h>

3742 double cbrt(double x);

3743

3744 **DESCRIPTION**3745 The *cbrt()* function computes the cube root of *x*.3746 **RETURN VALUE**3747 On successful completion, *cbrt()* returns the cube root of *x*. If *x* is NaN, *cbrt()* returns NaN and
3748 *errno* may be set to [EDOM].3749 **ERRORS**3750 The *cbrt()* function may fail if:3751 [EDOM] The value of *x* is NaN.3752 **EXAMPLES**

3753 None.

3754 **APPLICATION USAGE**

3755 None.

3756 **FUTURE DIRECTIONS**

3757 None.

3758 **SEE ALSO**

3759 <math.h>.

3760 **CHANGE HISTORY**

3761 First released in Issue 4, Version 2.

3762 **Issue 5**

3763 Moved from X/OPEN UNIX extension to BASE.

3764 **NAME**

3765 ceil — ceiling value function

3766 **SYNOPSIS**

3767 #include <math.h>

3768 double ceil(double x);

3769 **DESCRIPTION**3770 The *ceil()* function computes the smallest integral value not less than *x*.

3771 An application wishing to check for error situations should set *errno* to 0 before calling *ceil()*. If
 3772 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

3773 **RETURN VALUE**

3774 Upon successful completion, *ceil()* returns the smallest integral value not less than *x*, expressed
 3775 as a type **double**.

3776 EX If *x* is NaN, NaN is returned and *errno* may be set to [EDOM].3777 EX If the correct value would cause overflow, HUGE_VAL is returned and *errno* is set to [ERANGE].3778 If *x* is $\pm\text{Inf}$ or ± 0 , the value of *x* is returned.3779 **ERRORS**3780 The *ceil()* function will fail if:

3781 [ERANGE] The result overflows.

3782 The *ceil()* function may fail if:3783 EX [EDOM] The value of *x* is NaN.

3784 EX No other errors will occur.

3785 **EXAMPLES**

3786 None.

3787 **APPLICATION USAGE**3788 The integral value returned by *ceil()* as a **double** need not be expressible as an **int** or **long int**.

3789 The return value should be tested before assigning it to an integer type to avoid the undefined
 3790 results of an integer overflow.

3791 The *ceil()* function can only overflow when the floating point representation has
 3792 DBL_MANT_DIG > DBL_MAX_EXP.

3793 **FUTURE DIRECTIONS**

3794 None.

3795 **SEE ALSO**3796 *floor()*, *isnan()*, <math.h>.3797 **CHANGE HISTORY**

3798 First released in Issue 1.

3799 Derived from Issue 1 of the SVID.

3800 **Issue 4**

3801 The following changes are incorporated in this issue:

- 3802 • Removed references to *matherr()*.
- 3803 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
3804 the ISO C standard and to rationalise error handling in the mathematics functions.
- 3805 • The return value specified for [EDOM] is marked as an extension.
- 3806 • Support for x being $\pm\text{Inf}$ or ± 0 is added to the RETURN VALUE section and marked as an
3807 extension.

3808 **Issue 5**

3809 The DESCRIPTION is updated to indicate how an application should check for an error. This
3810 text was previously published in the APPLICATION USAGE section.

3811 **NAME**

3812 cfgetispeed — get input baud rate

3813 **SYNOPSIS**

3814 #include <termios.h>

3815 speed_t cfgetispeed(const struct termios *termios_p);

3816 **DESCRIPTION**3817 The *cfgetispeed()* function extracts the input baud rate from the **termios** structure to which the
3818 *termios_p* argument points.3819 This function returns exactly the value in the **termios** data structure, without interpretation.3820 **RETURN VALUE**3821 Upon successful completion, *cfgetispeed()* returns a value of type **speed_t** representing the input
3822 baud rate.3823 **ERRORS**

3824 No errors are defined.

3825 **EXAMPLES**

3826 None.

3827 **APPLICATION USAGE**

3828 None.

3829 **FUTURE DIRECTIONS**

3830 None.

3831 **SEE ALSO**3832 *cfgetospeed()*, *cfsetispeed()*, *cfsetospeed()*, *tcgetattr()*, <termios.h>, the XBD specification, **Chapter**
3833 **9, General Terminal Interface**.3834 **CHANGE HISTORY**

3835 First released in Issue 3.

3836 Entry included for alignment with the POSIX.1-1988 standard.

3837 **Issue 4**

3838 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- 3839 • The type of the argument *termios_p* is changed from **struct termios*** to **const struct termios***.
- 3840 • The DESCRIPTION is changed to indicate that the function simply returns the value from
- 3841 *termios_p*, irrespective of how that structure was obtained. Issue 3 states that if *termios_p* was
- 3842 not obtained by a successful call to *tcgetattr()*, the behaviour is undefined.

3843 **NAME**

3844 cfgetospeed — get output baud rate

3845 **SYNOPSIS**

3846 #include <termios.h>

3847 speed_t cfgetospeed(const struct termios *termios_p);

3848 **DESCRIPTION**3849 The *cfgetospeed()* function extracts the output baud rate from the **termios** structure to which the
3850 *termios_p* argument points.3851 This function returns exactly the value in the **termios** data structure, without interpretation.3852 **RETURN VALUE**3853 Upon successful completion, *cfgetospeed()* returns a value of type **speed_t** representing the
3854 output baud rate.3855 **ERRORS**

3856 No errors are defined.

3857 **EXAMPLES**

3858 None.

3859 **APPLICATION USAGE**

3860 None.

3861 **FUTURE DIRECTIONS**

3862 None.

3863 **SEE ALSO**3864 *cfgetispeed()*, *cfsetispeed()*, *cfsetospeed()*, *tcgetattr()*, <termios.h>, the XBD specification, Chapter
3865 9, General Terminal Interface.3866 **CHANGE HISTORY**

3867 First released in Issue 3.

3868 Entry included for alignment with the POSIX.1-1988 standard.

3869 **Issue 4**

3870 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- 3871 • The type of the argument *termios_p* is changed from **struct termios*** to **const struct termios***.
- 3872 • The DESCRIPTION is changed to indicate that the function simply returns the value from
- 3873 *termios_p*, irrespective of how that structure was obtained. Issue 3 states that if *termios_p* was
- 3874 not obtained by a successful call to *tcgetattr()*, the behaviour is undefined.

3875 **NAME**

3876 cfsetispeed — set input baud rate

3877 **SYNOPSIS**

3878 #include <termios.h>

3879 int cfsetispeed(struct termios *termios_p, speed_t speed);

3880 **DESCRIPTION**3881 The *cfsetispeed()* function sets the input baud rate stored in the structure pointed to by *termios_p*
3882 to *speed*.3883 There is no effect on the baud rates set in the hardware until a subsequent successful call to
3884 *tcsetattr()* on the same **termios** structure.3885 **RETURN VALUE**3886 **EX** Upon successful completion, *cfsetispeed()* returns 0. Otherwise -1 is returned, and *errno* may be
3887 set to indicate the error.3888 **ERRORS**3889 The *cfsetispeed()* function may fail if:3890 **EX** [EINVAL] The *speed* value is not a valid baud rate.3891 **EX** [EINVAL] The value of *speed* is outside the range of possible speed values as specified in
3892 <termios.h>.3893 **EXAMPLES**

3894 None.

3895 **APPLICATION USAGE**

3896 None.

3897 **FUTURE DIRECTIONS**

3898 None.

3899 **SEE ALSO**3900 *cfgetispeed()*, *cfgetospeed()*, *cfsetospeed()*, *tcsetattr()*, <termios.h>, the XBD specification, Chapter
3901 9, General Terminal Interface.3902 **CHANGE HISTORY**

3903 First released in Issue 3.

3904 Entry included for alignment with the POSIX.1-1988 standard.

3905 **Issue 4**

3906 The following change is incorporated in this issue:

- 3907
- The first description of the [EINVAL] error is added and is marked as an extension.

3908 **Issue 4, Version 2**3909 The ERRORS section is changed to indicate that [EINVAL] may be returned if the specified
3910 speed is outside the range of possible speed values given in <termios.h>.

3911 **NAME**

3912 cfsetospeed — set output baud rate

3913 **SYNOPSIS**

3914 #include <termios.h>

3915 int cfsetospeed(struct termios *termios_p, speed_t speed);

3916 **DESCRIPTION**3917 The *cfsetospeed()* function sets the output baud rate stored in the structure pointed to by
3918 *termios_p* to *speed*.3919 There is no effect on the baud rates set in the hardware until a subsequent successful call to
3920 *tcsetattr()* on the same **termios** structure.3921 **RETURN VALUE**3922 EX Upon successful completion, *cfsetospeed()* returns 0. Otherwise it returns -1 and *errno* may be set
3923 to indicate the error.3924 **ERRORS**3925 The *cfsetospeed()* function may fail if:3926 EX [EINVAL] The *speed* value is not a valid baud rate.3927 EX [EINVAL] The value of *speed* is outside the range of possible speed values as specified in
3928 <termios.h>.3929 **EXAMPLES**

3930 None.

3931 **APPLICATION USAGE**

3932 None.

3933 **FUTURE DIRECTIONS**

3934 None.

3935 **SEE ALSO**3936 *cfgetispeed()*, *cfgetospeed()*, *cfsetispeed()*, *tcsetattr()*, <termios.h>, the XBD specification, Chapter
3937 9, General Terminal Interface.3938 **CHANGE HISTORY**

3939 First released in Issue 3.

3940 Entry included for alignment with the POSIX.1-1988 standard.

3941 **Issue 4**

3942 The following change is incorporated in this issue:

- 3943
- The first description of the [EINVAL] error is added and is marked as an extension.

3944 **Issue 4, Version 2**3945 The ERRORS section is changed to indicate that [EINVAL] may be returned if the specified
3946 speed is outside the range of possible speed values given in <termios.h>.

3947 **NAME**

3948 chdir — change working directory

3949 **SYNOPSIS**

3950 #include <unistd.h>

3951 int chdir(const char *path);

3952 **DESCRIPTION**

3953 The *chdir()* function causes the directory named by the pathname pointed to by the *path*
 3954 argument to become the current working directory; that is, the starting point for path searches
 3955 for pathnames not beginning with */*.

3956 **RETURN VALUE**

3957 Upon successful completion, 0 is returned. Otherwise, -1 is returned, the current working
 3958 directory remains unchanged and *errno* is set to indicate the error.

3959 **ERRORS**3960 The *chdir()* function will fail if:

3961 [EACCES] Search permission is denied for any component of the pathname.

3962 EX [ELOOP] Too many symbolic links were encountered in resolving *path*.

3963 FIPS [ENAMETOOLONG]

3964 The *path* argument exceeds {PATH_MAX} in length or a pathname component
 3965 is longer than {NAME_MAX}.

3966 [ENOENT] A component of *path* does not name an existing directory or *path* is an empty
 3967 string.

3968 [ENOTDIR] A component of the pathname is not a directory.

3969 The *chdir()* function may fail if:

3970 EX [ENAMETOOLONG]

3971 Pathname resolution of a symbolic link produced an intermediate result
 3972 whose length exceeds {PATH_MAX}.

3973 **EXAMPLES**

3974 None.

3975 **APPLICATION USAGE**

3976 None.

3977 **FUTURE DIRECTIONS**

3978 None.

3979 **SEE ALSO**3980 *getcwd()*, <unistd.h>.3981 **CHANGE HISTORY**

3982 First released in Issue 1.

3983 Derived from Issue 1 of the SVID.

3984 **Issue 4**

3985 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 3986 • The type of argument *path* is changed from **char *** to **const char ***.

3987 The following change is incorporated for alignment with the FIPS requirements:

3988 • In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a
3989 pathname component is larger than {NAME_MAX} is now defined as mandatory and marked
3990 as an extension. |

3991 Another change is incorporated as follows:

3992 • The <unistd.h> header is added to the SYNOPSIS section. |

3993 **Issue 4, Version 2**

3994 The ERRORS section is updated for X/OPEN UNIX conformance as follows: |

3995 • It states that [ELOOP] will be returned if too many symbolic links are encountered during
3996 pathname resolution.

3997 • A second [ENAMETOOLONG] condition is defined that may report excessive length of an
3998 intermediate result of pathname resolution of a symbolic link. |

3999 **NAME**4000 `chmod` — change mode of a file4001 **SYNOPSIS**4002 OH `#include <sys/types.h>`4003 `#include <sys/stat.h>`4004 `int chmod(const char *path, mode_t mode);`4005 **DESCRIPTION**

4006 EX The `chmod()` function changes `S_ISUID`, `S_ISGID`, `S_ISVTX` and the file permission bits of the file
 4007 named by the pathname pointed to by the `path` argument to the corresponding bits in the `mode`
 4008 argument. The effective user ID of the process must match the owner of the file or the process
 4009 must have appropriate privileges in order to do this.

4010 `S_ISUID`, `S_ISGID` and the file permission bits are described in `<sys/stat.h>`.

4011 EX If a directory is writable and the mode bit `S_ISVTX` is set on the directory, a process may remove
 4012 or rename files within that directory only if one or more of the following is true:

- 4013 • The effective user ID of the process is the same as that of the owner ID of the file.
- 4014 • The effective user ID of the process is the same as that of the owner ID of the directory.
- 4015 • The process has appropriate privileges.

4016 If the `S_ISVTX` bit is set on a non-directory file, the behaviour is unspecified.

4017 If the calling process does not have appropriate privileges, and if the group ID of the file does
 4018 not match the effective group ID or one of the supplementary group IDs and if the file is a
 4019 regular file, bit `S_ISGID` (set-group-ID on execution) in the file's mode will be cleared upon
 4020 successful return from `chmod()`.

4021 Additional implementation-dependent restrictions may cause the `S_ISUID` and `S_ISGID` bits in
 4022 `mode` to be ignored.

4023 The effect on file descriptors for files open at the time of a call to `chmod()` is implementation-
 4024 dependent.

4025 Upon successful completion, `chmod()` will mark for update the `st_ctime` field of the file.4026 **RETURN VALUE**

4027 Upon successful completion, 0 is returned. Otherwise, -1 is returned and `errno` is set to indicate
 4028 the error. If -1 is returned, no change to the file mode will occur.

4029 **ERRORS**4030 The `chmod()` function will fail if:4031 `[EACCES]` Search permission is denied on a component of the path prefix.4032 EX `[ELOOP]` Too many symbolic links were encountered in resolving `path`.4033 FIPS `[ENAMETOOLONG]`

4034 The length of the `path` argument exceeds `{PATH_MAX}` or a pathname
 4035 component is longer than `{NAME_MAX}`.

4036 `[ENOTDIR]` A component of the path prefix is not a directory.4037 `[ENOENT]` A component of `path` does not name an existing file or `path` is an empty string.

4038 `[EPERM]` The effective user ID does not match the owner of the file and the process
 4039 does not have appropriate privileges.

4040 [EROFS] The named file resides on a read-only file system.

4041 The *chmod()* function may fail if:

4042 EX [EINTR] A signal was caught during execution of the function.

4043 EX [EINVAL] The value of the *mode* argument is invalid.

4044 EX [ENAMETOOLONG]

4045 Pathname resolution of a symbolic link produced an intermediate result
4046 whose length exceeds {PATH_MAX}.

4047 EXAMPLES

4048 None.

4049 APPLICATION USAGE

4050 In order to ensure that the S_ISUID and S_ISGID bits are set, an application requiring this should
4051 use *stat()* after a successful *chmod()* to verify this.

4052 Any file descriptors currently open by any process on the file may become invalid if the mode of
4053 the file is changed to a value which would deny access to that process. One situation where this
4054 could occur is on a stateless file system.

4055 FUTURE DIRECTIONS

4056 None.

4057 SEE ALSO

4058 *chown()*, *mkdir()*, *mkfifo()*, *open()*, *stat()*, *statvfs()*, <sys/stat.h>, <sys/types.h>.

4059 CHANGE HISTORY

4060 First released in Issue 1.

4061 Derived from Issue 1 of the SVID.

4062 Issue 4

4063 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 4064 • The type of argument *path* is changed from **char *** to **const char ***.

4065 The following change is incorporated for alignment with the FIPS requirements:

- 4066 • In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a
4067 pathname component is larger than {NAME_MAX} is now defined as mandatory and marked
4068 as an extension.

4069 Other changes are incorporated as follows:

- 4070 • The <sys/types.h> header is now marked as optional (OH); this header need not be included
4071 on XSI-conformant systems.
- 4072 • The [EINVAL] error is marked as an extension.

4073 Issue 4, Version 2

4074 The following changes are incorporated for X/OPEN UNIX conformance:

- 4075 • The DESCRIPTION is updated to describe X/OPEN UNIX functionality relating to
4076 permission checks applied when removing or renaming files in a directory having the
4077 S_ISVTX bit set.
- 4078 • In the ERRORS section, the condition whereby [ELOOP] will be returned if too many
4079 symbolic links are encountered during pathname resolution is defined as mandatory, and
4080 [EINTR] is added as an optional error.

- 4081 • In the ERRORS section, a second [ENAMETOOLONG] condition is defined that may report |
4082 excessive length of an intermediate result of pathname resolution of a symbolic link.

4083 **NAME**

4084 chown — change owner and group of a file

4085 **SYNOPSIS**

4086 OH #include <sys/types.h>

4087 #include <unistd.h>

4088 int chown(const char *path, uid_t owner, gid_t group);

4089 **DESCRIPTION**4090 The *path* argument points to a pathname 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.

4092 FIPS On XSI-conformant systems {_POSIX_CHOWN_RESTRICTED} is always defined, therefore:

- 4093 • Changing the user ID is restricted to processes with appropriate privileges.
- 4094 • 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.

4098 If the *path* argument refers to a regular file, the set-user-ID (S_ISUID) and set-group-ID (S_ISGID) bits of the file mode are cleared upon successful return from *chown()*, unless the call is made by a process with appropriate privileges, in which case it is implementation-dependent whether these bits are altered. If *chown()* is successfully invoked on a file that is not a regular file, these bits may be cleared. These bits are defined in <sys/stat.h>.

4103 EX If *owner* or *group* is specified as (**uid_t**)−1 or (**gid_t**)−1 respectively, the corresponding ID of the file is unchanged.

4105 Upon successful completion, *chown()* will mark for update the *st_ctime* field of the file.4106 **RETURN VALUE**

4107 Upon successful completion, 0 is returned. Otherwise, −1 is returned and *errno* is set to indicate the error. If −1 is returned, no changes are made in the user ID and group ID of the file.

4109 **ERRORS**4110 The *chown()* function will fail if:

4111 [EACCES] Search permission is denied on a component of the path prefix.

4112 EX [ELOOP] Too many symbolic links were encountered in resolving *path*.

4113 FIPS [ENAMETOOLONG]

4114 The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

4116 [ENOTDIR] A component of the path prefix is not a directory.

4117 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

4118 FIPS [EPERM] The effective user ID does not match the owner of the file, or the calling process does not have appropriate privileges.

4120 [EROFS] The named file resides on a read-only file system.

4121 The *chown()* function may fail if:

4122 EX [EIO] An I/O error occurred while reading or writing to the file system.

4123 [EINTR] The *chown()* function was interrupted by a signal which was caught.

4124 [EINVAL] The owner or group ID supplied is not a value supported by the
4125 implementation.

4126 EX [ENAMETOOLONG]
4127 Pathname resolution of a symbolic link produced an intermediate result
4128 whose length exceeds {PATH_MAX}.

4129 EXAMPLES

4130 None.

4131 APPLICATION USAGE

4132 Because {_POSIX_CHOWN_RESTRICTED} is always defined with a value other than -1 on
4133 XSI-conformant systems, the error [EPERM] is always returned if the effective user ID does not
4134 match the owner of the file, or the calling process does not have appropriate privileges.

4135 FUTURE DIRECTIONS

4136 None.

4137 SEE ALSO

4138 *chmod()*, <sys/types.h>, <unistd.h>.

4139 CHANGE HISTORY

4140 First released in Issue 1.

4141 Derived from Issue 1 of the SVID.

4142 Issue 4

4143 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 4144 • The type of argument *path* is changed from **char *** to **const char ***.

4145 The following changes are incorporated for alignment with the FIPS requirements:

- 4146 • In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a
4147 pathname component is larger than {NAME_MAX} is now defined as mandatory and marked
4148 as an extension.
- 4149 • In the ERRORS section, the condition whereby [EPERM] will be returned when an attempt is
4150 made to change the user ID of a file and the caller does not have appropriate privileges is
4151 now defined as mandatory and marked as an extension.

4152 Other changes are incorporated as follows:

- 4153 • The <sys/types.h> header is now marked as optional (OH); this header need not be included
4154 on XSI-conformant systems.
- 4155 • The value for *owner* of (uid_t)-1 is added to the DESCRIPTION to allow the use of -1 by the
4156 owner of a file to change the group ID only.
- 4157 • The APPLICATION USAGE section is added.

4158 Issue 4, Version 2

4159 The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- 4160 • It states that [ELOOP] will be returned if too many symbolic links are encountered during
4161 pathname resolution.
- 4162 • The [EIO] and [EINTR] optional conditions are added.
- 4163 • A second [ENAMETOOLONG] condition is defined that may report excessive length of an
4164 intermediate result of pathname resolution of a symbolic link.

4165 **NAME**4166 chroot — change root directory (**LEGACY**)4167 **SYNOPSIS**4168 EX `#include <unistd.h>`4169 `int chroot(const char *path);`

4170

4171 **DESCRIPTION**

4172 The *path* argument points to a pathname naming a directory. The *chroot()* function causes the
 4173 named directory to become the root directory; that is, the starting point for path searches for
 4174 pathnames beginning with /. The process' working directory is unaffected by *chroot()*.

4175 The process must have appropriate privileges to change the root directory.

4176 The dot-dot entry in the root directory is interpreted to mean the root directory itself. Thus,
 4177 dot-dot cannot be used to access files outside the subtree rooted at the root directory.

4178 This interface need not be reentrant.

4179 **RETURN VALUE**

4180 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate
 4181 the error. If -1 is returned, no change is made in the root directory.

4182 **ERRORS**

4183 The *chroot()* function will fail if:

4184 [EACCES] Search permission is denied for a component of *path*.

4185 [ELOOP] Too many symbolic links were encountered in resolving *path*.

4186 [ENAMETOOLONG]

4187 The length of the *path* argument exceeds {PATH_MAX} or a pathname
 4188 component is longer than {NAME_MAX}.

4189 [ENOENT] A component of *path* does not name an existing directory or *path* is an empty
 4190 string.

4191 [ENOTDIR] A component of the *path* name is not a directory.

4192 [EPERM] The effective user ID does not have appropriate privileges.

4193 The *chroot()* function may fail if:

4194 [ENAMETOOLONG]

4195 Pathname resolution of a symbolic link produced an intermediate result
 4196 whose length exceeds {PATH_MAX}.

4197 **EXAMPLES**

4198 None.

4199 **APPLICATION USAGE**

4200 There is no portable use that an application could make of this interface.

4201 **FUTURE DIRECTIONS**

4202 None.

4203 **SEE ALSO**

4204 *chdir()*, *<unistd.h>*.

4205 **CHANGE HISTORY**

4206 First released in Issue 1.

4207 Derived from Issue 1 of the SVID.

4208 **Issue 4**

4209 Changes are incorporated as follows:

- 4210 • The interface is marked TO BE WITHDRAWN, as there is no portable use that an application
4211 could make of this interface.
- 4212 • The <unistd.h> header is added to the SYNOPSIS section. |
- 4213 • The type of argument *path* is changed from **char *** to **const char ***.
- 4214 • The APPLICATION USAGE section is added. |
- 4215 • The DESCRIPTION now refers to the process' working directory instead of the user's
4216 working directory. |

4217 **Issue 4, Version 2**

4218 The ERRORS section is updated for X/OPEN UNIX conformance as follows: |

- 4219 • It states that [ELOOP] will be returned if too many symbolic links are encountered during
4220 pathname resolution.
- 4221 • A second [ENAMETOOLONG] condition is defined that may report excessive length of an
4222 intermediate result of pathname resolution of a symbolic link.

4223 **Issue 5**

4224 Marked LEGACY. |

4225 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

4226 NAME

4227 clearerr — clear indicators on a stream

4228 SYNOPSIS

4229 #include <stdio.h>

4230 void clearerr(FILE **stream*);

4231 DESCRIPTION

4232 The *clearerr()* function clears the end-of-file and error indicators for the stream to which *stream*
4233 points.

4234 RETURN VALUE

4235 The *clearerr()* function returns no value.

4236 ERRORS

4237 No errors are defined.

4238 EXAMPLES

4239 None.

4240 APPLICATION USAGE

4241 None.

4242 FUTURE DIRECTIONS

4243 None.

4244 SEE ALSO

4245 <stdio.h>.

4246 CHANGE HISTORY

4247 First released in Issue 1.

4248 Derived from Issue 1 of the SVID.

4249 **NAME**

4250 clock — report CPU time used

4251 **SYNOPSIS**

4252 #include <time.h>

4253 clock_t clock(void);

4254 **DESCRIPTION**

4255 The *clock()* function returns the implementation's best approximation to the processor time used
 4256 by the process since the beginning of an implementation-dependent time related only to the
 4257 process invocation.

4258 **RETURN VALUE**

4259 To determine the time in seconds, the value returned by *clock()* should be divided by the value
 4260 of the macro `CLOCKS_PER_SEC`. `CLOCKS_PER_SEC` is defined to be one million in <time.h>. If
 4261 the processor time used is not available or its value cannot be represented, the function returns
 4262 the value `(clock_t)-1`.

4263 **ERRORS**

4264 No errors are defined.

4265 **EXAMPLES**

4266 None.

4267 **APPLICATION USAGE**

4268 In order to measure the time spent in a program, *clock()* should be called at the start of the
 4269 program and its return value subtracted from the value returned by subsequent calls. The value
 4270 returned by *clock()* is defined for compatibility across systems that have clocks with different
 4271 resolutions. The resolution on any particular system need not be to microsecond accuracy.

4272 The value returned by *clock()* may wrap around on some systems. For example, on a machine
 4273 with 32-bit values for `clock_t`, it will wrap after 2147 seconds or 36 minutes.

4274 **FUTURE DIRECTIONS**

4275 None.

4276 **SEE ALSO**

4277 *asctime()*, *ctime()*, *difftime()*, *gmtime()*, *localtime()*, *mktime()*, *strftime()*, *strptime()*, *time()*, *utime()*,
 4278 <time.h>.

4279 **CHANGE HISTORY**

4280 First released in Issue 1.

4281 Derived from Issue 1 of the SVID.

4282 **Issue 4**

4283 The following changes are incorporated for alignment with the ISO C standard:

- 4284 • The <time.h> header is added to the SYNOPSIS section.
- 4285 • The DESCRIPTION and RETURN VALUE sections, though functionally equivalent to Issue
 4286 3, are rewritten for clarity and consistency with the ISO C standard. This issue also defines
 4287 under what circumstances `(clock_t)-1` can be returned by the function.
- 4288 • The function is no longer marked as an extension.

4289 Other changes are incorporated as follows:

4290 • Reference to the resolution of CLOCKS_PER_SEC is marked as an extension.

4291 • The ERRORS section is added. |

4292 • Advice on how to calculate the time spent in a program is added to the APPLICATION |
4293 USAGE section.

4294 **NAME**4295 clock_settime, clock_gettime, clock_getres — clock and timer functions (**REALTIME**)4296 **SYNOPSIS**4297 RT `#include <time.h>`4298 `int clock_settime(clockid_t clock_id, const struct timespec *tp);`4299 `int clock_gettime(clockid_t clock_id, struct timespec *tp);`4300 `int clock_getres(clockid_t clock_id, struct timespec *res);`

4301

4302 **DESCRIPTION**

4303 The `clock_settime()` function sets the specified clock, `clock_id`, to the value specified by `tp`. Time
 4304 values that are between two consecutive non-negative integer multiples of the resolution of the
 4305 specified clock are truncated down to the smaller multiple of the resolution.

4306 The `clock_gettime()` function returns the current value `tp` for the specified clock, `clock_id`.

4307 The resolution of any clock can be obtained by calling `clock_getres()`. Clock resolutions are
 4308 implementation-dependent and cannot be set by a process. If the argument `res` is not NULL, the
 4309 resolution of the specified clock is stored in the location pointed to by `res`. If `res` is NULL, the
 4310 clock resolution is not returned. If the time argument of `clock_settime()` is not a multiple of `res`,
 4311 then the value is truncated to a multiple of `res`.

4312 A clock may be systemwide (that is, visible to all processes) or per-process (measuring time that
 4313 is meaningful only within a process). All implementations support a `clock_id` of
 4314 `CLOCK_REALTIME` defined in `<time.h>`. This clock represents the realtime clock for the
 4315 system. For this clock, the values returned by `clock_gettime()` and specified by `clock_settime()`
 4316 represent the amount of time (in seconds and nanoseconds) since the Epoch. An
 4317 implementation may also support additional clocks. The interpretation of time values for these
 4318 clocks is unspecified.

4319 The effect of setting a clock via `clock_settime()` on armed per-process timers associated with that
 4320 clock is implementation-dependent.

4321 The appropriate privilege to set a particular clock is implementation-dependent.

4322 **RETURN VALUE**

4323 A return value of 0 indicates that the call succeeded. A return value of -1 indicates that an error
 4324 occurred, and `errno` is set to indicate the error.

4325 **ERRORS**

4326 The `clock_settime()`, `clock_gettime()` and `clock_getres()` functions will fail if:

4327 [EINVAL] The `clock_id` argument does not specify a known clock.

4328 [ENOSYS] The functions `clock_settime()`, `clock_gettime()`, and `clock_getres()` are not
 4329 supported by this implementation.

4330 The `clock_settime()` function will fail if:

4331 [EINVAL] The `tp` argument to `clock_settime()` is outside the range for the given clock id.

4332 [EINVAL] The `tp` argument specified a nanosecond value less than zero or greater than
 4333 or equal to 1000 million.

4334 The `clock_settime()` function may fail if:

4335 [EPERM] The requesting process does not have the appropriate privilege to set the
 4336 specified clock.

4337 **EXAMPLES**

4338 None.

4339 **APPLICATION USAGE**

4340 None.

4341 **FUTURE DIRECTIONS**

4342 None.

4343 **SEE ALSO**4344 *timer_gettime()*, *time()*, *ctime()*, <**time.h**>.4345 **CHANGE HISTORY**

4346 First released in Issue 5.

4347 Included for alignment with the POSIX Realtime Extension.

4348 **NAME**

4349 close — close a file descriptor

4350 **SYNOPSIS**

4351 #include <unistd.h>

4352 int close(int *fildes*);4353 **DESCRIPTION**

4354 The *close()* function will deallocate the file descriptor indicated by *fildes*. To deallocate means to
 4355 make the file descriptor available for return by subsequent calls to *open()* or other functions that
 4356 allocate file descriptors. All outstanding record locks owned by the process on the file
 4357 associated with the file descriptor will be removed (that is, unlocked).

4358 If *close()* is interrupted by a signal that is to be caught, it will return -1 with *errno* set to [EINTR]
 4359 and the state of *fildes* is unspecified.

4360 When all file descriptors associated with a pipe or FIFO special file are closed, any data
 4361 remaining in the pipe or FIFO will be discarded.

4362 When all file descriptors associated with an open file description have been closed the open file
 4363 description will be freed.

4364 If the link count of the file is 0, when all file descriptors associated with the file are closed, the
 4365 space occupied by the file will be freed and the file will no longer be accessible.

4366 EX If a STREAMS-based *fildes* is closed and the calling process was previously registered to receive
 4367 a SIGPOLL signal for events associated with that STREAM, the calling process will be
 4368 unregistered for events associated with the STREAM. The last *close()* for a STREAM causes the
 4369 STREAM associated with *fildes* to be dismantled. If O_NONBLOCK is not set and there have
 4370 been no signals posted for the STREAM, and if there is data on the module's write queue, *close()*
 4371 waits for an unspecified time (for each module and driver) for any output to drain before
 4372 dismantling the STREAM. The time delay can be changed via an I_SETCLTIME *ioctl()* request.
 4373 If the O_NONBLOCK flag is set, or if there are any pending signals, *close()* does not wait for
 4374 output to drain, and dismantles the STREAM immediately.

4375 If the implementation supports STREAMS-based pipes, and *fildes* is associated with one end of a
 4376 pipe, the last *close()* causes a hangup to occur on the other end of the pipe. In addition, if the
 4377 other end of the pipe has been named by *fattach()*, then the last *close()* forces the named end to
 4378 be detached by *fdetach()*. If the named end has no open file descriptors associated with it and
 4379 gets detached, the STREAM associated with that end is also dismantled.

4380 If *fildes* refers to the master side of a pseudo-terminal, and this is the last close, a SIGHUP signal
 4381 is sent to the process group, if any, for which the slave side of the pseudo-terminal is the
 4382 controlling terminal. It is unspecified whether closing the master side of the pseudo-terminal
 4383 flushes all queued input and output.

4384 If *fildes* refers to the slave side of a STREAMS-based pseudo-terminal, a zero-length message
 4385 may be sent to the master.

4386 RT If the Asynchronous Input and Output option is supported:

When there is an outstanding cancelable asynchronous I/O operation against *fil*des 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 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 cancelled, and which I/O operation may be cancelled upon *close()*, is implementation-dependent.

If the Mapped Files or Shared Memory Objects option is supported:

If a memory object remains referenced at the last close (that is, a process has it mapped), then the entire contents of the memory object persist until the memory object becomes unreferenced. If this is the last close of a memory object and the close results in the memory object becoming unreferenced, and the memory object has been unlinked, then the memory object will be removed.

4400

4401 RETURN VALUE

Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

4404 ERRORS

The *close()* function will fail if:

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

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

EX The *close()* function may fail if:

[EIO] An I/O error occurred while reading from or writing to the file system.

4410 EXAMPLES

None.

4412 APPLICATION USAGE

An application that had used the *stdio* routine *fopen()* to open a file should use the corresponding *fclose()* routine rather than *close()*.

4415 FUTURE DIRECTIONS

None.

4417 SEE ALSO

fattach(), *fclose()*, *fdetach()*, *fopen()*, *ioctl()*, *open()*, <unistd.h>, Section 2.5 on page 34.

4419 CHANGE HISTORY

First released in Issue 1.

Derived from Issue 1 of the SVID.

4422 Issue 4

The following change is incorporated in this issue:

- The <unistd.h> header is added to the SYNOPSIS section.

4425 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.

- 4429 • The ERRORS section describes a condition under which the [EIO] error may be returned. |
- 4430 **Issue 5** |
- 4431 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

4432 **NAME**

4433 closedir — close a directory stream

4434 **SYNOPSIS**

4435 OH #include <sys/types.h>

4436 #include <dirent.h>

4437 int closedir(DIR *dirp);

4438 **DESCRIPTION**

4439 The *closedir()* function closes the directory stream referred to by the argument *dirp*. Upon
 4440 return, the value of *dirp* may no longer point to an accessible object of the type **DIR**. If a file
 4441 descriptor is used to implement type **DIR**, that file descriptor will be closed.

4442 **RETURN VALUE**

4443 Upon successful completion, *closedir()* returns 0. Otherwise, -1 is returned and *errno* is set to
 4444 indicate the error.

4445 **ERRORS**4446 The *closedir()* function may fail if:

4447 [EBADF] The *dirp* argument does not refer to an open directory stream.

4448 EX [EINTR] The *closedir()* function was interrupted by a signal.

4449 **EXAMPLES**

4450 None.

4451 **APPLICATION USAGE**

4452 None.

4453 **FUTURE DIRECTIONS**

4454 None.

4455 **SEE ALSO**4456 *opendir()*, <dirent.h>, <sys/types.h>.4457 **CHANGE HISTORY**

4458 First released in Issue 2.

4459 **Issue 4**

4460 The following changes are incorporated in this issue:

- 4461 • The <sys/types.h> header is now marked as optional (OH); this header need not be included
- 4462 on XSI-conformant systems.
- 4463 • The [EINTR] error is marked as an extension.

4464 **NAME**

4465 closelog, openlog, setlogmask, syslog — control system log

4466 **SYNOPSIS**

```

4467 EX      #include <syslog.h>

4468          void closelog(void);
4469          void openlog(const char *ident, int logopt, int facility);
4470          int setlogmask(int maskpri);
4471          void syslog(int priority, const char *message, ... /* arguments */);
4472

```

4473 **DESCRIPTION**

4474 The *syslog()* function sends a message to an implementation-dependent logging facility, which
 4475 may log it in an implementation-dependent system log, write it to the system console, forward it
 4476 to a list of users, or forward it to the logging facility on another host over the network. The
 4477 logged message includes a message header and a message body. The message header contains
 4478 at least a timestamp and a tag string.

4479 The message body is generated from the *message* and following arguments in the same manner
 4480 as if these were arguments to *printf()*, except that occurrences of %m in the format string
 4481 pointed to by the *message* argument are replaced by the error message string associated with the
 4482 current value of *errno*. A trailing newline character is added if needed.

4483 Values of the *priority* argument are formed by ORing together a severity level value and an
 4484 optional facility value. If no facility value is specified, the current default facility value is used.

4485 Possible values of severity level include:

4486	LOG_EMERG	A panic condition.
4487	LOG_ALERT	A condition that should be corrected immediately, such as a corrupted system
4488		database.
4489	LOG_CRIT	Critical conditions, such as hard device errors.
4490	LOG_ERR	Errors.
4491	LOG_WARNING	
4492		Warning messages.
4493	LOG_NOTICE	Conditions that are not error conditions, but that may require special
4494		handling.
4495	LOG_INFO	Informational messages.
4496	LOG_DEBUG	Messages that contain information normally of use only when debugging a
4497		program.

4498 The facility indicates the application or system component generating the message. Possible
 4499 facility values include:

4500	LOG_USER	Messages generated by random processes. This is the default facility identifier
4501		if none is specified.
4502	LOG_LOCAL0	Reserved for local use.
4503	LOG_LOCAL1	Reserved for local use.
4504	LOG_LOCAL2	Reserved for local use.

4505 LOG_LOCAL3 Reserved for local use.

4506 LOG_LOCAL4 Reserved for local use.

4507 LOG_LOCAL5 Reserved for local use.

4508 LOG_LOCAL6 Reserved for local use.

4509 LOG_LOCAL7 Reserved for local use.

4510 The *openlog()* function sets process attributes that affect subsequent calls to *syslog()*. The *ident*
 4511 argument is a string that is prepended to every message. The *logopt* argument indicates logging
 4512 options. Values for *logopt* are constructed by a bitwise-inclusive OR of zero or more of the
 4513 following:

4514 LOG_PID Log the process ID with each message. This is useful for identifying specific
 4515 processes.

4516 LOG_CONS Write messages to the system console if they cannot be sent to the logging
 4517 facility. The *syslog()* function ensures that the process does not acquire the
 4518 console as a controlling terminal in the process of writing the message.

4519 LOG_NDELAY Open the connection to the logging facility immediately. Normally the open
 4520 is delayed until the first message is logged. This is useful for programs that
 4521 need to manage the order in which file descriptors are allocated.

4522 LOG_ODELAY Delay open until *syslog()* is called.

4523 LOG_NOWAIT Do not wait for child processes that may have been created during the course
 4524 of logging the message. This option should be used by processes that enable
 4525 notification of child termination using SIGCHLD, since *syslog()* may
 4526 otherwise block waiting for a child whose exit status has already been
 4527 collected.

4528 The *facility* argument encodes a default facility to be assigned to all messages that do not have
 4529 an explicit facility already encoded. The initial default facility is LOG_USER.

4530 The *openlog()* and *syslog()* functions may allocate a file descriptor. It is not necessary to call
 4531 *openlog()* prior to calling *syslog()*.

4532 The *closelog()* function closes any open file descriptors allocated by previous calls to *openlog()* or
 4533 *syslog()*.

4534 The *setlogmask()* function sets the log priority mask for the current process to *maskpri* and
 4535 returns the previous mask. If the *maskpri* argument is 0, the current log mask is not modified.
 4536 Calls by the current process to *syslog()* with a priority not set in *maskpri* are rejected. The default
 4537 log mask allows all priorities to be logged. A call to *openlog* is not required prior to calling
 4538 *setlogmask()*.

4539 Symbolic constants for use as values of the *logopt*, *facility*, *priority*, and *maskpri* arguments are
 4540 defined in the <syslog.h> header.

4541 **RETURN VALUE**

4542 The *setlogmask()* function returns the previous log priority mask. The *closelog()*, *openlog()* and
 4543 *syslog()* functions return no value.

4544 **ERRORS**

4545 No errors are defined.

4546 **EXAMPLES**

4547 None.

4548 **APPLICATION USAGE**

4549 None.

4550 **FUTURE DIRECTIONS**

4551 None.

4552 **SEE ALSO**4553 *printf()*, <syslog.h>.4554 **CHANGE HISTORY**

4555 First released in Issue 4, Version 2.

4556 **Issue 5**

4557 Moved from X/OPEN UNIX extension to BASE.

4558 **NAME**4559 compile — produce a compiled regular expression (**LEGACY**)4560 **SYNOPSIS**4561 EX `#include <regex.h>`

```
4562 char *compile(char *instring, char *expbuf,
4563               const char *endbuf, int eof);
4564
```

4565 **DESCRIPTION**4566 Refer to *regex()*.4567 **CHANGE HISTORY**

4568 First released in Issue 2.

4569 Derived from Issue 2 of the SVID.

4570 **Issue 4**

4571 The following changes are incorporated in this issue:

- 4572 • The **<regex.h>** header is added to the SYNOPSIS section.
- 4573 • The type of argument *endbuf* is changed from **char *** to **const char ***.
- 4574 • The interface is marked TO BE WITHDRAWN, because improved functionality is now
- 4575 provided by interfaces introduced for alignment with the ISO POSIX-2 standard.

4576 **Issue 5**

4577 Marked LEGACY.

4578 **NAME**

4579 confstr — get configurable variables

4580 **SYNOPSIS**

4581 #include <unistd.h>

4582 size_t confstr(int name, char *buf, size_t len);

4583 **DESCRIPTION**

4584 The *confstr()* function provides a method for applications to get configuration-defined string values. Its use and purpose are similar to *sysconf()*, but it is used where string values rather than numeric values are returned.

4587 The *name* argument represents the system variable to be queried. The implementation supports the following name values, defined in <unistd.h>. It may support others:

4589 _CS_PATH
 4590 EX _CS_XBS5_ILP32_OFF32_CFLAGS
 4591 _CS_XBS5_ILP32_OFF32_LDFLAGS
 4592 _CS_XBS5_ILP32_OFF32_LIBS
 4593 _CS_XBS5_ILP32_OFF32_LINTFLAGS
 4594 _CS_XBS5_ILP32_OFFBIG_CFLAGS
 4595 _CS_XBS5_ILP32_OFFBIG_LDFLAGS
 4596 _CS_XBS5_ILP32_OFFBIG_LIBS
 4597 _CS_XBS5_ILP32_OFFBIG_LINTFLAGS
 4598 _CS_XBS5_LP64_OFF64_CFLAGS
 4599 _CS_XBS5_LP64_OFF64_LDFLAGS
 4600 _CS_XBS5_LP64_OFF64_LIBS
 4601 _CS_XBS5_LP64_OFF64_LINTFLAGS
 4602 _CS_XBS5_LPBIG_OFFBIG_CFLAGS
 4603 _CS_XBS5_LPBIG_OFFBIG_LDFLAGS
 4604 _CS_XBS5_LPBIG_OFFBIG_LIBS
 4605 _CS_XBS5_LPBIG_OFFBIG_LINTFLAGS
 4606

4607 If *len* is not 0, and if *name* has a configuration-defined value, *confstr()* copies that value into the *len*-byte buffer pointed to by *buf*. If the string to be returned is longer than *len* bytes, including the terminating null, then *confstr()* truncates the string to *len*−1 bytes and null-terminates the result. The application can detect that the string was truncated by comparing the value returned by *confstr()* with *len*.

4612 If *len* is 0 and *buf* is a null pointer, then *confstr()* still returns the integer value as defined below, but does not return a string. If *len* is 0 but *buf* is not a null pointer, the result is unspecified.

4614 **RETURN VALUE**

4615 If *name* has a configuration-defined value, *confstr()* returns the size of buffer that would be needed to hold the entire configuration-defined value. If this return value is greater than *len*, the string returned in *buf* is truncated.

4618 If *name* is invalid, *confstr()* returns 0 and sets *errno* to indicate the error.

4619 If *name* does not have a configuration-defined value, *confstr()* returns 0 and leaves *errno* unchanged.

4621 **ERRORS**4622 The *confstr()* function will fail if:4623 [EINVAL] The value of the *name* argument is invalid.4624 **EXAMPLES**

4625 None.

4626 **APPLICATION USAGE**

4627 An application can distinguish between an invalid *name* parameter value and one that
 4628 corresponds to a configurable variable that has no configuration-defined value by checking if
 4629 *errno* is modified. This mirrors the behaviour of *sysconf()*.

4630 The original need for this function was to provide a way of finding the configuration-defined
 4631 default value for the environment variable *PATH*. Since *PATH* can be modified by the user to
 4632 include directories that could contain utilities replacing **XCU** specification standard utilities,
 4633 applications need a way to determine the system-supplied *PATH* environment variable value
 4634 that contains the correct search path for the standard utilities.

4635 An application could use:

4636 `confstr(name, (char *)NULL, (size_t)0)`

4637 to find out how big a buffer is needed for the string value; use *malloc()* to allocate a buffer to
 4638 hold the string; and call *confstr()* again to get the string. Alternately, it could allocate a fixed,
 4639 static buffer that is big enough to hold most answers (perhaps 512 or 1024 bytes), but then use
 4640 *malloc()* to allocate a larger buffer if it finds that this is too small.

4641 **FUTURE DIRECTIONS**

4642 None.

4643 **SEE ALSO**4644 *pathconf()*, *sysconf()*, `<unistd.h>`, the **XCU** specification of *getconf*.4645 **CHANGE HISTORY**

4646 First released in Issue 4.

4647 Derived from the ISO POSIX-2 standard.

4648 **Issue 5**

4649 A table indicating the permissible values of *name* are added to the DESCRIPTION. All those
 4650 marked EX are new in this issue.

4651 **NAME**

4652 cos — cosine function

4653 **SYNOPSIS**

4654 #include <math.h>

4655 double cos(double x);

4656 **DESCRIPTION**4657 The *cos()* function computes the cosine of *x*, measured in radians.

4658 An application wishing to check for error situations should set *errno* to 0 before calling *cos()*. If
 4659 *errno* is non-zero on return, or the returned value is NaN, an error has occurred.

4660 **RETURN VALUE**4661 Upon successful completion, *cos()* returns the cosine of *x*.4662 EX If *x* is NaN, NaN is returned and *errno* may be set to [EDOM].

4663 EX If *x* is $\pm\text{Inf}$, either 0 is returned and *errno* is set to [EDOM], or NaN is returned and *errno* may be
 4664 set to [EDOM].

4665 If the result underflows, 0 is returned and *errno* may be set to [ERANGE].4666 **ERRORS**4667 The *cos()* function may fail if:4668 EX [EDOM] The value of *x* is NaN or *x* is $\pm\text{Inf}$.

4669 [ERANGE] The result underflows.

4670 EX No other errors will occur.

4671 **EXAMPLES**

4672 None.

4673 **APPLICATION USAGE**4674 The *cos()* function may lose accuracy when its argument is far from 0.4675 **FUTURE DIRECTIONS**

4676 None.

4677 **SEE ALSO**4678 *acos()*, *isnan()*, *sin()*, *tan()*, <math.h>.4679 **CHANGE HISTORY**

4680 First released in Issue 1.

4681 Derived from Issue 1 of the SVID.

4682 **Issue 4**

4683 The following changes are incorporated in this issue:

- 4684 • Removed references to *matherr()*.
- 4685 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
 4686 the ISO C standard and to rationalise error handling in the mathematics functions.
- 4687 • The return value specified for [EDOM] is marked as an extension.

4688 **Issue 5**

4689 The DESCRIPTION is updated to indicate how an application should check for an error. This
 4690 text was previously published in the APPLICATION USAGE section.

4691 **NAME**

4692 cosh — hyperbolic cosine function

4693 **SYNOPSIS**

4694 #include <math.h>

4695 double cosh(double x);

4696 **DESCRIPTION**4697 The *cosh()* function computes the hyperbolic cosine of *x*.4698 An application wishing to check for error situations should set *errno* to 0 before calling *cosh()*. If
4699 *errno* is non-zero on return, or the returned value is NaN, an error has occurred.4700 **RETURN VALUE**4701 Upon successful completion, *cosh()* returns the hyperbolic cosine of *x*.4702 If the result would cause an overflow, HUGE_VAL is returned and *errno* is set to [ERANGE].4703 EX If *x* is NaN, NaN is returned and *errno* may be set to [EDOM].4704 **ERRORS**4705 The *cosh()* function will fail if:

4706 [ERANGE] The result would cause an overflow.

4707 The *cosh()* function may fail if:4708 EX [EDOM] The value of *x* is NaN.

4709 EX No other errors will occur.

4710 **EXAMPLES**

4711 None.

4712 **APPLICATION USAGE**

4713 None.

4714 **FUTURE DIRECTIONS**

4715 None.

4716 **SEE ALSO**4717 *acosh()*, *isnan()*, *sinh()*, *tanh()*, <math.h>.4718 **CHANGE HISTORY**

4719 First released in Issue 1.

4720 Derived from Issue 1 of the SVID.

4721 **Issue 4**

4722 The following changes are incorporated in this issue:

- 4723 • Removed references to *matherr()*.
- 4724 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
4725 the ISO C standard and to rationalise error handling in the mathematics functions.
- 4726 • The return value specified for [EDOM] is marked as an extension.

4727 **Issue 5**

4728 The DESCRIPTION is updated to indicate how an application should check for an error. This
4729 text was previously published in the APPLICATION USAGE section.

4730 **NAME**

4731 creat — create a new file or rewrite an existing one

4732 **SYNOPSIS**

4733 OH #include <sys/types.h>

4734 #include <sys/stat.h>

4735 #include <fcntl.h>

4736 int creat(const char *path, mode_t mode);

4737 **DESCRIPTION**

4738 The function call:

4739 creat(path, mode)

4740 is equivalent to:

4741 open(path, O_WRONLY|O_CREAT|O_TRUNC, mode)

4742 **RETURN VALUE**4743 Refer to *open()*.4744 **ERRORS**4745 Refer to *open()*.4746 **EXAMPLES**

4747 None.

4748 **APPLICATION USAGE**

4749 None.

4750 **FUTURE DIRECTIONS**

4751 None.

4752 **SEE ALSO**4753 *open()*, <fcntl.h>, <sys/stat.h>, <sys/types.h>.4754 **CHANGE HISTORY**

4755 First released in Issue 1.

4756 Derived from Issue 1 of the SVID.

4757 **Issue 4**

4758 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 4759
- The type of argument *path* is changed from **char *** to **const char ***.

4760 Other changes are incorporated as follows:

- 4761
- 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.
- 4762

4763 **NAME**4764 crypt — string encoding function (**CRYPT**)4765 **SYNOPSIS**4766 EX `#include <unistd.h>`4767 `char *crypt (const char *key, const char *salt);`

4768

4769 **DESCRIPTION**4770 The *crypt()* function is a string encoding function. The algorithm is implementation-dependent.4771 The *key* argument points to a string to be encoded. The *salt* argument is a string chosen from the
4772 set:

4773 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

4774 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

4775 0 1 2 3 4 5 6 7 8 9 . /

4776 The first two characters of this string may be used to perturb the encoding algorithm.

4777 The return value of *crypt()* points to static data that is overwritten by each call.

4778 This need not be a reentrant function.

4779 **RETURN VALUE**4780 Upon successful completion, *crypt()* returns a pointer to the encoded string. The first two
4781 characters of the returned value are those of the *salt* argument.4782 Otherwise it returns a null pointer and sets *errno* to indicate the error.4783 **ERRORS**4784 The *crypt()* function will fail if:

4785 [ENOSYS] The functionality is not supported on this implementation.

4786 **EXAMPLES**

4787 None.

4788 **APPLICATION USAGE**

4789 The values returned by this function need not be portable among XSI-conformant systems.

4790 **FUTURE DIRECTIONS**

4791 None.

4792 **SEE ALSO**4793 *encrypt()*, *setkey()*, **<unistd.h>**.4794 **CHANGE HISTORY**

4795 First released in Issue 1.

4796 Derived from Issue 1 of the SVID.

4797 **Issue 4**

4798 The following changes are incorporated in this issue:

- 4799 • The **<unistd.h>** header is added to the SYNOPSIS section.
- 4800 • The type of arguments *key* and *salt* are changed from **char *** to **const char ***.
- 4801 • The DESCRIPTION now explicitly defines the characters that can appear in the *salt*
- 4802 argument.

4803 **Issue 5**

4804 Normative text previously in the APPLICATION USAGE section is moved to the
4805 DESCRIPTION.

4806 **NAME**

4807 ctermid — generate a pathname for controlling terminal

4808 **SYNOPSIS**

4809 #include <stdio.h>

4810 char *ctermid(char *s);

4811 **DESCRIPTION**

4812 The *ctermid()* function generates a string that, when used as a pathname, refers to the current
4813 controlling terminal for the current process. If *ctermid()* returns a pathname, access to the file is
4814 not guaranteed.

4815 If the application uses any of the `_POSIX_THREAD_SAFE_FUNCTIONS` or `_POSIX_THREADS`
4816 interfaces, the *ctermid()* function must be called with a non-NULL parameter.

4817 **RETURN VALUE**

4818 If *s* is a null pointer, the string is generated in an area that may be static (and therefore may be
4819 overwritten by each call), the address of which is returned. Otherwise *s* is assumed to point to a
4820 character array of at least `{L_ctermid}` bytes; the string is placed in this array and the value of *s* is
4821 returned. The symbolic constant `{L_ctermid}` is defined in <stdio.h>, and will have a value
4822 greater than 0.

4823 The *ctermid()* function will return an empty string if the pathname that would refer to the
4824 controlling terminal cannot be determined, or if the function is unsuccessful.

4825 **ERRORS**

4826 No errors are defined.

4827 **EXAMPLES**

4828 None.

4829 **APPLICATION USAGE**

4830 The difference between *ctermid()* and *ttyname()* is that *ttyname()* must be handed a file
4831 descriptor and returns a path of the terminal associated with that file descriptor, while *ctermid()*
4832 returns a string (such as */dev/tty*) that will refer to the current controlling terminal if used as a
4833 pathname.

4834 **FUTURE DIRECTIONS**

4835 None.

4836 **SEE ALSO**4837 *ttyname()*, <stdio.h>.4838 **CHANGE HISTORY**

4839 First released in Issue 1.

4840 Derived from Issue 1 of the SVID.

4841 **Issue 4**

4842 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 4843 • The DESCRIPTION and RETURN VALUE sections, though functionally identical to Issue 3,
4844 are rewritten.

4845 **Issue 5**

4846 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

4847 **NAME**

4848 ctime, ctime_r — convert a time value to date and time string

4849 **SYNOPSIS**

4850 #include <time.h>

4851 char *ctime(const time_t *clock);

4852 char *ctime_r(const time_t *clock, char *buf);

4853 **DESCRIPTION**4854 The *ctime()* function converts 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:

4856 asctime(localtime(clock))

4857 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.4860 The *ctime()* interface need not be reentrant.4861 The *ctime_r()* function converts 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 returns *buf*.4864 Unlike *ctime()*, the thread-safe version *ctime_r()* is not required to set *tzname*.4865 **RETURN VALUE**4866 The *ctime()* function returns the pointer returned by *asctime()* with that broken-down time as an argument.4868 On successful completion, *ctime_r()* returns a pointer to the string pointed to by *buf*. When an error is encountered, a NULL pointer is returned.4870 **ERRORS**

4871 No errors are defined.

4872 **EXAMPLES**

4873 None.

4874 **APPLICATION USAGE**4875 Values for the broken-down time structure can be obtained by calling *gmtime()* or *localtime()*.
4876 This interface is included for compatibility with older implementations, and does not support
4877 localised date and time formats. Applications should use the *strftime()* interface to achieve
4878 maximum portability.4879 **FUTURE DIRECTIONS**

4880 None.

4881 **SEE ALSO**4882 *asctime()*, *clock()*, *difftime()*, *gmtime()*, *localtime()*, *mktime()*, *strftime()*, *strptime()*, *time()*, *utime()*,
4883 <time.h>.4884 **CHANGE HISTORY**

4885 First released in Issue 1.

4886 Derived from Issue 1 of the SVID.

4887 Issue 4

4888 The following change is incorporated for alignment with the ISO C standard:

- 4889 • The type of argument *clock* is changed from **time_t*** to **const time_t***.

4890 Another change is incorporated as follows:

- 4891 • The APPLICATION USAGE section is expanded to describe the time-handling functions
4892 generally and to refer users to *strftime()*, which is a locale-dependent time-handling function.

4893 Issue 5

4894 Normative text previously in the APPLICATION USAGE section is moved to the
4895 DESCRIPTION.

4896 The *ctime_r()* function is included for alignment with the POSIX Threads Extension.

4897 A note indicating that the *ctime()* interface need not be reentrant is added to the DESCRIPTION.

4898 **NAME**4899 cuserid — character login name of the user (**LEGACY**)4900 **SYNOPSIS**

4901 EX #include <stdio.h>

4902 char *cuserid(char *s);

4903

4904 **DESCRIPTION**4905 The *cuserid()* function generates a character representation of the name associated with the real
4906 or effective user ID of the process.4907 If *s* is a null pointer, this representation is generated in an area that may be static (and thus
4908 overwritten by subsequent calls to *cuserid()*), the address of which is returned. If *s* is not a null
4909 pointer, *s* is assumed to point to an array of at least {L_cuserid} bytes; the representation is
4910 deposited in this array. The symbolic constant {L_cuserid} is defined in <stdio.h> and has a
4911 value greater than 0.4912 If the application uses any of the _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS
4913 interfaces, the *cuserid()* function must be called with a non-NULL parameter.4914 **RETURN VALUE**4915 If *s* is not a null pointer, *s* is returned. If *s* is not a null pointer and the login name cannot be
4916 found, the null byte '\0' will be placed at **s*. If *s* is a null pointer and the login name cannot be
4917 found, *cuserid()* returns a null pointer. If *s* is a null pointer and the login name can be found, the
4918 address of a buffer (possibly static) containing the login name is returned.4919 **ERRORS**

4920 No errors are defined.

4921 **EXAMPLES**

4922 None.

4923 **APPLICATION USAGE**4924 The functionality of *cuserid()* defined in the POSIX.1-1988 standard (and Issue 3 of this
4925 specification) differs from that of historical implementations (and Issue 2 of this specification).
4926 In the ISO POSIX-1 standard, *cuserid()* is removed completely. In this specification, therefore,
4927 both functionalities are allowed.

4928 The Issue 2 functionality can be obtained by using:

4929 getpwuid(getuid())

4930 The Issue 3 functionality can be obtained by using:

4931 getpwuid(geteuid())

4932 **FUTURE DIRECTIONS**

4933 None.

4934 **SEE ALSO**

4935 getlogin(), getpwnam(), getpwuid(), getuid(), geteuid(), <stdio.h>.

4936 **CHANGE HISTORY**

4937 First released in Issue 1.

4938 Derived from System V Release 2.0.

4939 **Issue 4**

4940 The following changes are incorporated in this issue:

- 4941 • The interface is marked TO BE WITHDRAWN, because of differences between the historical
4942 definition of this interface and the definition published in the POSIX.1-1988 standard (and
4943 hence Issue 3). The interface has also been removed from the ISO POSIX-1 standard.
- 4944 • The interface is now marked as an extension.
- 4945 • The DESCRIPTION is changed to indicate that an implementation can determine the name
4946 returned by the function from the real or effective user ID of the process. |
- 4947 • The APPLICATION USAGE section is rewritten to describe the historical development of
4948 this interface, and to indicate transition between this and previous issues. |
- 4949 • The RETURN VALUE section has been expanded. |

4950 **Issue 5**

4951 Marked LEGACY. |

4952 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

4953 **NAME**
4954 daylight — daylight savings time flag

4955 **SYNOPSIS**

```
4956 EX      #include <time.h>  
4957          extern int daylight;  
4958
```

4959 **DESCRIPTION**

4960 Refer to `tzset()`.

4961 **CHANGE HISTORY**

4962 First released in Issue 1.

4963 Derived from Issue 1 of the SVID.

4964 NAME

4965 dbm_clearerr, dbm_close, dbm_delete, dbm_error, dbm_fetch, dbm_firstkey, dbm_nextkey,
 4966 dbm_open, dbm_store — database functions

4967 SYNOPSIS

```
4968 EX #include <ndbm.h>

4969 int dbm_clearerr(DBM *db);
4970 void dbm_close(DBM *db);
4971 int dbm_delete(DBM *db, datum key);
4972 int dbm_error(DBM *db);
4973 datum dbm_fetch(DBM *db, datum key);
4974 datum dbm_firstkey(DBM *db);
4975 datum dbm_nextkey(DBM *db);
4976 DBM *dbm_open(const char *file, int open_flags, mode_t file_mode);
4977 int dbm_store(DBM *db, datum key, datum content, int store_mode);
4978
```

4979 DESCRIPTION

4980 These functions create, access and modify a database.

4981 A **datum** consists of at least two members, **dptr** and **dsize**. The **dptr** member points to an object
 4982 that is **dsize** bytes in length. Arbitrary binary data, as well as character strings, may be stored in
 4983 the object pointed to by **dptr**.

4984 The database is stored in two files. One file is a directory containing a bit map of keys and has
 4985 **.dir** as its suffix. The second file contains all data and has **.pag** as its suffix.

4986 The *dbm_open()* function opens a database. The *file* argument to the function is the pathname of
 4987 the database. The function opens two files named *file.dir* and *file.pag*. The *open_flags* argument
 4988 has the same meaning as the *flags* argument of *open()* except that a database opened for write-
 4989 only access opens the files for read and write access and the behaviour of the O_APPEND flag is
 4990 unspecified. The *file_mode* argument has the same meaning as the third argument of *open()*.

4991 The *dbm_close()* function closes a database. The argument *db* must be a pointer to a **dbm**
 4992 structure that has been returned from a call to *dbm_open()*.

4993 The *dbm_fetch()* function reads a record from a database. The argument *db* is a pointer to a
 4994 database structure that has been returned from a call to *dbm_open()*. The argument *key* is a
 4995 **datum** that has been initialised by the application program to the value of the key that matches
 4996 the key of the record the program is fetching.

4997 The *dbm_store()* function writes a record to a database. The argument *db* is a pointer to a
 4998 database structure that has been returned from a call to *dbm_open()*. The argument *key* is a
 4999 **datum** that has been initialised by the application program to the value of the key that identifies
 5000 (for subsequent reading, writing or deleting) the record the program is writing. The argument
 5001 *content* is a **datum** that has been initialised by the application program to the value of the record
 5002 the program is writing. The argument *store_mode* controls whether *dbm_store()* replaces any
 5003 pre-existing record that has the same key that is specified by the *key* argument. The application
 5004 program must set *store_mode* to either DBM_INSERT or DBM_REPLACE. If the database
 5005 contains a record that matches the *key* argument and *store_mode* is DBM_REPLACE, the existing
 5006 record is replaced with the new record. If the database contains a record that matches the *key*
 5007 argument and *store_mode* is DBM_INSERT, the existing record is not replaced with the new
 5008 record. If the database does not contain a record that matches the *key* argument and *store_mode*
 5009 is either DBM_INSERT or DBM_REPLACE, the new record is inserted in the database.

5010 The sum of the sizes of a key/content pair must not exceed the internal block size. Moreover, all
 5011 key/content pairs that hash together must fit on a single block. The *dbm_store()* function returns
 5012 an error in the event that a disk block fills with inseparable data.

5013 The *dbm_delete()* function deletes a record and its key from the database. The argument *db* is a
 5014 pointer to a database structure that has been returned from a call to *dbm_open()*. The argument
 5015 *key* is a **datum** that has been initialised by the application program to the value of the key that
 5016 identifies the record the program is deleting.

5017 The *dbm_firstkey()* function returns the first key in the database. The argument *db* is a pointer to
 5018 a database structure that has been returned from a call to *dbm_open()*.

5019 The *dbm_nextkey()* function returns the next key in the database. The argument *db* is a pointer to
 5020 a database structure that has been returned from a call to *dbm_open()*. The *dbm_firstkey()*
 5021 function must be called before calling *dbm_nextkey()*. Subsequent calls to *dbm_nextkey()* return
 5022 the next key until all of the keys in the database have been returned.

5023 The *dbm_error()* function returns the error condition of the database. The argument *db* is a
 5024 pointer to a database structure that has been returned from a call to *dbm_open()*.

5025 The *dbm_clearerr()* function clears the error condition of the database. The argument *db* is a
 5026 pointer to a database structure that has been returned from a call to *dbm_open()*.

5027 These database functions support key/content pairs of at least 1023 bytes.

5028 The **dp**tr pointers returned by these functions may point into static storage that may be changed
 5029 by subsequent calls.

5030 These interfaces need not be reentrant.

5031 RETURN VALUE

5032 The *dbm_store()* and *dbm_delete()* functions return 0 when they succeed and a negative value
 5033 when they fail.

5034 The *dbm_store()* function returns 1 if it is called with a *flags* value of DBM_INSERT and the
 5035 function finds an existing record with the same key.

5036 The *dbm_error()* function returns 0 if the error condition is not set and returns a non-zero value if
 5037 the error condition is set.

5038 The return value of *dbm_clearerr()* is unspecified .

5039 The *dbm_firstkey()* and *dbm_nextkey()* functions return a key **datum**. When the end of the
 5040 database is reached, the **dp**tr member of the key is a null pointer. If an error is detected, the **dp**tr
 5041 member of the key is a null pointer and the error condition of the database is set.

5042 The *dbm_fetch()* function returns a content **datum**. If no record in the database matches the key
 5043 or if an error condition has been detected in the database, the **dp**tr member of the content is a
 5044 null pointer.

5045 The *dbm_open()* function returns a pointer to a database structure. If an error is detected during
 5046 the operation, *dbm_open()* returns a (**DBM ***)0.

5047 ERRORS

5048 No errors are defined.

5049 EXAMPLES

5050 None.

5051 **APPLICATION USAGE**

5052 The following code can be used to traverse the database:

5053

```
for(key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))
```

5054 The *dbm_* functions provided in this library should not be confused in any way with those of a
5055 general-purpose database management system. These functions do not provide for multiple
5056 search keys per entry, they do not protect against multi-user access (in other words they do not
5057 lock records or files), and they do not provide the many other useful database functions that are
5058 found in more robust database management systems. Creating and updating databases by use
5059 of these functions is relatively slow because of data copies that occur upon hash collisions.
5060 These functions are useful for applications requiring fast lookup of relatively static information
5061 that is to be indexed by a single key.

5062 The *dbm_delete()* function need not physically reclaim file space, although it does make it
5063 available for reuse by the database.

5064 After calling *dbm_store()* or *dbm_delete()* during a pass through the keys by *dbm_firstkey()* and
5065 *dbm_nextkey()*, the application should reset the database by calling *dbm_firstkey()* before again
5066 calling *dbm_nextkey()*. The contents of these files are unspecified and may not be portable.

5067 **FUTURE DIRECTIONS**

5068 None.

5069 **SEE ALSO**5070 *open()*, <ndbm.h>.5071 **CHANGE HISTORY**

5072 First released in Issue 4, Version 2.

5073 **Issue 5**

5074 Moved from X/OPEN UNIX extension to BASE.

5075 Normative text previously in the APPLICATION USAGE section is moved to the
5076 DESCRIPTION.

5077 A note indicating that these interfaces need not be reentrant is added to the DESCRIPTION.

5078 **NAME**
5079 difftime — compute the difference between two calendar time values

5080 **SYNOPSIS**
5081 #include <time.h>
5082 double difftime(time_t *time1*, time_t *time0*);

5083 **DESCRIPTION**
5084 The *difftime()* function computes the difference between two calendar times (as returned by
5085 *time()*): *time1* − *time0*.

5086 **RETURN VALUE**
5087 The *difftime()* function returns the difference expressed in seconds as a type **double**.

5088 **ERRORS**
5089 No errors are defined.

5090 **EXAMPLES**
5091 None.

5092 **APPLICATION USAGE**
5093 None.

5094 **FUTURE DIRECTIONS**
5095 None.

5096 **SEE ALSO**
5097 *asctime()*, *clock()*, *ctime()*, *gmtime()*, *localtime()*, *mktime()*, *strftime()*, *strptime()*, *time()*, *utime()*,
5098 <time.h>.

5099 **CHANGE HISTORY**
5100 First released in Issue 4.
5101 Derived from the ISO C standard.

5102 NAME

5103 dirname — report the parent directory name of a file pathname

5104 SYNOPSIS

5105 EX #include <libgen.h>

5106 char *dirname(char *path);

5107

5108 DESCRIPTION

5109 The *dirname()* function takes a pointer to a character string that contains a pathname, and
5110 returns a pointer to a string that is a pathname of the parent directory of that file. Trailing '/'
5111 characters in the path are not counted as part of the path.

5112 If *path* does not contain a '/', then *dirname()* returns a pointer to the string ".". If *path* is a null
5113 pointer or points to an empty string, *dirname()* returns a pointer to the string ".".

5114 This interface need not be reentrant.

5115 RETURN VALUE

5116 The *dirname()* function returns a pointer to a string that is the parent directory of *path*. If *path* is
5117 a null pointer or points to an empty string, a pointer to a string "." is returned.

5118 The *dirname()* function may modify the string pointed to by *path*, and may return a pointer to
5119 static storage that may then be overwritten by subsequent calls to *dirname()*.

5120 ERRORS

5121 No errors are defined.

5122 EXAMPLES

5123

5124

5125

5126

5127

5128

5129

5130

Input String	Output String
"/usr/lib"	"/usr"
"/usr/"	"/"
"usr"	."
"/"	"/"
."	."
.."	."

5131 The following code fragment reads a pathname, changes the current working directory to the
5132 parent directory, and opens the file.

```
5133       char path[MAXPATHLEN], *pathcopy;
5134       int fd;
5135       fgets(path, MAXPATHLEN, stdin);
5136       pathcopy = strdup(path);
5137       chdir(dirname(pathcopy));
5138       fd = open(basename(path), O_RDONLY);
```

5139 APPLICATION USAGE

5140 The *dirname()* and *basename()* functions together yield a complete pathname. The expression
5141 *dirname(path)* obtains the pathname of the directory where *basename(path)* is found.

5142 FUTURE DIRECTIONS

5143 None.

5144 SEE ALSO

5145 *basename()*, <libgen.h>.

5146 **CHANGE HISTORY**

5147 First released in Issue 4, Version 2.

5148 **Issue 5**

5149 Moved from X/OPEN UNIX extension to BASE.

5150 Normative text previously in the APPLICATION USAGE section is moved to the
5151 DESCRIPTION.

5152 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

5153 **NAME**
5154 `div` — compute the quotient and remainder of an integer division

5155 **SYNOPSIS**
5156 `#include <stdlib.h>`
5157 `div_t div(int numer, int denom);`

5158 **DESCRIPTION**
5159 The `div()` function computes the quotient and remainder of the division of the numerator *numer*
5160 by the denominator *denom*. If the division is inexact, the resulting quotient is the integer of lesser
5161 magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the
5162 behaviour is undefined; otherwise, *quot* * *denom* + *rem* will equal *numer*.

5163 **RETURN VALUE**
5164 The `div()` function returns a structure of type **div_t**, comprising both the quotient and the
5165 remainder. The structure includes the following members, in any order:

5166 `int quot; /* quotient */`
5167 `int rem; /* remainder */`

5168 **ERRORS**
5169 No errors are defined.

5170 **EXAMPLES**
5171 None.

5172 **APPLICATION USAGE**
5173 None.

5174 **FUTURE DIRECTIONS**
5175 None.

5176 **SEE ALSO**
5177 `ldiv()`, `<stdlib.h>`.

5178 **CHANGE HISTORY**
5179 First released in Issue 4.
5180 Derived from the ISO C standard.

5181 **NAME**

5182 dlclose — close a dlopen() object

5183 **SYNOPSIS**

5184 EX #include <dlfcn.h>

5185 int dlclose(void *handle);

5186

5187 **DESCRIPTION**5188 *dlclose()* is used to inform the system that the object referenced by a *handle* returned from a
5189 previous *dlopen()* invocation is no longer needed by the application.5190 The use of *dlclose()* reflects a statement of intent on the part of the process, but does not create
5191 any requirement upon the implementation, such as removal of the code or symbols referenced
5192 by *handle*. Once an object has been closed using *dlclose()* an application should assume that its
5193 symbols are no longer available to *dlsym()*. All objects loaded automatically as a result of
5194 invoking *dlopen()* on the referenced object are also closed.5195 Although a *dlclose()* operation is not required to remove structures from an address space,
5196 neither is an implementation prohibited from doing so. The only restriction on such a removal is
5197 that no object will be removed to which references have been relocated, until or unless all such
5198 references are removed. For instance, an object that had been loaded with a *dlopen()* operation
5199 specifying the RTLD_GLOBAL flag might provide a target for dynamic relocations performed in
5200 the processing of other objects – in such environments, an application may assume that no
5201 relocation, once made, will be undone or remade unless the object requiring the relocation has
5202 itself been removed.5203 **RETURN VALUE**5204 If the referenced object was successfully closed, *dlclose()* returns 0. If the object could not be
5205 closed, or if *handle* does not refer to an open object, *dlclose()* returns a non-zero value. More
5206 detailed diagnostic information will be available through *dLError()*.5207 **ERRORS**

5208 No errors are defined.

5209 **EXAMPLES**

5210 None.

5211 **APPLICATION USAGE**5212 A portable application will employ a *handle* returned from a *dlopen()* invocation only within a
5213 given scope bracketed by the *dlopen()* and *dlclose()* operations. Implementations are free to use
5214 reference counting or other techniques such that multiple calls to *dlopen()* referencing the same
5215 object may return the same object for *handle*. Implementations are also free to re-use a *handle*.
5216 For these reasons, the value of a *handle* must be treated as an opaque object by the application,
5217 used only in calls to *dlsym()* and *dlclose()*.5218 **FUTURE DIRECTIONS**

5219 None.

5220 **SEE ALSO**5221 *dLError()*, *dlopen()*, *dlsym()*.5222 **CHANGE HISTORY**

5223 First released in Issue 5.

5224 NAME

5225 **dlderror** — get diagnostic information

5226 SYNOPSIS

5227 EX `#include <dldfcn.h>`

5228 `char *dlderror(void);`

5229

5230 DESCRIPTION

5231 *dlderror()* returns a null-terminated character string (with no trailing newline) that describes the
5232 last error that occurred during dynamic linking processing. If no dynamic linking errors have
5233 occurred since the last invocation of *dlderror()*, *dlderror()* returns NULL. Thus, invoking *dlderror()* a
5234 second time, immediately following a prior invocation, will result in NULL being returned.

5235 RETURN VALUE

5236 If successful, *dlderror()* returns a null-terminated character string. Otherwise, NULL is returned.

5237 ERRORS

5238 No errors are defined.

5239 EXAMPLES

5240 None.

5241 APPLICATION USAGE

5242 The messages returned by *dlderror()* may reside in a static buffer that is overwritten on each call
5243 to *dlderror()*. Application code should not write to this buffer. Programs wishing to preserve an
5244 error message should make their own copies of that message. Depending on the application
5245 environment with respect to asynchronous execution events, such as signals or other
5246 asynchronous computation sharing the address space, portable applications should use a critical
5247 section to retrieve the error pointer and buffer.

5248 FUTURE DIRECTIONS

5249 None.

5250 SEE ALSO

5251 *dldclose()*, *dldopen()*, *dldsym()*.

5252 CHANGE HISTORY

5253 First released in Issue 5.

5254 **NAME**

5255 dlopen — gain access to an executable object file

5256 **SYNOPSIS**

5257 EX #include <dlfcn.h>

5258 void *dlopen(const char *file, int mode);

5259

5260 **DESCRIPTION**

5261 *dlopen()* makes an executable object file specified by *file* available to the calling program. The
 5262 class of files eligible for this operation and the manner of their construction are specified by the
 5263 implementation, though typically such files are executable objects such as shared libraries,
 5264 relocatable files or programs. Note that some implementations permit the construction of
 5265 dependencies between such objects that are embedded within files. In such cases, a *dlopen()*
 5266 operation will load such dependencies in addition to the object referenced by *file*.
 5267 Implementations may also impose specific constraints on the construction of programs that can
 5268 employ *dlopen()* and its related services.

5269 A successful *dlopen()* returns a *handle* which the caller may use on subsequent calls to *dlsym()*
 5270 and *dlclose()*. The value of this *handle* should not be interpreted in any way by the caller.

5271 *file* is used to construct a pathname to the object file. If *file* contains a slash character, the *file*
 5272 argument is used as the pathname for the file. Otherwise, *file* is used in an implementation-
 5273 dependent manner to yield a pathname.

5274 If the value of *file* is 0, *dlopen()* provides a *handle* on a global symbol object. This object provides
 5275 access to the symbols from an ordered set of objects consisting of the original program image
 5276 file, together with any objects loaded at program startup as specified by that process image file
 5277 (for example, shared libraries), and the set of objects loaded using a *dlopen()* operation together
 5278 with the RTLD_GLOBAL flag. As the latter set of objects can change during execution, the set
 5279 identified by *handle* can also change dynamically.

5280 Only a single copy of an object file is brought into the address space, even if *dlopen()* is invoked
 5281 multiple times in reference to the file, and even if different pathnames are used to reference the
 5282 file.

5283 The *mode* parameter describes how *dlopen()* will operate upon *file* with respect to the processing
 5284 of relocations and the scope of visibility of the symbols provided within *file*. When an object is
 5285 brought into the address space of a process, it may contain references to symbols whose
 5286 addresses are not known until the object is loaded. These references must be relocated before the
 5287 symbols can be accessed. The *mode* parameter governs when these relocations take place and
 5288 may have the following values:

5289	RTLD_LAZY	Relocations are performed at an implementation-dependent time, ranging
5290		from the time of the <i>dlopen()</i> call until the first reference to a given
5291		symbol occurs. Specifying RTLD_LAZY should improve performance on
5292		implementations supporting dynamic symbol binding as a process may
5293		not reference all of the functions in any given object. And, for systems
5294		supporting dynamic symbol resolution for normal process execution, this
5295		behaviour mimics the normal handling of process execution.

5296	RTLD_NOW	All necessary relocations are performed when the object is first loaded.
5297		This may waste some processing if relocations are performed for
5298		functions that are never referenced. This behaviour may be useful for
5299		applications that need to know as soon as an object is loaded that all
5300		symbols referenced during execution will be 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 startup, 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 bitwise or'ed with one of the following values:

RTLD_GLOBAL The object's symbols are 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 are not made available for the relocation processing of any other object.

If neither RTLD_GLOBAL nor RTLD_LOCAL are specified, then an implementation-specified default behaviour will 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 will 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 will maintain the RTLD_GLOBAL status regardless of any previous or future specification of RTLD_LOCAL, so 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 will 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.

5341 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()* will return NULL. More detailed diagnostic information will be available through *dlerror()*.

5346 ERRORS

No errors are defined.

5348 **EXAMPLES**

5349 None.

5350 **APPLICATION USAGE**

5351 None.

5352 **FUTURE DIRECTIONS**

5353 None.

5354 **SEE ALSO**5355 *dlclose()*, *dLError()*, *dlsym()*.5356 **CHANGE HISTORY**

5357 First released in Issue 5.

5358 **NAME**

5359 dlsym — obtain the address of a symbol from a dlopen() object

5360 **SYNOPSIS**

5361 EX #include <dlfcn.h>

5362 void *dlsym(void *handle, const char *name);

5363

5364 **DESCRIPTION**

5365 *dlsym()* allows a process to obtain the address of a symbol defined within an object made
 5366 accessible through a *dlopen()* call. *handle* is the value returned from a call to *dlopen()* (and which
 5367 has not since been released via a call to *dlclose()*), *name* is the symbol's name as a character
 5368 string.

5369 *dlsym()* will search for the named symbol in all objects loaded automatically as a result of
 5370 loading the object referenced by *handle* (see *dlopen()*). *Load* ordering is used in *dlsym()*
 5371 operations upon the global symbol object. The symbol resolution algorithm used will be
 5372 *dependency* order as described in *dlopen()*.

5373 **RETURN VALUE**

5374 If *handle* does not refer to a valid object opened by *dlopen()*, or if the named symbol cannot be
 5375 found within any of the objects associated with *handle*, *dlsym()* will return NULL. More detailed
 5376 diagnostic information will be available through *dlerror()*.

5377 **ERRORS**

5378 No errors are defined.

5379 **EXAMPLES**

5380 The following example shows how one can use *dlopen()* and *dlsym()* to access either function or
 5381 data objects. For simplicity, error checking has been omitted.

```
5382       void     *handle;
5383       int     *iptr, (*fptr)(int);

5384       /* open the needed object */
5385       handle = dlopen("/usr/home/me/libfoo.so.1", RTLD_LAZY);

5386       /* find the address of function and data objects */
5387       fptr = (int (*)(int))dlsym(handle, "my_function");
5388       iptr = (int *)dlsym(handle, "my_object");

5389       /* invoke function, passing value of integer as a parameter */
5390       (*fptr)(*iptr);
```

5391 **APPLICATION USAGE**

5392 Special purpose values for *handle* are reserved for future use. These values and their meanings
 5393 are:

5394 RTLD_NEXT	5395 Specifies the next object after this one that defines <i>name</i> . <i>This one</i> refers to 5396 the object containing the invocation of <i>dlsym()</i> . The <i>next</i> object is the one 5397 found upon the application of a <i>load</i> order symbol resolution algorithm 5398 (see <i>dlopen()</i>). The next object is either one of global scope (because it 5399 was introduced as part of the original process image or because it was 5400 added with a <i>dlopen()</i> operation including the RTLD_GLOBAL flag), or is 5401 an object that was included in the same <i>dlopen()</i> operation that loaded this one.
----------------------	---

5402 The RTLD_NEXT flag is useful to navigate an intentionally created
5403 hierarchy of multiply defined symbols created through *interposition*. For
5404 example, if a program wished to create an implementation of *malloc()*
5405 that embedded some statistics gathering about memory allocations, such
5406 an implementation could use the real *malloc()* definition to perform the
5407 memory allocation – and itself only embed the necessary logic to
5408 implement the statistics gathering function.

5409 **FUTURE DIRECTIONS**

5410 None.

5411 **SEE ALSO**

5412 *dlclose()*, *dlderror()*, *dlopen()*.

5413 **CHANGE HISTORY**

5414 First released in Issue 5.

5415 NAME

5416 drand48, erand48, jrand48, lcong48, lrand48, mrand48, nrand48, seed48, srand48 — generate
 5417 uniformly distributed pseudo-random numbers

5418 SYNOPSIS

```
5419 EX #include <stdlib.h>

5420 double drand48(void);
5421 double erand48(unsigned short int xsubi[3]);
5422 long int jrand48(unsigned short int xsubi[3]);
5423 void lcong48(unsigned short int param[7]);
5424 long int lrand48(void);
5425 long int mrand48(void);
5426 long int nrand48(unsigned short int xsubi[3]);
5427 unsigned short int *seed48(unsigned short int seed16v[3]);
5428 void srand48(long int seedval);
5429
```

5430 DESCRIPTION

5431 This family of functions generates pseudo-random numbers using a linear congruential
 5432 algorithm and 48-bit integer arithmetic.

5433 The *drand48()* and *erand48()* functions return non-negative, double-precision, floating-point
 5434 values, uniformly distributed over the interval [0.0,1.0).

5435 The *lrnd48()* and *nrand48()* functions return non-negative, long integers, uniformly distributed
 5436 over the interval $[0, 2^{31})$.

5437 The *mrnd48()* and *jrnd48()* functions return signed long integers uniformly distributed over
 5438 the interval $[-2^{31}, 2^{31})$.

5439 The *srand48()*, *seed48()* and *lcong48()* are initialisation entry points, one of which should be
 5440 invoked before either *drand48()*, *lrnd48()* or *mrnd48()* is called. (Although it is not
 5441 recommended practice, constant default initialiser values will be supplied automatically if
 5442 *drand48()*, *lrnd48()* or *mrnd48()* is called without a prior call to an initialisation entry point.)
 5443 The *erand48()*, *nrand48()* and *jrnd48()* functions do not require an initialisation entry point to
 5444 be called first.

5445 All the routines work by generating a sequence of 48-bit integer values, X_i , according to the
 5446 linear congruential formula:

$$5447 \quad X_{n+1} = (aX_n + c)_{\text{mod } m} \quad n \geq 0$$

5448 The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless *lcong48()* is
 5449 invoked, the multiplier value a and the addend value c are given by:

$$5450 \quad a = 5\text{DEECE66D}_{16} = 273673163155_8$$

$$5451 \quad c = \text{B}_{16} = 13_8$$

5452 The value returned by any of the *drand48()*, *erand48()*, *jrnd48()*, *lrnd48()*, *mrnd48()* or
 5453 *nrand48()* functions is computed by first generating the next 48-bit X_i in the sequence. Then the
 5454 appropriate number of bits, according to the type of data item to be returned, are copied from
 5455 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 they must be initialised 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 initialised; 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 will *not* depend upon how many times the routines are called to generate numbers for the other streams.

The initialiser 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 initialiser 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-initialise via *seed48()* when the program is restarted.

The initialiser function *lcg48()* 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 *lcg48()* is called, a subsequent call to either *srand48()* or *seed48()* will restore the standard multiplier and addend values, *a* and *c*, specified above.

The *drand48()*, *lrand48()* and *mrand48()* interfaces need not be reentrant.

5481 RETURN VALUE

5482 As described in the DESCRIPTION above.

5483 ERRORS

5484 No errors are defined.

5485 EXAMPLES

5486 None.

5487 APPLICATION USAGE

5488 None.

5489 FUTURE DIRECTIONS

5490 None.

5491 SEE ALSO

5492 *rand()*, <stdlib.h>.

5493 CHANGE HISTORY

5494 First released in Issue 1.

5495 Derived from Issue 1 of the SVID.

5496 Issue 4

5497 The following changes are incorporated in this issue:

- 5498 • The type **long** is replaced by **long int** and the type **unsigned short** is replaced by **unsigned short int** in the SYNOPSIS section.

- 5500 • In the DESCRIPTION, the description of *srand48()* is amended to fix a limitation in Issue 3, |
5501 which indicates that the high-order 32 bits of X_i are set to the {LONG_BIT} bits in the |
5502 argument. Though unintentional, the implication of this statement is that {LONG_BIT} |
5503 would be 32 on all systems compliant with Issue 3, when in fact Issue 3 imposes no such |
5504 restriction. |
- 5505 • The header <stdlib.h> is added to the SYNOPSIS section. |
- 5506 **Issue 5** |
- 5507 A note indicating that these interfaces need not be reentrant is added to the DESCRIPTION. |

5508 **NAME**

5509 dup, dup2 — duplicate an open file descriptor

5510 **SYNOPSIS**

5511 #include <unistd.h>

5512 int dup(int *fildes*);5513 int dup2(int *fildes*, int *fildes2*);5514 **DESCRIPTION**5515 The *dup()* and *dup2()* functions provide an alternative interface to the service provided by *fcntl()* using the F_DUPFD command. The call:5517 fid = dup(*fildes*);

5518 is equivalent to:

5519 fid = fcntl(*fildes*, F_DUPFD, 0);

5520 The call:

5521 fid = dup2(*fildes*, *fildes2*);

5522 is equivalent to:

5523 close(*fildes2*);5524 fid = fcntl(*fildes*, F_DUPFD, *fildes2*);

5525 except for the following:

- 5526 • If *fildes2* is less than 0 or greater than or equal to {OPEN_MAX}, *dup2()* returns -1 with *errno*
- 5527 set to [EBADF].
- 5528 • If *fildes* is a valid file descriptor and is equal to *fildes2*, *dup2()* returns *fildes2* without closing
- 5529 it.
- 5530 • If *fildes* is not a valid file descriptor, *dup2()* returns -1 and does not close *fildes2*.
- 5531 • The value returned is equal to the value of *fildes2* upon successful completion, or is -1 upon
- 5532 failure.

5533 **RETURN VALUE**5534 Upon successful completion a non-negative integer, namely the file descriptor, is returned.
5535 Otherwise, -1 is returned and *errno* is set to indicate the error.5536 **ERRORS**5537 The *dup()* function will fail if:5538 [EBADF] The *fildes* argument is not a valid open file descriptor.5539 [EMFILE] The number of file descriptors in use by this process would exceed
5540 {OPEN_MAX}.5541 The *dup2()* function will fail if:5542 [EBADF] The *fildes* argument is not a valid open file descriptor or the argument *fildes2*
5543 is negative or greater than or equal to {OPEN_MAX}.5544 [EINTR] The *dup2()* function was interrupted by a signal.5545 **EXAMPLES**

5546 None.

5547 **APPLICATION USAGE**

5548 None.

5549 **FUTURE DIRECTIONS**

5550 None.

5551 **SEE ALSO**5552 *close()*, *fcntl()*, *open()*, **<unistd.h>**.5553 **CHANGE HISTORY**

5554 First released in Issue 1.

5555 Derived from Issue 1 of the SVID.

5556 **Issue 4**

5557 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

5558 • In the DESCRIPTION, the fourth bullet item describing differences between *dup()* and
5559 *dup2()* is added.5560 • In the ERRORS section, error values returned by *dup()* and *dup2()* are now described
5561 separately.

5562 Other changes are incorporated as follows:

5563 • The header **<unistd.h>** is added to the SYNOPSIS section.5564 • [EINTR] is no longer required for *dup()* because *fcntl()* does not return [EINTR] for
5565 F_DUPFD.

5566 **NAME**

5567 ecvt, fcvt, gcvt — convert a floating-point number to a string

5568 **SYNOPSIS**

5569 EX #include <stdlib.h>

5570 char *ecvt(double value, int ndigit, int *decpt, int *sign);

5571 char *fcvt(double value, int ndigit, int *decpt, int *sign);

5572 char *gcvt(double value, int ndigit, char *buf);

5573

5574 **DESCRIPTION**5575 The *ecvt()*, *fcvt()* and *gcvt()* functions convert floating-point numbers to null-terminated strings.

5576 *ecvt()* Converts *value* to a null-terminated string of *ndigit* digits (where *ndigit* is reduced to an
 5577 unspecified limit determined by the precision of a **double**) and returns a pointer to the
 5578 string. The high-order digit is non-zero, unless the value is 0. The low-order digit is
 5579 rounded. The position of the radix character relative to the beginning of the string is
 5580 stored in the integer pointed to by *decpt* (negative means to the left of the returned
 5581 digits). If *value* is zero, it is unspecified whether the integer pointed to by *decpt* would
 5582 be 0 or 1. The radix character is not included in the returned string. If the sign of the
 5583 result is negative, the integer pointed to by *sign* is non-zero, otherwise it is 0.

5584 If the converted value is out of range or is not representable, the contents of the
 5585 returned string are unspecified.

5586 *fcvt()* Identical to *ecvt()* except that *ndigit* specifies the number of digits desired after the
 5587 radix point. The total number of digits in the result string is restricted to an unspecified
 5588 limit as determined by the precision of a **double**.

5589 *gcvt()* Converts *value* to a null-terminated string (similar to that of the %g format of *printf()*)
 5590 in the array pointed to by *buf* and returns *buf*. It produces *ndigit* significant digits
 5591 (limited to an unspecified value determined by the precision of a **double**) in %f if
 5592 possible, or %e (scientific notation) otherwise. A minus sign is included in the returned
 5593 string if *value* is less than 0. A radix character is included in the returned string if *value*
 5594 is not a whole number. Trailing zeros are suppressed where *value* is not a whole
 5595 number. The radix character is determined by the current locale. If *setlocale()* has not
 5596 been called successfully, the default locale, "POSIX", is used. The default locale
 5597 specifies a period (.) as the radix character. The LC_NUMERIC category determines
 5598 the value of the radix character within the current locale.

5599 These interfaces need not be reentrant.

5600 **RETURN VALUE**5601 The *ecvt()* and *fcvt()* functions return a pointer to a null-terminated string of digits.5602 The *gcvt()* function returns *buf*.

5603 The return values from *ecvt()* and *fcvt()* may point to static data which may be overwritten by
 5604 subsequent calls to these functions.

5605 **ERRORS**

5606 No errors are defined.

5607 **EXAMPLES**

5608 None.

5609 **APPLICATION USAGE**5610 For portability to implementations conforming to earlier versions of this specification, *sprintf()* is

5611 preferred over this function.

5612 **FUTURE DIRECTIONS**

5613 None.

5614 **SEE ALSO**

5615 *printf()*, *setlocale()*, *<stdlib.h>*.

5616 **CHANGE HISTORY**

5617 First released in Issue 4, Version 2.

5618 **Issue 5**

5619 Moved from X/OPEN UNIX extension to BASE.

5620 Normative text previously in the APPLICATION USAGE section is moved to the
5621 DESCRIPTION.

5622 A note indicating that these interfaces need not be reentrant is added to the DESCRIPTION.

5623 **NAME**5624 encrypt — encoding function (**CRYPT**)5625 **SYNOPSIS**

5626 EX #include <unistd.h>

5627 void encrypt(char block[64], int edflag);

5628

5629 **DESCRIPTION**

5630 The *encrypt()* function provides (rather primitive) access to an implementation-dependent
 5631 encoding algorithm. The key generated by *setkey()* is used to encrypt the string *block* with
 5632 *encrypt()*.

5633 The *block* argument to *encrypt()* is an array of length 64 bytes containing only the bytes with
 5634 numerical value of 0 and 1. The array is modified in place to a similar array using the key set by
 5635 *setkey()*. If *edflag* is 0, the argument is encoded. If *edflag* is 1, the argument may be decoded (see
 5636 the APPLICATION USAGE section below); if the argument is not decoded, *errno* will be set to
 5637 [ENOSYS].

5638 The *encrypt()* function will not change the setting of *errno* if successful.

5639 This interface need not be reentrant.

5640 **RETURN VALUE**

5641 The *encrypt()* function returns no value.

5642 **ERRORS**

5643 The *encrypt()* function will fail if:

5644 [ENOSYS] The functionality is not supported on this implementation.

5645 **EXAMPLES**

5646 None.

5647 **APPLICATION USAGE**

5648 In some environments, decoding might not be implemented. This is related to U.S. Government
 5649 restrictions on encryption and decryption routines: the DES decryption algorithm cannot be
 5650 exported outside the U.S.A. Historical practice has been to ship a different version of the
 5651 encryption library without the decryption feature in the routines supplied. Thus the exported
 5652 version of *encrypt()* does encoding but not decoding.

5653 **FUTURE DIRECTIONS**

5654 None.

5655 **SEE ALSO**

5656 *crypt()*, *setkey()*, <unistd.h>.

5657 **CHANGE HISTORY**

5658 First released in Issue 1.

5659 Derived from Issue 1 of the SVID.

5660 **Issue 4**

5661 The following changes are incorporated in this issue:

- 5662 • The header <unistd.h> is added to the SYNOPSIS section.
- 5663 • The DESCRIPTION is amended (a) to specify the encoding algorithm as implementation-
- 5664 dependent (b) to change entry to function and (c) to make decoding optional.

5665 • The APPLICATION USAGE section is expanded to explain the restrictions on the availability
5666 of the DES decryption algorithm. |

5667 **Issue 5** |

5668 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

5669 **NAME**

5670 endgrent, getgrent, setgrent — group database entry functions

5671 **SYNOPSIS**

5672 EX #include <grp.h>

5673 void endgrent(void);

5674 struct group *getgrent(void);

5675 void setgrent(void);

5676

5677 **DESCRIPTION**

5678 The *getgrent()* function returns a pointer to a structure containing the broken-out fields of an entry in the group database. When first called, *getgrent()* returns a pointer to a **group** structure containing the first entry in the group database. Thereafter, it returns 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.

5683 The *setgrent()* function effectively rewinds the group database to allow repeated searches.

5684 The *endgrent()* function may be called to close the group database when processing is complete.

5685 These interfaces need not be reentrant.

5686 **RETURN VALUE**

5687 When first called, *getgrent()* will return a pointer to the first group structure in the group database. Upon subsequent calls it returns the next group structure in the group database. The *getgrent()* function returns a null pointer on end-of-file or an error and *errno* may be set to indicate the error.

5691 The return value may point to a static area which is overwritten by a subsequent call to *getgrgid()*, *getgrnam()* or *getgrent()*.

5693 **ERRORS**

5694 The *getgrent()* function may fail if:

5695 [EINTR] A signal was caught during the operation.

5696 [EIO] An I/O error has occurred.

5697 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

5698 [ENFILE] The maximum allowable number of files is currently open in the system.

5699 **EXAMPLES**

5700 None.

5701 **APPLICATION USAGE**

5702 These functions are provided due to their historical usage. Applications should avoid dependencies on fields in the group database, whether the database is a single file, or where in the filesystem namespace the database resides. Applications should use *getgrnam()* and *getgrgid()* whenever possible both because it avoids these dependencies and for greater portability with systems that conform to earlier versions of this specification.

5707 **FUTURE DIRECTIONS**

5708 None.

5709 **SEE ALSO**

5710 *getgrgid()*, *getgrnam()*, *getlogin()*, *getpwent()*, <grp.h>.

5711 **CHANGE HISTORY**

5712 First released in Issue 4, Version 2.

5713 **Issue 5**

5714 Moved from X/OPEN UNIX extension to BASE.

5715 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
5716 VALUE section.

5717 A note indicating that these interfaces need not be reentrant is added to the DESCRIPTION.

5718 **NAME**

5719 endpwent, getpwent, setpwent — user database functions

5720 **SYNOPSIS**

5721 EX #include <pwd.h>

5722 void endpwent(void);

5723 struct passwd *getpwent(void);

5724 void setpwent(void);

5725

5726 **DESCRIPTION**

5727 The *getpwent()* function returns a pointer to a structure containing the broken-out fields of an
 5728 entry in the user database. Each entry in the user database contains a **passwd** structure. When
 5729 first called, *getpwent()* returns a pointer to a **passwd** structure containing the first entry in the
 5730 user database. Thereafter, it returns a pointer to a **passwd** structure containing the next entry in
 5731 the user database. Successive calls can be used to search the entire user database.

5732 If an end-of-file or an error is encountered on reading, *getpwent()* returns a null pointer.

5733 The *setpwent()* function effectively rewinds the user database to allow repeated searches.

5734 The *endpwent()* function may be called to close the user database when processing is complete.

5735 These interfaces need not be reentrant.

5736 **RETURN VALUE**

5737 The *getpwent()* function returns a null pointer on end-of-file or error.

5738 **ERRORS**

5739 The *getpwent()*, *setpwent()* and *endpwent()* functions may fail if:

5740 [EIO] An I/O error has occurred.

5741 In addition, *getpwent()* and *setpwent()* may fail if:

5742 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

5743 [ENFILE] The maximum allowable number of files is currently open in the system.

5744 The return value may point to a static area which is overwritten by a subsequent call to
 5745 *getpwuid()*, *getpwnam()* or *getpwent()*.

5746 **EXAMPLES**

5747 None.

5748 **APPLICATION USAGE**

5749 These functions are provided due to their historical usage. Applications should avoid
 5750 dependencies on fields in the password database, whether the database is a single file, or where
 5751 in the filesystem namespace the database resides. Applications should use *getpwuid()* whenever
 5752 possible both because it avoids these dependencies and for greater portability with systems that
 5753 conform to earlier versions of this specification.

5754 **FUTURE DIRECTIONS**

5755 None.

5756 **SEE ALSO**

5757 *endgrent()*, *getlogin()*, *getpwnam()*, *getpwuid()*, <pwd.h>.

5758 **CHANGE HISTORY**

5759 First released in Issue 4, Version 2.

5760 **Issue 5**

5761 Moved from X/OPEN UNIX extension to BASE.

5762 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
5763 VALUE section.

5764 A note indicating that these interfaces need not be reentrant is added to the DESCRIPTION.

5765 **NAME**

5766 endutxent, getutxent, getutxid, getutxline, pututxline, setutxent — user accounting database
5767 functions

5768 **SYNOPSIS**

```
5769 EX #include <utmpx.h>

5770 void endutxent(void);
5771 struct utmpx *getutxent(void);
5772 struct utmpx *getutxid(const struct utmpx *id);
5773 struct utmpx *getutxline(const struct utmpx *line);
5774 struct utmpx *pututxline(const struct utmpx *utmpx);
5775 void setutxent(void);
5776
```

5777 **DESCRIPTION**

5778 These functions provide access to the user accounting database.

5779 The *getutxent()* function reads in the next entry from the user accounting database. If the
5780 database is not already open, it opens it. If it reaches the end of the database, it fails.

5781 The *getutxid()* function searches forward from the current point in the database. If the **ut_type**
5782 value of the **utmpx** structure pointed to by *id* is **BOOT_TIME**, **OLD_TIME** or **NEW_TIME**, then it
5783 stops when it finds an entry with a matching **ut_type** value. If the **ut_type** value is
5784 **INIT_PROCESS**, **LOGIN_PROCESS**, **USER_PROCESS**, or **DEAD_PROCESS**, then it stops when
5785 it finds an entry whose type is one of these four and whose **ut_id** member matches the **ut_id**
5786 member of the **utmpx** structure pointed to by *id*. If the end of the database is reached without a
5787 match, *getutxid()* fails.

5788 The *getutxid()* or *getutxline()* may cache data. For this reason, to use *getutxline()* to search for
5789 multiple occurrences, it is necessary to zero out the static data after each success, or *getutxline()*
5790 could just return a pointer to the same **utmpx** structure over and over again.

5791 There is one exception to the rule about removing the structure before further reads are done.
5792 The implicit read done by *pututxline()* (if it finds that it is not already at the correct place in the
5793 user accounting database) will not modify the static structure returned by *getutxent()*, *getutxid()*
5794 or *getutxline()*, if the application has just modified this structure and passed the pointer back to
5795 *pututxline()*.

5796 For all entries that match a request, the **ut_type** member indicates the type of the entry. Other
5797 members of the entry will contain meaningful data based on the value of the **ut_type** member as
5798 follows:

5800 ut_type Member	Other Members with Meaningful Data
5801 EMPTY	No others
5802 BOOT_TIME	ut_tv
5803 OLD_TIME	ut_tv
5804 NEW_TIME	ut_tv
5805 USER_PROCESS	ut_id , ut_user (login name of the user), ut_line , ut_pid , ut_tv
5806 INIT_PROCESS	ut_id , ut_pid , ut_tv
5807 LOGIN_PROCESS	ut_id , ut_user (implementation-dependent name of the login 5808 process), ut_pid , ut_tv
5809 DEAD_PROCESS	ut_id , ut_pid , ut_tv

5810 The *getutxline()* function searches forward from the current point in the database until it finds an
5811 entry of the type **LOGIN_PROCESS** or **USER_PROCESS** which also has a **ut_line** value
5812 matching that in the **utmpx** structure pointed to by *line*. If the end of the database is reached

5813 without a match, *getutxline()* fails.

5814 If the process has appropriate privileges, the *pututxline()* function writes out the structure into
 5815 the user accounting database. It uses *getutxid()* to search for a record that satisfies the request.
 5816 If this search succeeds, then the entry is replaced. Otherwise, a new entry is made at the end of
 5817 the user accounting database.

5818 The *setutxent()* function resets the input to the beginning of the database. This should be done
 5819 before each search for a new entry if it is desired that the entire database be examined.

5820 The *endutxent()* function closes the user accounting database.

5821 These interfaces need not be reentrant.

5822 RETURN VALUE

5823 Upon successful completion, *getutxent()*, *getutxid()* and *getutxline()* return a pointer to a **utmpx**
 5824 structure containing a copy of the requested entry in the user accounting database. Otherwise a
 5825 null pointer is returned.

5826 The return value may point to a static area which is overwritten by a subsequent call to
 5827 *getutxid()* or *getutxline()*.

5828 Upon successful completion, *pututxline()* returns a pointer to a **utmpx** structure containing a
 5829 copy of the entry added to the user accounting database. Otherwise a null pointer is returned.

5830 The *endutxent()* and *setutxent()* functions return no value.

5831 ERRORS

5832 No errors are defined for the *endutxent()*, *getutxent()*, *getutxid()*, *getutxline()* and *setutxent()*
 5833 functions.

5834 The *pututxline()* function may fail if:

5835 [EPERM] The process does not have appropriate privileges.

5836 EXAMPLES

5837 None.

5838 APPLICATION USAGE

5839 The sizes of the arrays in the structure can be found using the **sizeof** operator.

5840 FUTURE DIRECTIONS

5841 None.

5842 SEE ALSO

5843 <utmpx.h>.

5844 CHANGE HISTORY

5845 First released in Issue 4, Version 2.

5846 Issue 5

5847 Moved from X/OPEN UNIX extension to BASE.

5848 Normative text previously in the APPLICATION USAGE section is moved to the
 5849 DESCRIPTION.

5850 A note indicating that these interfaces need not be reentrant is added to the DESCRIPTION.

5851	NAME	
5852	environ — array of character pointers to the environment strings	
5853	SYNOPSIS	
5854	extern char **environ;	
5855	DESCRIPTION	
5856	Refer to the XBD specification, Chapter 6, Environment Variables and <i>exec</i> .	
5857	CHANGE HISTORY	
5858	First released in Issue 1.	
5859	Derived from Issue 1 of the SVID.	

5860 **NAME**

5861 erand48 — generate uniformly distributed pseudo-random numbers

5862 **SYNOPSIS**

5863 EX #include <stdlib.h>

5864 double erand48(unsigned short int xsubi[3]);

5865

5866 **DESCRIPTION**5867 Refer to *drand48()*.5868 **CHANGE HISTORY**

5869 First released in Issue 1.

5870 Derived from Issue 1 of the SVID.

5871 **Issue 4**

5872 The following change is incorporated in this issue:

- 5873
- The <stdlib.h> header is added to the SYNOPSIS section.

5874 **NAME**5875 *erf*, *erfc* — error and complementary error functions5876 **SYNOPSIS**5877 EX `#include <math.h>`5878 `double erf(double x);`5879 `double erfc(double x);`

5880

5881 **DESCRIPTION**5882 The *erf()* function computes the error function of *x*, defined as:

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

5884 The *erfc()* function computes $1.0 - \text{erf}(x)$.

5885 An application wishing to check for error situations should set *errno* to 0 before calling *erf()*. If
 5886 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

5887 **RETURN VALUE**5888 Upon successful completion, *erf()* and *erfc()* return the value of the error function and
 5889 complementary error function, respectively.5890 If *x* is NaN, NaN is returned and *errno* may be set to [EDOM].5891 If the correct value would cause underflow, 0 is returned and *errno* may be set to [ERANGE].5892 **ERRORS**5893 The *erf()* and *erfc()* functions may fail if:5894 [EDOM] The value of *x* is NaN.

5895 [ERANGE] The result underflows.

5896 No other errors will occur.

5897 **EXAMPLES**

5898 None.

5899 **APPLICATION USAGE**

5900 The *erfc()* function is provided because of the extreme loss of relative accuracy if *erf(x)* is called
 5901 for large *x* and the result subtracted from 1.0.

5902 **FUTURE DIRECTIONS**

5903 None.

5904 **SEE ALSO**5905 *isnan()*, `<math.h>`.5906 **CHANGE HISTORY**

5907 First released in Issue 1.

5908 Derived from Issue 1 of the SVID.

5909 **Issue 4**

5910 The following changes are incorporated in this issue:

- 5911 • Removed references to *matherr()*.
- 5912 • The RETURN VALUE and ERRORS sections are substantially rewritten to rationalise error
- 5913 handling in the mathematics functions.

5914 **Issue 5**

5915 The DESCRIPTION is updated to indicate how an application should check for an error. This

5916 text was previously published in the APPLICATION USAGE section.

5917 **NAME**

5918 errno — XSI error return value

5919 **SYNOPSIS**

5920 #include <errno.h>

5921 **DESCRIPTION**5922 *errno* is used by many XSI functions to return error values.

5923 Many functions provide an error number in *errno* which has type **int** and is defined in <**errno.h**>. The value of *errno* will be defined only after a call to a function for which it is explicitly stated to be set and until it is changed by the next function call. The value of *errno* should only be examined when it is indicated to be valid by a function's return value. Programs should obtain the definition of *errno* by the inclusion of <**errno.h**>. The practice of defining *errno* in a program as **extern int errno** is obsolescent. No function in this specification sets *errno* to 0 to indicate an error.

5930 It is unspecified whether *errno* is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual object, or a program defines an identifier with the name **errno**, the behaviour is undefined.

5933 The symbolic values stored in *errno* are documented in the ERRORS sections on all relevant pages.

5935 **RETURN VALUE**

5936 None.

5937 **ERRORS**

5938 None.

5939 **EXAMPLES**

5940 None.

5941 **APPLICATION USAGE**

5942 Previously both POSIX and X/Open documents were more restrictive than the ISO C standard in that they required *errno* to be defined as an external variable, whereas the ISO C standard required only that *errno* be defined as a modifiable **lvalue** with type **int**.

5945 A program that uses *errno* for error checking should set it to 0 before a function call, then inspect it before a subsequent function call.

5947 **FUTURE DIRECTIONS**

5948 None.

5949 **SEE ALSO**5950 <**errno.h**>, Section 2.3 on page 22.5951 **CHANGE HISTORY**

5952 First released in Issue 1.

5953 Derived from Issue 1 of the SVID.

5954 **Issue 4**

5955 The following changes are incorporated for alignment with the ISO C standard:

- 5956 • The DESCRIPTION now guarantees that *errno* is set to 0 at program startup, and that it is never reset to 0 by any XSI function.
- 5958 • The APPLICATION USAGE section is added. This revision is aligned with the ISO C standard, which permits *errno* to be a macro.

5960 Another change is incorporated as follows:

- 5961 • The FUTURE DIRECTIONS section is deleted.

5962 **Issue 5**

5963 The following sentence is deleted from the DESCRIPTION: "The value of *errno* is 0 at program
5964 startup, but is never set to 0 by any XSI function". The DESCRIPTION also no longer states that
5965 conforming implementations may support the declaration:

5966 extern int errno;

5967 Both these historical behaviours are obsolete and may not be supported by some
5968 implementations.

5969 **NAME**

5970 environ, execl, execv, execl, execve, execlp, execvp — execute a file

5971 **SYNOPSIS**

5972 #include <unistd.h>

5973 extern char **environ;

5974 int execl(const char *path, const char *arg0, ... /*, (char *)0 */);

5975 int execv(const char *path, char *const argv[]);

5976 int execl(const char *path,

5977 const char *arg0, ... /*, (char *)0, char *const envp[]*/);

5978 int execve(const char *path, char *const argv[], char *const envp[]);

5979 int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);

5980 int execvp(const char *file, char *const argv[]);

5981 **DESCRIPTION**

5982 The *exec* functions replace the current process image with a new process image. The new image
 5983 is constructed from a regular, executable file called the *new process image file*. There is no return
 5984 from a successful *exec*, because the calling process image is overlaid by the new process image.

5985 When a C-language program is executed as a result of this call, it is entered as a C-language
 5986 function call as follows:

5987 int main (int argc, char *argv[]);

5988 where *argc* is the argument count and *argv* is an array of character pointers to the arguments
 5989 themselves. In addition, the following variable:

5990 extern char **environ;

5991 is initialised as a pointer to an array of character pointers to the environment strings. The *argv*
 5992 and *environ* arrays are each terminated by a null pointer. The null pointer terminating the *argv*
 5993 array is not counted in *argc*.

5994 Conforming multi-threaded applications will not use the *environ* variable to access or modify
 5995 any environment variable while any other thread is concurrently modifying any environment
 5996 variable. A call to any function dependent on any environment variable is considered a use of
 5997 the *environ* variable to access that environment variable.

5998 The arguments specified by a program with one of the *exec* functions are passed on to the new
 5999 process image in the corresponding *main()* arguments.

6000 The argument *path* points to a pathname that identifies the new process image file.

6001 The argument *file* is used to construct a pathname that identifies the new process image file. If
 6002 the *file* argument contains a slash character, the *file* argument is used as the pathname for this
 6003 file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as
 6004 the environment variable *PATH* (see **XBD specification, Chapter 6, Environment Variables**). If
 6005 this environment variable is not present, the results of the search are implementation-dependent.

6006 **EX** If the process image file is not a valid executable object, *execlp()* and *execvp()* use the contents of
 6007 that file as standard input to a command interpreter conforming to *system()*. In this case, the
 6008 command interpreter becomes the new process image.

6009 The arguments represented by *arg0, ...* are pointers to null-terminated character strings. These
 6010 strings constitute the argument list available to the new process image. The list is terminated by
 6011 a null pointer. The argument *arg0* should point to a filename that is associated with the process
 6012 being started by one of the *exec* functions.

6013		The argument <i>argv</i> is an array of character pointers to null-terminated strings. The last member of this array must be a null pointer. These strings constitute the argument list available to the new process image. The value in <i>argv</i> [0] should point to a filename that is associated with the process being started by one of the <i>exec</i> functions.	
6014			
6015			
6016			
6017		The argument <i>envp</i> is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The <i>envp</i> array is terminated by a null pointer.	
6018			
6019			
6020		For those forms not containing an <i>envp</i> pointer (<i>execl</i> (), <i>execv</i> (), <i>execlp</i> () and <i>execvp</i> ()), the environment for the new process image is taken from the external variable <i>environ</i> in the calling process.	
6021			
6022			
6023		The number of bytes available for the new process' combined argument and environment lists is {ARG_MAX}. It is implementation-dependent whether null terminators, pointers, and/or any alignment bytes are included in this total.	
6024			
6025			
6026		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, including file locks remain unchanged.	
6027			
6028			
6029		Directory streams open in the calling process image are closed in the new process image.	
6030	EX	The state of conversion descriptors and message catalogue descriptors in the new process image is undefined. For the new process, the equivalent of:	
6031			
6032		<code>setlocale(LC_ALL, "C")</code>	
6033		is executed at startup.	
6034		Signals set to the default action (SIG_DFL) in the calling process image are set to the default action in the new process image. Signals set to be ignored (SIG_IGN) by the calling process image are set to be ignored by the new process image. Signals set to be caught by the calling process image are set to the default action in the new process image (see <signal.h>). After a successful call to any of the <i>exec</i> functions, alternate signal stacks are not preserved and the SA_ONSTACK flag is cleared for all signals.	
6035			
6036			
6037	EX		
6038			
6039			
6040		After a successful call to any of the <i>exec</i> functions, any functions previously registered by <i>atexit</i> () are no longer registered.	
6041			
6042	EX	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 are saved (as the saved set-user-ID and the saved set-group-ID for use by <i>setuid</i> ()).	
6043			
6044			
6045			
6046			
6047			
6048			
6049	FIPS		
6050			
6051			
6052	EX	Any shared memory segments attached to the calling process image will not be attached to the new process image.	
6053			
6054	EX	Any mappings established through <i>mmap</i> () are not preserved across an <i>exec</i> .	
6055	RT	If _XOPEN_REALTIME is defined and has a value other than -1, any named semaphores open in the calling process are closed as if by appropriate calls to <i>sem_close</i> ()).	
6056			

6057		If the Process Memory Locking option is supported, memory locks established by the calling
6058		process via calls to <i>mlockall()</i> or <i>mlock()</i> are removed. If locked pages in the address space of the
6059		calling process are also mapped into the address spaces of other processes and are locked by
6060		those processes, the locks established by the other processes will be unaffected by the call by this
6061		process to the <i>exec</i> function. If the <i>exec</i> function fails, the effect on memory locks is unspecified.
6062		Memory mappings created in the process are unmapped before the address space is rebuilt for
6063		the new process image.
6064	RT	If the Process Scheduling option is supported, for the SCHED_FIFO and SCHED_RR scheduling
6065		policies, the policy and priority settings are not changed by a call to an <i>exec</i> function. For other
6066		scheduling policies, the policy and priority settings on <i>exec</i> are implementation-dependent.
6067		If the Timers option is supported, per-process timers created by the calling process are deleted
6068		before replacing the current process image with the new process image.
6069		If the Message Passing option is supported, all open message queue descriptors in the calling
6070		process are closed, as described in <i>mq_close()</i> .
6071		If the Asynchronous Input and Output option is supported, any outstanding asynchronous I/O
6072		operations may be canceled. Those asynchronous I/O operations that are not canceled will
6073		complete as if the <i>exec</i> function had not yet occurred, but any associated signal notifications are
6074		suppressed. It is unspecified whether the <i>exec</i> function itself blocks awaiting such I/O
6075		completion. In no event, however, will the new process image created by the <i>exec</i> function be
6076		affected by the presence of outstanding asynchronous I/O operations at the time the <i>exec</i>
6077		function is called. Whether any I/O is cancelled, and which I/O may be cancelled upon <i>exec</i> , is
6078		implementation-dependent.
6079		The new process also inherits at least the following attributes from the calling process image:
6080	EX	nice value (see <i>nice()</i>)
6081	EX	<i>semadj</i> values (see <i>semop()</i>)
6082		process ID
6083		parent process ID
6084		process group ID
6085		session membership
6086		real user ID
6087		real group ID
6088		supplementary group IDs
6089		time left until an alarm clock signal (see <i>alarm()</i>)
6090		current working directory
6091		root directory
6092		file mode creation mask (see <i>umask()</i>)
6093	EX	file size limit (see <i>ulimit()</i>)
6094		process signal mask (see <i>sigprocmask()</i>)
6095		pending signal (see <i>sigpending()</i>)
6096		<i>tms_utime</i> , <i>tms_stime</i> , <i>tms_cutime</i> , and <i>tms_cstime</i> (see <i>times()</i>)
6097	EX	resource limits
6098	EX	controlling terminal
6099	EX	interval timers
6100		All other process attributes defined in this document will be the same in the new and old process
6101		images. The inheritance of process attributes not defined by this specification is
6102		implementation-dependent.
6103		A call to any <i>exec</i> function from a process with more than one thread results in all threads being
6104		terminated and the new executable image being loaded and executed. No destructor functions

6105		will be called.	
6106		Upon successful completion, the <i>exec</i> functions mark for update the <i>st_atime</i> field of the file. If an	
6107		<i>exec</i> function failed but was able to locate the <i>process image file</i> , whether the <i>st_atime</i> field is	
6108		marked for update is unspecified. Should the <i>exec</i> function succeed, the process image file is	
6109		considered to have been opened with <i>open()</i> . The corresponding <i>close()</i> is considered to occur at	
6110		a time after this open, but before process termination or successful completion of a subsequent	
6111		call to one of the <i>exec</i> functions. The <i>argv[]</i> and <i>envp[]</i> arrays of pointers and the strings to	
6112		which those arrays point will not be modified by a call to one of the <i>exec</i> functions, except as a	
6113		consequence of replacing the process image.	
6114	EX	The saved resource limits in the new process image are set to be a copy of the process's	
6115		corresponding hard and soft limits.	
6116		RETURN VALUE	
6117		If one of the <i>exec</i> functions returns to the calling process image, an error has occurred; the return	
6118		value is <i>-1</i> , and <i>errno</i> is set to indicate the error.	
6119		ERRORS	
6120		The <i>exec</i> functions will fail if:	
6121		[E2BIG] The number of bytes used by the new process image's argument list and	
6122		environment list is greater than the system-imposed limit of {ARG_MAX}	
6123		bytes.	
6124		[EACCES] Search permission is denied for a directory listed in the new process image	
6125		file's path prefix, or the new process image file denies execution permission,	
6126		or the new process image file is not a regular file and the implementation does	
6127		not support execution of files of its type.	
6128	EX	[ELOOP] Too many symbolic links were encountered in resolving <i>path</i> .	
6129	FIPS	[ENAMETOOLONG]	
6130		The length of the <i>path</i> or <i>file</i> arguments, or an element of the environment	
6131		variable <i>PATH</i> prefixed to a file, exceeds {PATH_MAX}, or a pathname	
6132		component is longer than {NAME_MAX}.	
6133		[ENOENT] A component of <i>path</i> or <i>file</i> does not name an existing file or <i>path</i> or <i>file</i> is an	
6134		empty string.	
6135		[ENOTDIR] A component of the new process image file's path prefix is not a directory.	
6136		The <i>exec</i> functions, except for <i>execlp()</i> and <i>execvp()</i> , will fail if:	
6137		[ENOEXEC] The new process image file has the appropriate access permission but is not in	
6138		the proper format.	
6139		The <i>exec</i> functions may fail if:	
6140	EX	[ENAMETOOLONG]	
6141		Pathname resolution of a symbolic link produced an intermediate result	
6142		whose length exceeds {PATH_MAX}.	
6143		[ENOMEM] The new process image requires more memory than is allowed by the	
6144		hardware or system-imposed memory management constraints.	
6145	EX	[ETXTBSY] The new process image file is a pure procedure (shared text) file that is	
6146		currently open for writing by some process.	

6147 **EXAMPLES**

6148 None.

6149 **APPLICATION USAGE**

6150 As the state of conversion descriptors and message catalogue descriptors in the new process
 6151 image is undefined, portable applications should not rely on their use and should close them
 6152 prior to calling one of the *exec* functions.

6153 Applications that require other than the default POSIX locale should call *setlocale()* with the
 6154 appropriate parameters to establish the locale of the new process.

6155 The *environ* array should not be accessed directly by the application.

6156 **FUTURE DIRECTIONS**

6157 None.

6158 **SEE ALSO**

6159 *alarm()*, *atexit()*, *chmod()*, *exit()*, *fcntl()*, *fork()*, *fstatvfs()*, *getenv()*, *getitimer()*, *getrlimit()*, *mmap()*,
 6160 *nice()*, *putenv()*, *semop()*, *setlocale()*, *shmat()*, *sigaction()*, *sigaltstack()*, *sigpending()*, *sigprocmask()*,
 6161 *system()*, *times()*, *ulimit()*, *umask()*, **<unistd.h>**, XBD specification, **Chapter 9, General Terminal**
 6162 **Interface**.

6163 **CHANGE HISTORY**

6164 First released in Issue 1.

6165 Derived from Issue 1 of the SVID.

6166 **Issue 4**

6167 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 6168 • In the ERRORS section, (a) the description of the [ENOEXEC] error is changed to indicate
 6169 that this error does not apply to *execlp()* and *execvp()*, and (b) the [ENOMEM] error is added.

6170 The following change is incorporated for alignment with the FIPS requirements:

- 6171 • In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a
 6172 pathname component is larger than {NAME_MAX} is now defined as mandatory and marked
 6173 as an extension.

6174 Other changes are incorporated as follows:

- 6175 • The header **<unistd.h>** is added to the SYNOPSIS section.
- 6176 • The **const** keyword is added to identifiers of constant type (for example, *path*, *file*).
- 6177 • In the DESCRIPTION, (a) an indication of the disposition of conversion descriptors after a
 6178 call to one of the *exec* functions is added, (b) a statement about the interaction between *exec*
 6179 and *atexit()* is added, (c) *usually* in the descriptions of argument pointers is removed, (d)
 6180 *owner ID* is changed to *user ID*, (e) shared memory is no longer optional and (f) the
 6181 penultimate paragraph is changed to correct an error in Issue 3: it now refers to “All other
 6182 process attributes...” instead of “All the process attributes...”
- 6183 • A note about the initialisation of locales is added to the APPLICATION USAGE section.

6184 **Issue 4, Version 2**

6185 The following changes are incorporated for X/OPEN UNIX conformance:

- 6186 • The DESCRIPTION is changed to indicate the disposition of alternate signal stacks, the
 6187 SA_ONSTACK flag and mappings established through *mmap()* after a successful call to one
 6188 of the *exec* functions. The effects of ST_NOSUID being set for a file system are defined. A
 6189 statement is added that mappings established through *mmap()* are not preserved across an

- 6190 *exec*. The list of inherited process attributes is extended to include resource limits, the
6191 controlling terminal and interval timers.
- 6192 • In the ERRORS section, the condition whereby [ELOOP] will be returned if too many
6193 symbolic links are encountered during pathname resolution is defined as mandatory.
- 6194 • In the ERRORS section, a second [ENAMETOOLONG] condition is defined that may report
6195 excessive length of an intermediate result of pathname resolution of a symbolic link.
- 6196 **Issue 5**
- 6197 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
6198 Threads Extension.
- 6199 Large File Summit extensions added.

6200 **NAME**6201 `exit, _exit` — terminate a process6202 **SYNOPSIS**6203 `#include <stdlib.h>`6204 `void exit(int status);`6205 `#include <unistd.h>`6206 `void _exit(int status);`6207 **DESCRIPTION**6208 The `exit()` function first calls all functions registered by `atexit()`, in the reverse order of their
6209 registration. Each function is called as many times as it was registered.6210 If a function registered by a call to `atexit()` fails to return, the remaining registered functions are
6211 not called and the rest of the `exit()` processing is not completed. If `exit()` is called more than once,
6212 the effects are undefined.6213 The `exit()` function then flushes all output streams, closes all open streams, and removes all files
6214 created by `tmpfile()`. Finally, control is returned to the host environment as described below.
6215 The values of `status` can be `EXIT_SUCCESS` or `EXIT_FAILURE`, as described in `<stdlib.h>`, or any
6216 implementation-dependent value, although note that only the range 0 through 255 will be
6217 available to a waiting parent process.6218 The `_exit()` and `exit()` functions terminate the calling process with the following consequences:

- 6219 EX • All of the file descriptors, directory streams, conversion descriptors and message catalogue
6220 descriptors open in the calling process are closed.
- 6221 EX • If the parent process of the calling process is executing a `wait()`, `wait3()`, `waitid()` or `waitpid()`,
6222 and has neither set its `SA_NOCLDWAIT` flag nor set `SIGCHLD` to `SIG_IGN`, it is notified of
6223 the calling process' termination and the low-order eight bits (that is, bits 0377) of `status` are
6224 made available to it. If the parent is not waiting, the child's status will be made available to it
6225 EX when the parent subsequently executes `wait()`, `wait3()`, `waitid()` or `waitpid()`.
- 6226 EX • If the parent process of the calling process is not executing a `wait()`, `wait3()`, `waitid()` or
6227 EX `waitpid()`, and has not set its `SA_NOCLDWAIT` flag, or set `SIGCHLD` to `SIG_IGN`, the calling
6228 process is transformed into a *zombie process*. A *zombie process* is an inactive process and it will
6229 EX be deleted at some later time when its parent process executes `wait()`, `wait3()`, `waitid()` or
6230 `waitpid()`.
- 6231 • Termination of a process does not directly terminate its children. The sending of a `SIGHUP`
6232 signal as described below indirectly terminates children in some circumstances.
- 6233 • If the implementation supports the `SIGCHLD` signal, a `SIGCHLD` will be sent to the parent
6234 process.
- 6235 EX • If the parent process has set its `SA_NOCLDWAIT` flag, or set `SIGCHLD` to `SIG_IGN`, the
6236 status will be discarded, and the lifetime of the calling process will end immediately. If
6237 `SA_NOCLDWAIT` is set, it is implementation-dependent whether a `SIGCHLD` signal will be
6238 sent to the parent process.
- 6239 • The parent process ID of all of the calling process' existing child processes and zombie
6240 processes is set to the process ID of an implementation-dependent system process. That is,
6241 these processes are inherited by a special system process.
- 6242 EX • Each attached shared-memory segment is detached and the value of `shm_nattch` (see
6243 `shmget()`) in the data structure associated with its shared memory ID is decremented by 1.

6244	EX	<ul style="list-style-type: none"> For each semaphore for which the calling process has set a <i>semadj</i> value, see <i>semop()</i>, that value is added to the <i>semval</i> of the specified semaphore.
6245		
6246		<ul style="list-style-type: none"> If the process is a controlling process, the SIGHUP signal will be sent to each process in the foreground process group of the controlling terminal belonging to the calling process.
6247		
6248		<ul style="list-style-type: none"> 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.
6249		
6250		<ul style="list-style-type: none"> 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 will be sent to each process in the newly-orphaned process group.
6251		
6252		
6253	RT	<ul style="list-style-type: none"> If the Semaphores option is supported, all open named semaphores in the calling process are closed as if by appropriate calls to <i>sem_close()</i>.
6254		
6255		<ul style="list-style-type: none"> If the Process Memory Locking option is supported, any memory locks established by the process via calls to <i>mlockall()</i> or <i>mlock()</i> are 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 will be unaffected by the call by this process to <i>_exit()</i>.
6256		
6257		
6258		
6259		
6260		<ul style="list-style-type: none"> Memory mappings created in the process are unmapped before the process is destroyed.
6261	RT	<ul style="list-style-type: none"> If the Message Passing option is supported, all open message queue descriptors in the calling process are closed as if by appropriate calls to <i>mq_close()</i>.
6262		
6263		<ul style="list-style-type: none"> If the Asynchronous Input and Output option is supported any outstanding cancelable asynchronous I/O operations may be canceled. Those asynchronous I/O operations that are not canceled will complete as if the <i>_exit()</i> operation had not yet occurred, but any associated signal notifications will be suppressed. The <i>_exit()</i> operation itself may block awaiting such I/O completion. Whether any I/O is cancelled, and which I/O may be cancelled upon <i>_exit()</i>, is implementation-dependent.
6264		
6265		
6266		
6267		
6268		
6269		<ul style="list-style-type: none"> Threads terminated by a call to <i>_exit()</i> will not invoke their cancellation cleanup handlers or per-thread data destructors.
6270		
6271	RETURN VALUE	
6272	These functions do not return.	
6273	ERRORS	
6274	No errors are defined.	
6275	EXAMPLES	
6276	None.	
6277	APPLICATION USAGE	
6278	Normally applications should use <i>exit()</i> rather than <i>_exit()</i> .	
6279	FUTURE DIRECTIONS	
6280	None.	
6281	SEE ALSO	
6282	<i>atexit()</i> , <i>close()</i> , <i>fclose()</i> , <i>semop()</i> , <i>shmget()</i> , <i>sigaction()</i> , <i>wait()</i> , <i>wait3()</i> , <i>waitid()</i> , <i>waitpid()</i> ,	
6283	<stdlib.h>, <unistd.h>.	
6284	CHANGE HISTORY	
6285	First released in Issue 1.	
6286	Derived from Issue 1 of the SVID.	

6287 Issue 4

6288 The following change is incorporated for alignment with the ISO C standard:

- 6289 • In the DESCRIPTION, (a) interactions between *exit()* and *atexit()* are defined, and (b) it is
- 6290 now stated explicitly that all files created by *tmpfile()* are removed.

6291 Other changes are incorporated as follows:

- 6292 • The header **<unistd.h>** is added to the SYNOPSIS for *_exit()*.
- 6293 • In the DESCRIPTION, text describing (a) the behaviour when a function registered by
- 6294 *atexit()* fails to return, and (b) consequences of calling *exit()* more than once, are added.
- 6295 • The phrase “If the implementation supports job control” is removed from the last bullet in
- 6296 the DESCRIPTION. This is because job control is now defined as mandatory for all
- 6297 conforming implementations.

6298 Issue 4, Version 2

6299 The following changes to the DESCRIPTION are incorporated for X/OPEN UNIX conformance:

- 6300 • References to the functions *wait3()* and *waitid()* are added in appropriate places throughout
- 6301 the text.
- 6302 • Interactions with the SA_NOCLDWAIT flag and SIGCHLD signal are defined.
- 6303 • It is specified that each mapped memory object is unmapped.

6304 Issue 5

6305 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX

6306 Threads Extension.

6307 Interactions with the SA_NOCLDWAIT flag and SIGCHLD signal are further clarified.

6308 The values of *status* from *exit()* are better described.

6309 **NAME**

6310 exp — exponential function

6311 **SYNOPSIS**

6312 #include <math.h>

6313 double exp(double x);

6314 **DESCRIPTION**6315 The *exp()* function computes the exponent of *x*, defined as e^x .

6316 An application wishing to check for error situations should set *errno* to 0 before calling *exp()*. If
 6317 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

6318 **RETURN VALUE**6319 Upon successful completion, *exp()* returns the exponential value of *x*.

6320 If the correct value would cause overflow, *exp()* returns HUGE_VAL and sets *errno* to
 6321 [ERANGE].

6322 If the correct value would cause underflow, *exp()* returns 0 and may set *errno* to [ERANGE].

6323 EX If *x* is NaN, NaN is returned and *errno* may be set to [EDOM].

6324 **ERRORS**6325 The *exp()* function will fail if:

6326 [ERANGE] The result overflows.

6327 The *exp()* function may fail if:

6328 EX [EDOM] The value of *x* is NaN.

6329 [ERANGE] The result underflows.

6330 EX No other errors will occur.

6331 **EXAMPLES**

6332 None.

6333 **APPLICATION USAGE**

6334 None.

6335 **FUTURE DIRECTIONS**

6336 None.

6337 **SEE ALSO**

6338 isnan(), log(), <math.h>.

6339 **CHANGE HISTORY**

6340 First released in Issue 1.

6341 Derived from Issue 1 of the SVID.

6342 **Issue 4**

6343 The following changes are incorporated in this issue:

- 6344 • Removed references to *matherr()*.
- 6345 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
 6346 the ISO C standard and to rationalise error handling in the mathematics functions.
- 6347 • The return value specified for [EDOM] is marked as an extension.

6348 Issue 5

6349 The DESCRIPTION is updated to indicate how an application should check for an error. This
6350 text was previously published in the APPLICATION USAGE section.

6351 **NAME**

6352 expm1 — compute exponential functions

6353 **SYNOPSIS**6354 EX `#include <math.h>`6355 `double expm1 (double x);`

6356

6357 **DESCRIPTION**6358 The *expm1()* function computes $e^x - 1.0$.6359 **RETURN VALUE**6360 If *x* is NaN, then the function returns NaN and *errno* may be set to EDOM.6361 If *x* is positive infinity, *expm1()* returns positive infinity.6362 If *x* is negative infinity, *expm1()* returns -1.0 .6363 If the value overflows, *expm1()* returns HUGE_VAL and may set *errno* to ERANGE.6364 **ERRORS**6365 The *expm1()* function may fail if:6366 [EDOM] The value of *x* is NaN.

6367 [ERANGE] The result overflows.

6368 **EXAMPLES**

6369 None.

6370 **APPLICATION USAGE**6371 The value of *expm1(x)* may be more accurate than *exp(x)*−1.0 for small values of *x*.6372 The *expm1()* and *log1p()* functions are useful for financial calculations of $((1+x)^n - 1)/x$, namely:6373 $\text{expm1}(n * \log1p(x)) / x$ 6374 when *x* is very small (for example, when calculating small daily interest rates). These functions
6375 also simplify writing accurate inverse hyperbolic functions.6376 **FUTURE DIRECTIONS**

6377 None.

6378 **SEE ALSO**6379 *exp()*, *ilogb()*, *log1p()*, *<math.h>*.6380 **CHANGE HISTORY**

6381 First released in Issue 4, Version 2.

6382 **Issue 5**

6383 Moved from X/OPEN UNIX extension to BASE.

6384 **NAME**

6385 fabs — absolute value function

6386 **SYNOPSIS**

6387 #include <math.h>

6388 double fabs(double x);

6389 **DESCRIPTION**6390 The *fabs()* function computes the absolute value of *x*, $|x|$.

6391 An application wishing to check for error situations should set *errno* to 0 before calling *fabs()*. If
 6392 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

6393 **RETURN VALUE**6394 Upon successful completion, *fabs()* returns the absolute value of *x*.6395 EX If *x* is NaN, NaN is returned and *errno* may be set to [EDOM].6396 If the result underflows, 0 is returned and *errno* may be set to [ERANGE].6397 **ERRORS**6398 The *fabs()* function may fail if:6399 EX [EDOM] The value of *x* is NaN.

6400 [ERANGE] The result underflows.

6401 EX No other errors will occur.

6402 **EXAMPLES**

6403 None.

6404 **APPLICATION USAGE**

6405 None.

6406 **FUTURE DIRECTIONS**

6407 None.

6408 **SEE ALSO**6409 *isnan()*, <math.h>.6410 **CHANGE HISTORY**

6411 First released in Issue 1.

6412 Derived from Issue 1 of the SVID.

6413 **Issue 4**

6414 The following changes are incorporated in this issue:

- 6415 • Removed references to *matherr()*.
- 6416 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
- 6417 the ISO C standard and to rationalise error handling in the mathematics functions.
- 6418 • The return value specified for [EDOM] is marked as an extension.

6419 **Issue 5**

6420 The DESCRIPTION is updated to indicate how an application should check for an error. This
 6421 text was previously published in the APPLICATION USAGE section.

6422 **NAME**

6423 fattach — attach a STREAMS-based file descriptor to a file in the file system name space

6424 **SYNOPSIS**

6425 EX #include <stropts.h>

6426 int fattach(int *fildev*, const char **path*);

6427

6428 **DESCRIPTION**

6429 The *fattach()* function attaches a STREAMS-based file descriptor to a file, effectively associating
 6430 a pathname with *fildev*. The *fildev* argument must be a valid open file descriptor associated with
 6431 a STREAMS file. The *path* argument points to a pathname of an existing file. The process must
 6432 have appropriate privileges, or must be the owner of the file named by *path* and have write
 6433 permission. A successful call to *fattach()* causes all pathnames that name the file named by *path*
 6434 to name the STREAMS file associated with *fildev*, until the STREAMS file is detached from the
 6435 file. A STREAMS file can be attached to more than one file and can have several pathnames
 6436 associated with it.

6437 The attributes of the named STREAMS file are initialised as follows: the permissions, user ID,
 6438 group ID, and times are set to those of the file named by *path*, the number of links is set to 1, and
 6439 the size and device identifier are set to those of the STREAMS file associated with *fildev*. If any
 6440 attributes of the named STREAMS file are subsequently changed (for example, by *chmod()*),
 6441 neither the attributes of the underlying file nor the attributes of the STREAMS file to which *fildev*
 6442 refers are affected.

6443 File descriptors referring to the underlying file, opened prior to an *fattach()* call, continue to refer
 6444 to the underlying file.

6445 **RETURN VALUE**

6446 Upon successful completion, *fattach()* returns 0. Otherwise, -1 is returned and *errno* is set to
 6447 indicate the error.

6448 **ERRORS**6449 The *fattach()* function will fail if:

6450 [EACCES] Search permission is denied for a component of the path prefix, or the process
 6451 is the owner of *path* but does not have write permissions on the file named by
 6452 *path*.

6453 [EBADF] The *fildev* argument is not a valid open file descriptor.

6454 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

6455 [ENOTDIR] A component of the path prefix is not a directory.

6456 [EPERM] The effective user ID of the process is not the owner of the file named by *path*
 6457 and the process does not have appropriate privilege.

6458 [EBUSY] The file named by *path* is currently a mount point or has a STREAMS file
 6459 attached to it.

6460 [ENAMETOOLONG]

6461 The size of *path* exceeds {PATH_MAX}, or a component of *path* is longer than
 6462 {NAME_MAX}.

6463 [ELOOP] Too many symbolic links were encountered in resolving *path*.

6464 The *fattach()* function may fail if:

6465 [EINVAL] The *filides* argument does not refer to a STREAMS file.

6466 [ENAMETOOLONG]

6467 Pathname resolution of a symbolic link produced an intermediate result
 6468 whose length exceeds {PATH_MAX}.

6469 [EXDEV] A link to a file on another file system was attempted.

6470 **EXAMPLES**

6471 None.

6472 **APPLICATION USAGE**

6473 The *fattach()* function behaves similarly to the traditional *mount()* function in the way a file is
 6474 temporarily replaced by the root directory of the mounted file system. In the case of *fattach()*,
 6475 the replaced file need not be a directory and the replacing file is a STREAMS file.

6476 **FUTURE DIRECTIONS**

6477 None.

6478 **SEE ALSO**

6479 *fdetach()*, *isastream()*, <stropts.h>.

6480 **CHANGE HISTORY**

6481 First released in Issue 4, Version 2.

6482 **Issue 5**

6483 Moved from X/OPEN UNIX extension to BASE. The [EXDEV] error is added to the list of
 6484 optional errors in the ERRORS section.

6485 **NAME**

6486 fchdir — change working directory

6487 **SYNOPSIS**

6488 EX #include <unistd.h>

6489 int fchdir(int *fildev*);

6490

6491 **DESCRIPTION**6492 The *fchdir()* function has the same effect as *chdir()* except that the directory that is to be the new
6493 current working directory is specified by the file descriptor *fildev*.6494 **RETURN VALUE**6495 Upon successful completion, *fchdir()* returns 0. Otherwise, it returns -1 and sets *errno* to
6496 indicate the error. On failure the current working directory remains unchanged.6497 **ERRORS**6498 The *fchdir()* function will fail if:6499 [EACCES] Search permission is denied for the directory referenced by *fildev*.6500 [EBADF] The *fildev* argument is not an open file descriptor.6501 [ENOTDIR] The open file descriptor *fildev* does not refer to a directory.6502 The *fchdir()* may fail if:6503 [EINTR] A signal was caught during the execution of *fchdir()*.

6504 [EIO] An I/O error occurred while reading from or writing to the file system.

6505 **EXAMPLES**

6506 None.

6507 **APPLICATION USAGE**

6508 None.

6509 **FUTURE DIRECTIONS**

6510 None.

6511 **SEE ALSO**6512 *chdir()*, <unistd.h>.6513 **CHANGE HISTORY**

6514 First released in Issue 4, Version 2.

6515 **Issue 5**

6516 Moved from X/OPEN UNIX extension to BASE.

6517 **NAME**

6518 fchmod — change mode of a file

6519 **SYNOPSIS**

6520 #include <sys/stat.h>

6521 int fchmod(int *fil*des, mode_t *mode*);6522 **DESCRIPTION**6523 The *fchmod()* function has the same effect as *chmod()* except that the file whose permissions are
6524 to be changed is specified by the file descriptor *fil*des.6525 RT If the Shared Memory Objects option is supported, and *fil*des references a shared memory object,
6526 the *fchmod()* function need only affect the S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH,
6527 and S_IWOTH file permission bits.6528 **RETURN VALUE**6529 Upon successful completion, *fchmod()* returns 0. Otherwise, it returns -1 and sets *errno* to
6530 indicate the error.6531 **ERRORS**6532 The *fchmod()* function will fail if:6533 [EBADF] The *fil*des argument is not an open file descriptor.6534 [EPERM] The effective user ID does not match the owner of the file and the process
6535 does not have appropriate privilege.6536 [EROFS] The file referred to by *fil*des resides on a read-only file system.6537 The *fchmod()* function may fail if:6538 [EINTR] The *fchmod()* function was interrupted by a signal.6539 [EINVAL] The value of the *mode* argument is invalid.6540 [EINVAL] The *fil*des argument refers to a pipe and the implementation disallows
6541 execution of *fchmod()* on a pipe.6542 **EXAMPLES**

6543 None.

6544 **APPLICATION USAGE**

6545 None.

6546 **FUTURE DIRECTIONS**

6547 None.

6548 **SEE ALSO**6549 *chmod()*, *chown()*, *creat()*, *fcntl()*, *fstatvfs()*, *mknod()*, *open()*, *read()*, *stat()*, *write()*, <sys/stat.h>.6550 **CHANGE HISTORY**

6551 First released in Issue 4, Version 2.

6552 **Issue 5**6553 Moved from X/OPEN UNIX extension to BASE and aligned with *fchmod()* in the POSIX
6554 Realtime Extension. Specifically, the second paragraph of the DESCRIPTION is added and a
6555 second instance of [EINVAL] is defined in the list of optional errors.

6556 **NAME**

6557 fchown — change owner and group of a file

6558 **SYNOPSIS**

6559 EX #include <unistd.h>

6560 int fchown(int *fildev*, uid_t *owner*, gid_t *group*);

6561

6562 **DESCRIPTION**6563 The *fchown()* function has the same effect as *chown()* except that the file whose owner and group
6564 are to be changed is specified by the file descriptor *fildev*.6565 **RETURN VALUE**6566 Upon successful completion, *fchown()* returns 0. Otherwise, it returns -1 and sets *errno* to
6567 indicate the error.6568 **ERRORS**6569 The *fchown()* function will fail if:6570 [EBADF] The *fildev* argument is not an open file descriptor.6571 [EPERM] The effective user ID does not match the owner of the file or the process does
6572 not have appropriate privilege.6573 [EROFS] The file referred to by *fildev* resides on a read-only file system.6574 The *fchown()* function may fail if:

6575 [EINVAL] The owner or group ID is not a value supported by the implementation.

6576 [EIO] A physical I/O error has occurred.

6577 [EINTR] The *fchown()* function was interrupted by a signal which was caught.6578 **EXAMPLES**

6579 None.

6580 **APPLICATION USAGE**

6581 None.

6582 **FUTURE DIRECTIONS**

6583 None.

6584 **SEE ALSO**6585 *chown()*, <unistd.h>.6586 **CHANGE HISTORY**

6587 First released in Issue 4, Version 2.

6588 **Issue 5**

6589 Moved from X/OPEN UNIX extension to BASE.

6590 **NAME**

6591 fclose — close a stream

6592 **SYNOPSIS**

6593 #include <stdio.h>

6594 int fclose(FILE **stream*);6595 **DESCRIPTION**

6596 The *fclose()* function causes the stream pointed to by *stream* to be flushed and the associated file
 6597 to be closed. Any unwritten buffered data for the stream is written to the file; any unread
 6598 buffered data is discarded. The stream is disassociated from the file. If the associated buffer was
 6599 automatically allocated, it is deallocated. It marks for update the *st_ctime* and *st_mtime* fields of
 6600 the underlying file, if the stream was writable, and if buffered data had not been written to the
 6601 file yet. The *fclose()* function will perform a *close()* on the file descriptor that is associated with
 6602 the stream pointed to by *stream*.

6603 After the call to *fclose()*, any use of *stream* causes undefined behaviour.

6604 **RETURN VALUE**

6605 Upon successful completion, *fclose()* returns 0. Otherwise, it returns EOF and sets *errno* to
 6606 indicate the error.

6607 **ERRORS**

6608 The *fclose()* function will fail if:

6609 [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
 6610 process would be delayed in the write operation.

6611 [EBADF] The file descriptor underlying stream is not valid.

6612 EX [EFBIG] An attempt was made to write a file that exceeds the maximum file size or the
 6613 process' file size limit.

6614 EX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
 6615 offset maximum associated with the corresponding stream.

6616 [EINTR] The *fclose()* function was interrupted by a signal.

6617 [EIO] The process is a member of a background process group attempting to write
 6618 to its controlling terminal, TOSTOP is set, the process is neither ignoring nor
 6619 blocking SIGTTOU and the process group of the process is orphaned. This
 6620 error may also be returned under implementation-dependent conditions.

6621 [ENOSPC] There was no free space remaining on the device containing the file.

6622 [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
 6623 any process. A SIGPIPE signal will also be sent to the thread.

6624 The *fclose()* function may fail if:

6625 EX [ENXIO] A request was made of a non-existent device, or the request was outside the
 6626 capabilities of the device.

6627 **EXAMPLES**

6628 None.

6629 **APPLICATION USAGE**

6630 None.

6631 **FUTURE DIRECTIONS**

6632 None.

6633 **SEE ALSO**

6634 *close()*, *fopen()*, *getrlimit()*, *ulimit()*, **<stdio.h>**.

6635 **CHANGE HISTORY**

6636 First released in Issue 1.

6637 Derived from Issue 1 of the SVID.

6638 **Issue 4**

6639 The following changes are incorporated in this issue:

6640 • The last sentence of the first paragraph in the DESCRIPTION is changed to say *close()*
6641 instead of *fclose()*. This was an error in Issue 3.

6642 • The following paragraph is withdrawn from the DESCRIPTION (by POSIX as well as
6643 X/Open) because of the possibility of causing applications to malfunction, and the
6644 impossibility of implementing these mechanisms for pipes:

6645 If the file is not already at EOF, and the file is one capable of seeking, the file *offset* of the
6646 underlying open file description will be adjusted so that the next operation on the open
6647 file description deals with the byte after the last one read from or written to the stream
6648 being closed.

6649 It is replaced with a statement that any subsequent use of *stream* is undefined.

6650 • The [EFBIG] error is marked to indicate the extensions.

6651 **Issue 4, Version 2**

6652 A cross-reference to *getrlimit()* is added.

6653 **Issue 5**

6654 Large File Summit extensions added.

6655 NAME

6656 fcntl — file control

6657 SYNOPSIS

6658 OH #include <sys/types.h>

6659 #include <unistd.h>

6660 #include <fcntl.h>

6661 int fcntl(int *fil-des*, int *cmd*, ...);

6662 DESCRIPTION

6663 The *fcntl()* function provides for control over open files. The *fil-des* argument is a file descriptor.6664 The available values for *cmd* are defined in the header <fcntl.h>, which include:

6665 F_DUPFD Return a new file descriptor which is the lowest numbered available (that is,
 6666 not already open) file descriptor greater than or equal to the third argument,
 6667 *arg*, taken as an integer of type **int**. The new file descriptor refers to the same
 6668 open file description as the original file descriptor, and shares any locks. The
 6669 FD_CLOEXEC flag associated with the new file descriptor is cleared to keep
 6670 the file open across calls to one of the *exec* functions.

6671 F_GETFD Get the file descriptor flags defined in <fcntl.h> that are associated with the
 6672 file descriptor *fil-des*. File descriptor flags are associated with a single file
 6673 descriptor and do not affect other file descriptors that refer to the same file.

6674 F_SETFD Set the file descriptor flags defined in <fcntl.h>, that are associated with *fil-des*,
 6675 to the third argument, *arg*, taken as type **int**. If the FD_CLOEXEC flag in the
 6676 third argument is 0, the file will remain open across the *exec* functions;
 6677 otherwise the file will be closed upon successful execution of one of the *exec*
 6678 functions.

6679 F_GETFL Get the file status flags and file access modes, defined in <fcntl.h>, for the file
 6680 description associated with *fil-des*. The file access modes can be extracted from
 6681 the return value using the mask O_ACCMODE, which is defined in <fcntl.h>.
 6682 File status flags and file access modes are associated with the file description
 6683 and do not affect other file descriptors that refer to the same file with different
 6684 open file descriptions.

6685 F_SETFL Set the file status flags, defined in <fcntl.h>, for the file description associated
 6686 with *fil-des* from the corresponding bits in the third argument, *arg*, taken as
 6687 type **int**. Bits corresponding to the file access mode and the *oflag* values that
 6688 are set in *arg* are ignored. If any bits in *arg* other than those mentioned here are
 6689 changed by the application, the result is unspecified.

6690 The following values for *cmd* are available for advisory record locking. Record locking is
 6691 supported for regular files, and may be supported for other files.

6692 F_GETLK Get the first lock which blocks the lock description pointed to by the third
 6693 argument, *arg*, taken as a pointer to type **struct flock**, defined in <fcntl.h>.
 6694 The information retrieved overwrites the information passed to *fcntl()* in the
 6695 structure **flock**. If no lock is found that would prevent this lock from being
 6696 created, then the structure will be left unchanged except for the lock type
 6697 which will be set to F_UNLCK.

6698 F_SETLK Set or clear a file segment lock according to the lock description pointed to by
6699 the third argument, *arg*, taken as a pointer to type **struct flock**, defined in
6700 **<fcntl.h>**. F_SETLK is used to establish shared (or read) locks (F_RDLCK) or
6701 exclusive (or write) locks (F_WRLCK), as well as to remove either type of lock
6702 (F_UNLCK). F_RDLCK, F_WRLCK and F_UNLCK are defined in **<fcntl.h>**.
6703 If a shared or exclusive lock cannot be set, *fcntl()* will return immediately with
6704 a return value of **-1**.

6705 F_SETLKW This command is the same as F_SETLK except that if a shared or exclusive
6706 lock is blocked by other locks, the thread will wait until the request can be
6707 satisfied. If a signal that is to be caught is received while *fcntl()* is waiting for
6708 a region, *fcntl()* will be interrupted. Upon return from the signal handler,
6709 *fcntl()* will return **-1** with *errno* set to [EINTR], and the lock operation will not
6710 be done.

6711 Additional implementation-dependent values for *cmd* may be defined in **<fcntl.h>**. Their names
6712 will start with F_.

6713 When a shared lock is set on a segment of a file, other processes will be able to set shared locks
6714 on that segment or a portion of it. A shared lock prevents any other process from setting an
6715 exclusive lock on any portion of the protected area. A request for a shared lock will fail if the file
6716 descriptor was not opened with read access.

6717 An exclusive lock will prevent any other process from setting a shared lock or an exclusive lock
6718 on any portion of the protected area. A request for an exclusive lock will fail if the file descriptor
6719 was not opened with write access.

6720 The structure **flock** describes the type (**l_type**), starting offset (**l_whence**), relative offset
6721 (**l_start**), size (**l_len**) and process ID (**l_pid**) of the segment of the file to be affected.

6722 The value of **l_whence** is SEEK_SET, SEEK_CUR or SEEK_END, to indicate that the relative
6723 offset **l_start** bytes will be measured from the start of the file, current position or end of the file,
6724 EX respectively. The value of **l_len** is the number of consecutive bytes to be locked. The value of
6725 **l_len** may be negative (where the definition of **off_t** permits negative values of **l_len**). The **l_pid**
6726 field is only used with F_GETLK to return the process ID of the process holding a blocking lock.
6727 After a successful F_GETLK request, that is, one in which a lock was found, the value of
6728 **l_whence** will be SEEK_SET.

6729 EX If **l_len** is positive, the area affected starts at **l_start** and ends at **l_start + l_len - 1**. If **l_len** is
6730 negative, the area affected starts at **l_start + l_len** and ends at **l_start - 1**. Locks may start and
6731 extend beyond the current end of a file, but must not be negative relative to the beginning of the
6732 file. A lock will be set to extend to the largest possible value of the file offset for that file by
6733 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
6734 whole file will be locked.

6735 There will be at most one type of lock set for each byte in the file. Before a successful return
6736 from an F_SETLK or an F_SETLKW request when the calling process has previously existing
6737 locks on bytes in the region specified by the request, the previous lock type for each byte in the
6738 specified region will be replaced by the new lock type. As specified above under the
6739 descriptions of shared locks and exclusive locks, an F_SETLK or an F_SETLKW request will
6740 (respectively) fail or block when another process has existing locks on bytes in the specified
6741 region and the type of any of those locks conflicts with the type specified in the request.

6742 All locks associated with a file for a given process are removed when a file descriptor for that file
6743 is closed by that process or the process holding that file descriptor terminates. Locks are not
6744 inherited by a child process created using *fork()*.

6745 A potential for deadlock occurs if a process controlling a locked region is put to sleep by
 6746 attempting to lock another process' locked region. If the system detects that sleeping until a
 6747 locked region is unlocked would cause a deadlock, *fcntl()* will fail with an [EDEADLK] error.

6748 RT If `_XOPEN_REALTIME` is defined and has a value other than `-1`:

6749 When the file descriptor *fildev* refers to a shared memory object, the behaviour of *fcntl()* is the
 6750 same as for a regular file except the effect of the following values for the argument *cmd* are
 6751 unspecified: `F_SETFL`, `F_GETLK`, `F_SETLK`, and `F_SETLKW`.

6752 EX An unlock (`F_UNLCK`) request in which *l_len* is non-zero and the offset of the last byte of the
 6753 requested segment is the maximum value for an object of type `off_t`, when the process has an
 6754 existing lock in which *l_len* is 0 and which includes the last byte of the requested segment, will
 6755 be treated as a request to unlock from the start of the requested segment with an *l_len* equal to 0.
 6756 Otherwise an unlock (`F_UNLCK`) request will attempt to unlock only the requested segment.

6757 RETURN VALUE

6758 Upon successful completion, the value returned depends on *cmd* as follows:

6759 `F_DUPFD` A new file descriptor.

6760 `F_GETFD` Value of flags defined in `<fcntl.h>`. The return value will not be negative.

6761 `F_SETFD` Value other than `-1`.

6762 `F_GETFL` Value of file status flags and access modes. The return value will not be
 6763 negative.

6764 `F_SETFL` Value other than `-1`.

6765 `F_GETLK` Value other than `-1`.

6766 `F_SETLK` Value other than `-1`.

6767 `F_SETLKW` Value other than `-1`.

6768 Otherwise, `-1` is returned and *errno* is set to indicate the error.

6769 ERRORS

6770 The *fcntl()* function will fail if:

6771 [EACCES] or [EAGAIN]

6772 The *cmd* argument is `F_SETLK`; the type of lock (*l_type*) is a shared
 6773 (`F_RDLCK`) or exclusive (`F_WRLCK`) lock and the segment of a file to be
 6774 locked is already exclusive-locked by another process, or the type is an
 6775 exclusive lock and some portion of the segment of a file to be locked is already
 6776 shared-locked or exclusive-locked by another process.

6777 [EBADF] The *fildev* argument is not a valid open file descriptor, or the argument *cmd* is
 6778 `F_SETLK` or `F_SETLKW`, the type of lock, *l_type*, is a shared lock (`F_RDLCK`),
 6779 and *fildev* is not a valid file descriptor open for reading, or the type of lock
 6780 *l_type*, is an exclusive lock (`F_WRLCK`), and *fildev* is not a valid file descriptor
 6781 open for writing.

6782 [EINTR] The *cmd* argument is `F_SETLKW` and the function was interrupted by a signal.

6783 EX [EINVAL] The *cmd* argument is invalid, or the *cmd* argument is `F_DUPFD` and *arg* is
 6784 negative or greater than or equal to `{OPEN_MAX}`, or the *cmd* argument is
 6785 `F_GETLK`, `F_SETLK` or `F_SETLKW` and the data pointed to by *arg* is not valid,
 6786 or *fildev* refers to a file that does not support locking.

6787 [EMFILE] The argument *cmd* is F_DUPFD and {OPEN_MAX} file descriptors are
 6788 currently open in the calling process, or no file descriptors greater than or
 6789 equal to *arg* are available.

6790 [ENOLCK] The argument *cmd* is F_SETLK or F_SETLKW and satisfying the lock or unlock
 6791 request would result in the number of locked regions in the system exceeding
 6792 a system-imposed limit.

6793 EX [EOVERFLOW] One of the values to be returned cannot be represented correctly.

6794 EX [EOVERFLOW] The *cmd* argument is F_GETLK, F_SETLK or F_SETLKW and the smallest or, if
 6795 *l_len* is non-zero, the largest offset of any byte in the requested segment
 6796 cannot be represented correctly in an object of type **off_t**.

6797 The *fcntl()* function may fail if:

6798 [EDEADLK] The *cmd* argument is F_SETLKW, the lock is blocked by some lock from
 6799 another process and putting the calling process to sleep, waiting for that lock
 6800 to become free would cause a deadlock.

6801 EXAMPLES

6802 None.

6803 APPLICATION USAGE

6804 None.

6805 FUTURE DIRECTIONS

6806 None.

6807 SEE ALSO

6808 *close()*, *exec*, *open()*, *sigaction()*, <fcntl.h>, <signal.h>, <sys/types.h>, <unistd.h>.

6809 CHANGE HISTORY

6810 First released in Issue 1.

6811 Derived from Issue 1 of the SVID.

6812 Issue 4

6813 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 6814 • In the DESCRIPTION, the meaning of a successful F_SETLK or F_SETLKW request is
 6815 clarified, after a POSIX Request for Interpretation.

6816 Other changes are incorporated as follows:

- 6817 • The <sys/types.h> and <unistd.h> headers are now marked as optional (OH); these headers
 6818 do not need to be included on XSI-conformant systems.
- 6819 • In the DESCRIPTION (a) sentences describing behaviour when *l_len* is negative are marked
 6820 as an extension and (b) the description of locks is corrected to make it a requirement on the
 6821 application.

6822 Issue 5

6823 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
 6824 Threads Extension.

6825 Large File Summit extensions added.

6826 **NAME**

6827 fcvt — convert a floating-point number to a string

6828 **SYNOPSIS**

6829 EX #include <stdlib.h>

6830 char *fcvt(double *value*, int *ndigit*, int **decpt*, int **sign*);

6831

6832 **DESCRIPTION**6833 Refer to *ecvt()*.6834 **CHANGE HISTORY**

6835 First released in Issue 4, Version 2.

6836 **Issue 5**

6837 Moved from X/OPEN UNIX extension to BASE.

6838 **NAME**

6839 FD_CLR — macros for synchronous I/O multiplexing

6840 **SYNOPSIS**6841 EX `#include <sys/time.h>`

```
6842     FD_CLR(int fd, fd_set *fdset);
6843     FD_ISSET(int fd, fd_set *fdset);
6844     FD_SET(int fd, fd_set *fdset);
6845     FD_ZERO(fd_set *fdset);
6846
```

6847 **DESCRIPTION**6848 Refer to *select()*.6849 **SEE ALSO**6850 `<sys/time.h>`.6851 **CHANGE HISTORY**

6852 First released in Issue 4, Version 2.

6853 **Issue 5**

6854 Moved from X/OPEN UNIX extension to BASE.

6855 **NAME**6856 *fdatasync* — synchronise the data of a file (**REALTIME**)6857 **SYNOPSIS**6858 RT `#include <unistd.h>`6859 `int fdatasync(int fildes);`

6860

6861 **DESCRIPTION**6862 The *fdatasync()* function forces all currently queued I/O operations associated with the file
6863 indicated by file descriptor *fil*des to the synchronised I/O completion state.6864 The functionality is as described for *fsync()* (with the symbol `_XOPEN_REALTIME` defined),
6865 with the exception that all I/O operations are completed as defined for synchronised I/O data
6866 integrity completion.6867 **RETURN VALUE**6868 If successful, the *fdatasync()* function returns the value 0. Otherwise, the function returns the
6869 value `-1` and sets *errno* to indicate the error. If the *fdatasync()* function fails, outstanding I/O
6870 operations are not guaranteed to have been completed.6871 **ERRORS**6872 The *fdatasync()* function will fail if:6873 [EBADF] The *fil*des argument is not a valid file descriptor open for writing.

6874 [EINVAL] This implementation does not support synchronised I/O for this file.

6875 [ENOSYS] The function *fdatasync()* is not supported by this implementation.6876 In the event that any of the queued I/O operations fail, *fdatasync()* returns the error conditions
6877 defined for *read()* and *write()*.6878 **EXAMPLES**

6879 None.

6880 **APPLICATION USAGE**

6881 None.

6882 **FUTURE DIRECTIONS**

6883 None.

6884 **SEE ALSO**6885 *aio_fsync()*, *fcntl()*, *fsync()*, *open()*, *read()*, *write()*.6886 **CHANGE HISTORY**

6887 First released in Issue 5.

6888 Included for alignment with the POSIX Realtime Extension.

6889 **NAME**

6890 fdetach — detach a name from a STREAMS-based file descriptor

6891 **SYNOPSIS**6892 EX `#include <stropts.h>`6893 `int fdetach(const char *path);`

6894

6895 **DESCRIPTION**

6896 The *fdetach()* function detaches a STREAMS-based file from the file to which it was attached by a
 6897 previous call to *fattach()*. The *path* argument points to the pathname of the attached STREAMS
 6898 file. The process must have appropriate privileges or be the owner of the file. A successful call
 6899 to *fdetach()* causes all pathnames that named the attached STREAMS file to again name the file
 6900 to which the STREAMS file was attached. All subsequent operations on *path* will operate on the
 6901 underlying file and not on the STREAMS file.

6902 All open file descriptions established while the STREAMS file was attached to the file referenced
 6903 by *path*, will still refer to the STREAMS file after the *fdetach()* has taken effect.

6904 If there are no open file descriptors or other references to the STREAMS file, then a successful
 6905 call to *fdetach()* has the same effect as performing the last *close()* on the attached file.

6906 **RETURN VALUE**

6907 Upon successful completion, *fdetach()* returns 0. Otherwise, it returns *-1* and sets *errno* to
 6908 indicate the error.

6909 **ERRORS**6910 The *fdetach()* function will fail if:

- 6911 [EACCES] Search permission is denied on a component of the path prefix.
- 6912 [EPERM] The effective user ID is not the owner of *path* and the process does not have
6913 appropriate privileges.
- 6914 [ENOTDIR] A component of the path prefix is not a directory.
- 6915 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.
- 6916 [EINVAL] The *path* argument names a file that is not currently attached.
- 6917 [ENAMETOOLONG]
- 6918 The size of a pathname exceeds {PATH_MAX}, or a pathname component is
6919 longer than {NAME_MAX}.
- 6920 [ELOOP] Too many symbolic links were encountered in resolving *path*.

6921 The *fdetach()* function may fail if:

- 6922 [ENAMETOOLONG]
- 6923 Pathname resolution of a symbolic link produced an intermediate result
6924 whose length exceeds {PATH_MAX}.

6925 **EXAMPLES**

6926 None.

6927 **APPLICATION USAGE**

6928 None.

6929 **FUTURE DIRECTIONS**

6930 None.

6931 **SEE ALSO**6932 *fattach()*, <stropts.h>.6933 **CHANGE HISTORY**

6934 First released in Issue 4, Version 2.

6935 **Issue 5**

6936 Moved from X/OPEN UNIX extension to BASE.

6937 **NAME**

6938 fdopen — associate a stream with a file descriptor

6939 **SYNOPSIS**

6940 #include <stdio.h>

6941 FILE *fdopen(int *fildes*, const char **mode*);6942 **DESCRIPTION**6943 The *fdopen()* function associates a stream with a file descriptor.6944 The *mode* argument is a character string having one of the following values:

6945 EX r or rb Open a file for reading.

6946 EX w or wb Open a file for writing.

6947 EX a or ab Open a file for writing at end of file.

6948 EX r+ or rb+ or r+b Open a file for update (reading and writing).

6949 EX w+ or wb+ or w+b Open a file for update (reading and writing).

6950 EX a+ or ab+ or a+b Open a file for update (reading and writing) at end of file.

6951 The meaning of these flags is exactly as specified in *fopen()*, except that modes beginning with *w*
6952 do not cause truncation of the file.6953 Additional values for the *mode* argument may be supported by an implementation.6954 The mode of the stream must be allowed by the file access mode of the open file. The file
6955 position indicator associated with the new stream is set to the position indicated by the file
6956 offset associated with the file descriptor.6957 EX The error and end-of-file indicators for the stream are cleared. The *fdopen()* function may cause
6958 the *st_atime* field of the underlying file to be marked for update.6959 RT If *fildes* refers to a shared memory object, the result of the *fdopen()* function is unspecified.6960 EX The *fdopen()* function will preserve the offset maximum previously set for the open file
6961 description corresponding to *fildes*.6962 **RETURN VALUE**6963 Upon successful completion, *fdopen()* returns a pointer to a stream. Otherwise, a null pointer is
6964 returned and *errno* is set to indicate the error.6965 **ERRORS**6966 The *fdopen()* function may fail if:6967 EX [EBADF] The *fildes* argument is not a valid file descriptor.6968 [EINVAL] The *mode* argument is not a valid mode.

6969 [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

6970 [EMFILE] {STREAM_MAX} streams are currently open in the calling process.

6971 [ENOMEM] Insufficient space to allocate a buffer.

6972 **EXAMPLES**

6973 None.

6974 **APPLICATION USAGE**6975 File descriptors are obtained from calls like *open()*, *dup()*, *creat()* or *pipe()*, which open files but
6976 do not return streams.

6977 **FUTURE DIRECTIONS**

6978 None.

6979 **SEE ALSO**6980 *fclose()*, *fopen()*, *open()*, <stdio.h>, Section 2.4.1 on page 30.6981 **CHANGE HISTORY**

6982 First released in Issue 1.

6983 Derived from Issue 1 of the SVID.

6984 **Issue 4**

6985 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 6986
- The type of argument *mode* is changed from **char *** to **const char ***.

6987 Other changes are incorporated as follows:

- 6988
- In the DESCRIPTION, the use and settings of the *mode* argument are changed to include binary streams and are marked as extensions.

- 6990
- All errors identified in the ERRORS section are marked as extensions, and the [EMFILE] error is added.

- 6992
- The APPLICATION USAGE section is added.

6993 **Issue 5**

6994 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

6995 Large File Summit extensions added.

6996	NAME	
6997	feof — test end-of-file indicator on a stream	
6998	SYNOPSIS	
6999	#include <stdio.h>	
7000	int feof(FILE * <i>stream</i>);	
7001	DESCRIPTION	
7002	The <i>feof()</i> function tests the end-of-file indicator for the stream pointed to by <i>stream</i> .	
7003	RETURN VALUE	
7004	The <i>feof()</i> function returns non-zero if and only if the end-of-file indicator is set for <i>stream</i> .	
7005	ERRORS	
7006	No errors are defined.	
7007	EXAMPLES	
7008	None.	
7009	APPLICATION USAGE	
7010	None.	
7011	FUTURE DIRECTIONS	
7012	None.	
7013	SEE ALSO	
7014	<i>clearerr()</i> , <i>ferror()</i> , <i>fopen()</i> , <stdio.h>.	
7015	CHANGE HISTORY	
7016	First released in Issue 1.	
7017	Derived from Issue 1 of the SVID.	
7018	Issue 4	
7019	The following change is incorporated in this issue:	
7020	• The ERRORS section is rewritten, such that no error return values are now defined for this	
7021	interface.	

7022 **NAME**7023 *ferror* — test error indicator on a stream7024 **SYNOPSIS**7025 `#include <stdio.h>`7026 `int ferror(FILE *stream);`7027 **DESCRIPTION**7028 The *ferror()* function tests the error indicator for the stream pointed to by *stream*.7029 **RETURN VALUE**7030 The *ferror()* function returns non-zero if and only if the error indicator is set for *stream*.7031 **ERRORS**

7032 No errors are defined.

7033 **EXAMPLES**

7034 None.

7035 **APPLICATION USAGE**

7036 None.

7037 **FUTURE DIRECTIONS**

7038 None.

7039 **SEE ALSO**7040 *clearerr()*, *feof()*, *fopen()*, `<stdio.h>`.7041 **CHANGE HISTORY**

7042 First released in Issue 1.

7043 Derived from Issue 1 of the SVID.

7044 **Issue 4**

7045 The following change is incorporated in this issue:

- 7046 • The ERRORS section is rewritten, such that no error return values are now defined for this
- 7047 interface.

7048 **NAME**

7049 fflush — flush a stream

7050 **SYNOPSIS**

7051 #include <stdio.h>

7052 int fflush(FILE **stream*);7053 **DESCRIPTION**

7054 If *stream* points to an output stream or an update stream in which the most recent operation was
 7055 not input, *fflush()* causes any unwritten data for that stream to be written to the file, and the
 7056 *st_ctime* and *st_mtime* fields of the underlying file are marked for update.

7057 If *stream* is a null pointer, *fflush()* performs this flushing action on all streams for which the
 7058 behaviour is defined above.

7059 **RETURN VALUE**

7060 Upon successful completion, *fflush()* returns 0. Otherwise, it returns EOF and sets *errno* to
 7061 indicate the error.

7062 **ERRORS**7063 The *fflush()* function will fail if:

7064 [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
 7065 process would be delayed in the write operation.

7066 [EBADF] The file descriptor underlying *stream* is not valid.

7067 EX [EFBIG] An attempt was made to write a file that exceeds the maximum file size or the
 7068 process' file size limit.

7069 EX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
 7070 offset maximum associated with the corresponding stream.

7071 [EINTR] The *fflush()* function was interrupted by a signal.

7072 [EIO] The process is a member of a background process group attempting to write
 7073 to its controlling terminal, TOSTOP is set, the process is neither ignoring nor
 7074 blocking SIGTTOU and the process group of the process is orphaned. This
 7075 error may also be returned under implementation-dependent conditions.

7076 [ENOSPC] There was no free space remaining on the device containing the file.

7077 [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
 7078 any process. A SIGPIPE signal will also be sent to the thread.

7079 The *fflush()* function may fail if:

7080 EX [ENXIO] A request was made of a non-existent device, or the request was outside the
 7081 capabilities of the device.

7082 **EXAMPLES**

7083 None.

7084 **APPLICATION USAGE**

7085 None.

7086 **FUTURE DIRECTIONS**

7087 None.

7088 **SEE ALSO**7089 *getrlimit()*, *ulimit()*, <stdio.h>.

7090 **CHANGE HISTORY**

7091 First released in Issue 1.

7092 Derived from Issue 1 of the SVID.

7093 **Issue 4**

7094 The following change is incorporated for alignment with the ISO C standard:

- 7095
- The DESCRIPTION is changed to describe the behaviour of *fflush()* if *stream* is a null pointer.

7096 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- 7097
- The following two paragraphs are withdrawn from the DESCRIPTION (by POSIX as well as
-
- 7098 X/Open) because of the possibility of causing applications to malfunction, and the
-
- 7099 impossibility of implementing these mechanisms for pipes:

7100 If the stream is open for reading, any unread data buffered in the stream is discarded.

7101 For a stream open for reading, if the file is not already at EOF, and the file is one capable
7102 of seeking, the file offset of the underlying open file description is adjusted so that the
7103 next operation on the open file description deals with the byte after the last one read
7104 from, or written to, the stream being flushed.

- 7105
- The [EFBIG] error is marked to indicate the extensions.

7106 **Issue 5**

7107 Large File Summit extensions added.

7108 **NAME**

7109 ffs — find first set bit

7110 **SYNOPSIS**

7111 EX #include <strings.h>

7112 int ffs(int i);

7113

7114 **DESCRIPTION**

7115 The *ffs()* function finds the first bit set (beginning with the least significant bit) and returns the
7116 index of that bit. Bits are numbered starting at one (the least significant bit).

7117 **RETURN VALUE**7118 The *ffs()* function returns the index of the first bit set. If *i* is 0, then *ffs()* returns 0.7119 **ERRORS**

7120 No errors are defined.

7121 **EXAMPLES**

7122 None.

7123 **APPLICATION USAGE**

7124 None.

7125 **FUTURE DIRECTIONS**

7126 None.

7127 **SEE ALSO**

7128 <strings.h>.

7129 **CHANGE HISTORY**

7130 First released in Issue 4, Version 2.

7131 **Issue 5**

7132 Moved from X/OPEN UNIX extension to BASE.

7133 **NAME**

7134 fgetc — get a byte from a stream

7135 **SYNOPSIS**

7136 #include <stdio.h>

7137 int fgetc(FILE **stream*);7138 **DESCRIPTION**

7139 The *fgetc()* function obtains the next byte (if present) as an **unsigned char** converted to an **int**,
 7140 from the input stream pointed to by *stream*, and advances the associated file position indicator
 7141 for the stream (if defined).

7142 The *fgetc()* function may mark the *st_atime* field of the file associated with *stream* for update.
 7143 The *st_atime* field will be marked for update by the first successful execution of *fgetc()*, *fgets()*,
 7144 *fgetwc()*, *fgetws()*, *fread()*, *fscanf()*, *getc()*, *getchar()*, *gets()* or *scanf()* using *stream* that returns
 7145 data not supplied by a prior call to *ungetc()* or *ungetwc()*.

7146 **RETURN VALUE**

7147 Upon successful completion, *fgetc()* returns the next byte from the input stream pointed to by
 7148 *stream*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and *fgetc()*
 7149 returns EOF. If a read error occurs, the error indicator for the stream is set, *fgetc()* returns EOF
 7150 and sets *errno* to indicate the error.

7151 **ERRORS**7152 The *fgetc()* function will fail if data needs to be read and:

7153 [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
 7154 process would be delayed in the *fgetc()* operation.

7155 [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
 7156 reading.

7157 [EINTR] The read operation was terminated due to the receipt of a signal, and no data
 7158 was transferred.

7159 EX [EIO] A physical I/O error has occurred, or the process is in a background process
 7160 group attempting to read from its controlling terminal, and either the process
 7161 is ignoring or blocking the SIGTTIN signal or the process group is orphaned.
 7162 This error may also be generated for implementation-dependent reasons.

7163 EX [EOVERFLOW] The file is a regular file and an attempt was made to read at or beyond the
 7164 offset maximum associated with the corresponding stream.

7165 The *fgetc()* function may fail if:

7166 EX [ENOMEM] Insufficient storage space is available.

7167 EX [ENXIO] A request was made of a non-existent device, or the request was outside the
 7168 capabilities of the device.

7169 **EXAMPLES**

7170 None.

7171 **APPLICATION USAGE**

7172 If the integer value returned by *fgetc()* is stored into a variable of type **char** and then compared
 7173 against the integer constant EOF, the comparison may never succeed, because sign-extension of
 7174 a variable of type **char** on widening to integer is implementation-dependent.

7175 The *ferror()* or *feof()* functions must be used to distinguish between an error condition and an
 7176 end-of-file condition.

7177 **FUTURE DIRECTIONS**

7178 None.

7179 **SEE ALSO**7180 *feof()*, *ferror()*, *fopen()*, *getchar()*, *getc()*, *<stdio.h>*.7181 **CHANGE HISTORY**

7182 First released in Issue 1.

7183 Derived from Issue 1 of the SVID.

7184 **Issue 4**

7185 The following changes are incorporated in this issue:

7186 • In the DESCRIPTION:

7187 — The text is changed to make it clear that the function returns a byte value.

7188 — The list of functions that may cause the *st_atime* field to be updated is revised.7189 • In the ERRORS section, text is added to indicate that error returns will only be generated
7190 when data needs to be read into the stream buffer.7191 • Also in the ERRORS section, in previous issues generation of the [EIO] error depended on
7192 whether or not an implementation supported Job Control. This functionality is now defined
7193 as mandatory.

7194 • The [ENXIO] and [ENOMEM] errors are marked as extensions.

7195 • In the APPLICATION USAGE section, text is added to indicate how an application can
7196 distinguish between an error condition and an end-of-file condition.

7197 • The description of [EINTR] is amended.

7198 **Issue 4, Version 2**7199 In the ERRORS section, the description of [EIO] is updated to include the case where a physical
7200 I/O error occurs.7201 **Issue 5**

7202 Large File Summit extensions added.

7203 **NAME**

7204 fgetpos — get current file position information

7205 **SYNOPSIS**

7206 #include <stdio.h>

7207 int fgetpos(FILE **stream*, fpos_t **pos*);7208 **DESCRIPTION**

7209 The *fgetpos()* function stores the current value of the file position indicator for the stream
 7210 pointed to by *stream* in the object pointed to by *pos*. The value stored contains unspecified
 7211 information usable by *fsetpos()* for repositioning the stream to its position at the time of the call
 7212 to *fgetpos()*.

7213 **RETURN VALUE**

7214 Upon successful completion, *fgetpos()* returns 0. Otherwise, it returns a non-zero value and sets
 7215 *errno* to indicate the error.

7216 **ERRORS**7217 EX The *fgetpos()* function will fail if:

7218 [E_OVERFLOW] The current value of the file position cannot be represented correctly in an
 7219 object of type **fpos_t**.

7220 The *fgetpos()* function may fail if:7221 EX [E_BADF] The file descriptor underlying *stream* is not valid.7222 [E_SPIPE] The file descriptor underlying *stream* is associated with a pipe or FIFO.7223 **EXAMPLES**

7224 None.

7225 **APPLICATION USAGE**

7226 None.

7227 **FUTURE DIRECTIONS**

7228 None.

7229 **SEE ALSO**7230 *fopen()*, *ftell()*, *rewind()*, *ungetc()*, <stdio.h>.7231 **CHANGE HISTORY**

7232 First released in Issue 4.

7233 Derived from the ISO C standard.

7234 **Issue 5**

7235 Large File Summit extensions added.

7236 **NAME**

7237 fgets — get a string from a stream

7238 **SYNOPSIS**

7239 #include <stdio.h>

7240 char *fgets(char *s, int n, FILE *stream);

7241 **DESCRIPTION**

7242 The *fgets()* function reads bytes from *stream* into the array pointed to by *s*, until *n*−1 bytes are
7243 read, or a newline character is read and transferred to *s*, or an end-of-file condition is
7244 encountered. The string is then terminated with a null byte.

7245 The *fgets()* function may mark the *st_atime* field of the file associated with *stream* for update. The
7246 *st_atime* field will be marked for update by the first successful execution of *fgetc()*, *fgets()*,
7247 *fgetwc()*, *fgetws()*, *fread()*, *fscanf()*, *getc()*, *getchar()*, *gets()* or *scanf()* using *stream* that returns
7248 data not supplied by a prior call to *ungetc()* or *ungetwc()*.

7249 **RETURN VALUE**

7250 Upon successful completion, *fgets()* returns *s*. If the stream is at end-of-file, the end-of-file
7251 indicator for the stream is set and *fgets()* returns a null pointer. If a read error occurs, the error
7252 indicator for the stream is set, *fgets()* returns a null pointer and sets *errno* to indicate the error.

7253 **ERRORS**7254 Refer to *fgetc()*.7255 **EXAMPLES**

7256 None.

7257 **APPLICATION USAGE**

7258 None.

7259 **FUTURE DIRECTIONS**

7260 None.

7261 **SEE ALSO**7262 *fopen()*, *fread()*, *gets()*, <stdio.h>.7263 **CHANGE HISTORY**

7264 First released in Issue 1.

7265 Derived from Issue 1 of the SVID.

7266 **Issue 4**

7267 The following change is incorporated in this issue:

- 7268 • In the DESCRIPTION (a) the text is changed to make it clear that the function reads bytes
7269 rather than (possibly multi-byte) characters, and (b) the list of functions that may cause the
7270 *st_atime* field to be updated is revised.

7271 **NAME**7272 `fgetwc` — get a wide-character code from a stream7273 **SYNOPSIS**7274 `#include <stdio.h>`7275 `#include <wchar.h>`7276 `wint_t fgetwc(FILE *stream);`7277 **DESCRIPTION**

7278 The `fgetwc()` function obtains the next character (if present) from the input stream pointed to by *stream*, converts that to the corresponding wide-character code and advances the associated file position indicator for the stream (if defined).

7281 If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

7283 The `fgetwc()` function may mark the `st_atime` field of the file associated with *stream* for update. The `st_atime` field will 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()`.

7287 **RETURN VALUE**

7288 Upon successful completion the `fgetwc()` function returns the wide-character code of the character read from the input stream pointed to by *stream* converted to a type `wint_t`. If the stream is at end-of-file, the end-of-file indicator for the stream is set and `fgetwc()` returns WEOF. If a read error occurs, the error indicator for the stream is set, `fgetwc()` returns WEOF and sets `errno` to indicate the error.

7293 **ERRORS**7294 The `fgetwc()` function will fail if data needs to be read and:

7295 [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the process would be delayed in the `fgetwc()` operation.

7297 [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for reading.

7299 [EINTR] The read operation was terminated due to the receipt of a signal, and no data was transferred.

7301 EX [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-dependent reasons.

7305 EX [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.

7307 The `fgetwc()` function may fail if:

7308 [ENOMEM] Insufficient storage space is available.

7309 [ENXIO] A request was made of a non-existent device, or the request was outside the capabilities of the device.

7311 [EILSEQ] The data obtained from the input stream does not form a valid character.

7312 **EXAMPLES**

7313 None.

7314 **APPLICATION USAGE**

7315 The *ferror()* or *feof()* functions must be used to distinguish between an error condition and an
7316 end-of-file condition.

7317 **FUTURE DIRECTIONS**

7318 None.

7319 **SEE ALSO**

7320 *feof()*, *ferror()*, *fopen()*, `<stdio.h>`, `<wchar.h>`.

7321 **CHANGE HISTORY**

7322 First released in Issue 4.

7323 Derived from the MSE working draft.

7324 **Issue 4, Version 2**

7325 In the ERRORS section, the description of [EIO] is updated to include the case where a physical
7326 I/O error occurs.

7327 **Issue 5**

7328 The Optional Header (OH) marking is removed from `<stdio.h>`.

7329 Large File Summit extensions added.

7330 **NAME**7331 `fgetws` — get a wide-character string from a stream7332 **SYNOPSIS**7333 `#include <stdio.h>`7334 `#include <wchar.h>`7335 `wchar_t *fgetws(wchar_t *ws, int n, FILE *stream);`7336 **DESCRIPTION**

7337 The `fgetws()` function reads characters from the *stream*, converts these to the corresponding
 7338 wide-character codes, places them in the **wchar_t** array pointed to by *ws*, until *n*−1 characters are
 7339 read, or a newline character is read, converted and transferred to *ws*, or an end-of-file condition
 7340 is encountered. The wide-character string, *ws*, is then terminated with a null wide-character
 7341 code.

7342 If an error occurs, the resulting value of the file position indicator for the stream is
 7343 indeterminate.

7344 The `fgetws()` function may mark the *st_atime* field of the file associated with *stream* for update.
 7345 The *st_atime* field will be marked for update by the first successful execution of `fgetc()`, `fgets()`,
 7346 `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()` or `scanf()` using *stream* that returns
 7347 data not supplied by a prior call to `ungetc()` or `ungetwc()`.

7348 **RETURN VALUE**

7349 Upon successful completion, `fgetws()` returns *ws*. If the stream is at end-of-file, the end-of-file
 7350 indicator for the stream is set and `fgetws()` returns a null pointer. If a read error occurs, the error
 7351 indicator for the stream is set, `fgetws()` returns a null pointer and sets *errno* to indicate the error.

7352 **ERRORS**7353 Refer to `fgetwc()`.7354 **EXAMPLES**

7355 None.

7356 **APPLICATION USAGE**

7357 None.

7358 **FUTURE DIRECTIONS**

7359 None.

7360 **SEE ALSO**7361 `fopen()`, `fread()`, `<stdio.h>`, `<wchar.h>`.7362 **CHANGE HISTORY**

7363 First released in Issue 4.

7364 Derived from the MSE working draft.

7365 **Issue 5**7366 The Optional Header (OH) marking is removed from `<stdio.h>`.

7367 NAME

7368 `fileno` — map a stream pointer to a file descriptor

7369 SYNOPSIS

7370 `#include <stdio.h>`

7371 `int fileno(FILE *stream);`

7372 DESCRIPTION

7373 The `fileno()` function returns the integer file descriptor associated with the stream pointed to by
7374 *stream*.

7375 RETURN VALUE

7376 Upon successful completion, `fileno()` returns the integer value of the file descriptor associated
7377 with *stream*. Otherwise, the value `-1` is returned and `errno` is set to indicate the error.

7378 ERRORS

7379 The `fileno()` function may fail if:

7380 EX [EBADF] The *stream* argument is not a valid stream.

7381 EXAMPLES

7382 None.

7383 APPLICATION USAGE

7384 None.

7385 FUTURE DIRECTIONS

7386 None.

7387 SEE ALSO

7388 `fdopen()`, `fopen()`, `stdin`, `<stdio.h>`, Section 2.4.1 on page 30.

7389 CHANGE HISTORY

7390 First released in Issue 1.

7391 Derived from Issue 1 of the SVID.

7392 Issue 4

7393 The following change is incorporated in this issue:

- 7394 • The [EBADF] error is marked as an extension.

7395 **NAME**

7396 flockfile, ftrylockfile, funlockfile — stdio locking functions

7397 **SYNOPSIS**

7398 #include <stdio.h>

7399 void flockfile(FILE *file);

7400 int ftrylockfile(FILE *file);

7401 void funlockfile(FILE *file);

7402 **DESCRIPTION**

7403 The *flockfile()*, *ftrylockfile()* and *funlockfile()* functions provide for explicit application-level
 7404 locking of stdio (**FILE***) objects. These functions can be used by a thread to delineate a sequence
 7405 of I/O statements that are to be executed as a unit.

7406 The *flockfile()* function is used by a thread to acquire ownership of a (**FILE***) object.

7407 The *ftrylockfile()* function is used by a thread to acquire ownership of a (**FILE***) object if the
 7408 object is available; *ftrylockfile()* is a non-blocking version of *flockfile()*.

7409 The *funlockfile()* function is used to relinquish the ownership granted to the thread. The
 7410 behaviour is undefined if a thread other than the current owner calls the *funlockfile()* function.

7411 Logically, there is a lock count associated with each (**FILE***) object. This count is implicitly
 7412 initialised to zero when the (**FILE***) object is created. The (**FILE***) object is unlocked when the
 7413 count is zero. When the count is positive, a single thread owns the (**FILE***) object. When the
 7414 *flockfile()* function is called, if the count is zero or if the count is positive and the caller owns the
 7415 (**FILE***) object, the count is incremented. Otherwise, the calling thread is suspended, waiting for
 7416 the count to return to zero. Each call to *funlockfile()* decrements the count. This allows matching
 7417 calls to *flockfile()* (or successful calls to *ftrylockfile()*) and *funlockfile()* to be nested.

7418 All functions that reference (**FILE***) objects behave as if they use *flockfile()* and *funlockfile()*
 7419 internally to obtain ownership of these (**FILE***) objects.

7420 **RETURN VALUE**

7421 None for *flockfile()* and *funlockfile()*. The function *ftrylock()* returns zero for success and non-
 7422 zero to indicate that the lock cannot be acquired.

7423 **ERRORS**

7424 No errors are defined.

7425 **EXAMPLES**

7426 None.

7427 **APPLICATION USAGE**

7428 Realtime applications may encounter priority inversion when using FILE locks. The problem
 7429 occurs when a high priority thread “locks” a FILE that is about to be “unlocked” by a low
 7430 priority thread, but the low priority thread is preempted by a medium priority thread. This
 7431 scenario leads to priority inversion; a high priority thread is blocked by lower priority threads
 7432 for an unlimited period of time. During system design, realtime programmers must take into
 7433 account the possibility of this kind of priority inversion. They can deal with it in a number of
 7434 ways, such as by having critical sections that are guarded by FILE locks execute at a high
 7435 priority, so that a thread cannot be preempted while executing in its critical section.

7436 **FUTURE DIRECTIONS**

7437 None.

7438 **SEE ALSO**7439 *getc_unlocked()*, *getchar_unlocked()*, *putc_unlocked()*, *putchar_unlocked()*, <stdio.h>.

7440 **CHANGE HISTORY**

7441 First released in Issue 5.

7442 Included for alignment with the POSIX Threads Extension.

7443 **NAME**

7444 floor — floor function

7445 **SYNOPSIS**

7446 #include <math.h>

7447 double floor(double x);

7448 **DESCRIPTION**7449 The *floor()* function computes the largest integral value not greater than *x*.

7450 An application wishing to check for error situations should set *errno* to 0 before calling *floor()*. If
 7451 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

7452 **RETURN VALUE**

7453 Upon successful completion, *floor()* returns the largest integral value not greater than *x*,
 7454 expressed as a **double**.

7455 EX If *x* is NaN, NaN is returned and *errno* may be set to [EDOM].

7456 If the correct value would cause overflow, -HUGE_VAL is returned and *errno* is set to
 7457 [ERANGE].

7458 EX If *x* is ±Inf or ±0, the value of *x* is returned.7459 **ERRORS**7460 The *floor()* function will fail if:

7461 [ERANGE] The result would cause an overflow.

7462 The *floor()* function may fail if:7463 EX [EDOM] The value of *x* is NaN.

7464 EX No other errors will occur.

7465 **EXAMPLES**

7466 None.

7467 **APPLICATION USAGE**

7468 The integral value returned by *floor()* as a **double** might not be expressible as an **int** or **long int**.
 7469 The return value should be tested before assigning it to an integer type to avoid the undefined
 7470 results of an integer overflow.

7471 The *floor()* function can only overflow when the floating point representation has
 7472 DBL_MANT_DIG > DBL_MAX_EXP.

7473 **FUTURE DIRECTIONS**

7474 None.

7475 **SEE ALSO**7476 *ceil()*, *isnan()*, <math.h>.7477 **CHANGE HISTORY**

7478 First released in Issue 1.

7479 Derived from Issue 1 of the SVID.

7480 **Issue 4**

7481 The following changes are incorporated in this issue:

- 7482 • Removed references to *matherr()*.

7483			
7484		• The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the ISO C standard and to rationalise handling in the mathematics functions.	
7485		• The word long has been replaced with the words long int in the APPLICATION USAGE	
7486		section.	
7487		• The return value specified for [EDOM] is marked as an extension.	
7488	Issue 5		
7489		The DESCRIPTION is updated to indicate how an application should check for an error. This	
7490		text was previously published in the APPLICATION USAGE section.	

7491 **NAME**

7492 fmod — floating-point remainder value function

7493 **SYNOPSIS**

7494 #include <math.h>

7495 double fmod(double x, double y);

7496 **DESCRIPTION**7497 The *fmod()* function returns the floating-point remainder of the division of *x* by *y*.7498 An application wishing to check for error situations should set *errno* to 0 before calling *fmod()*. If
7499 *errno* is non-zero on return, or the return value is NaN, an error has occurred.7500 **RETURN VALUE**7501 The *fmod()* function returns the value $x - i * y$, for some integer *i* such that, if *y* is non-zero, the
7502 result has the same sign as *x* and magnitude less than the magnitude of *y*.7503 EX If *x* or *y* is NaN, NaN is returned and *errno* may be set to [EDOM].7504 EX If *y* is 0, NaN is returned and *errno* is set to [EDOM], or 0 is returned and *errno* may be set to
7505 [EDOM].7506 EX If *x* is $\pm\text{Inf}$, either 0 is returned and *errno* is set to [EDOM], or NaN is returned and *errno* may be
7507 set to [EDOM].7508 If *y* is non-zero, *fmod*($\pm 0, y$) returns the value of *x*. If *x* is not $\pm\text{Inf}$, *fmod*(*x*, $\pm\text{Inf}$) returns the value
7509 of *x*.7510 If the result underflows, 0 is returned and *errno* may be set to [ERANGE].7511 **ERRORS**7512 The *fmod()* function may fail if:7513 EX [EDOM] One or both of the arguments is NaN, or *y* is 0, or *x* is $\pm\text{Inf}$.

7514 [ERANGE] The result underflows.

7515 EX No other errors will occur.

7516 **EXAMPLES**

7517 None.

7518 **APPLICATION USAGE**7519 Portable applications should not call *fmod()* with *y* equal to 0, because the result is
7520 implementation-dependent. The application should verify *y* is non-zero before calling *fmod()*.7521 **FUTURE DIRECTIONS**

7522 None.

7523 **SEE ALSO**

7524 isnan(), <math.h>.

7525 **CHANGE HISTORY**

7526 First released in Issue 1.

7527 Derived from Issue 1 of the SVID.

7528 **Issue 4**

7529 The following changes are incorporated in this issue:

- 7530 • References to
- matherr()*
- are removed.

- 7531 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with |
- 7532 the ISO C standard and to rationalise error handling in the mathematics functions.
- 7533 • The return value specified for [EDOM] is marked as an extension.
- 7534 **Issue 5** |
- 7535 The DESCRIPTION is updated to indicate how an application should check for an error. This
- 7536 text was previously published in the APPLICATION USAGE section.

7537 **NAME**

7538 fmtmsg — display a message in the specified format on standard error and/or a system console

7539 **SYNOPSIS**

7540 EX #include <fmtmsg.h>

7541 int fmtmsg(long *classification*, const char **label*, int *severity*,
7542 const char **text*, const char **action*, const char **tag*);

7543

7544 **DESCRIPTION**7545 The *fmtmsg()* function can be used to display messages in a specified format instead of the
7546 traditional *printf()* function.7547 Based on a message's classification component, *fmtmsg()* writes a formatted message either to
7548 standard error, to the console, or to both.7549 A formatted message consists of up to five components as defined below. The component
7550 *classification* is not part of a message displayed to the user, but defines the source of the message
7551 and directs the display of the formatted message.7552 *classification* Contains identifiers from the following groups of major classifications and
7553 subclassifications. Any one identifier from a subclass may be used in
7554 combination with a single identifier from a different subclass. Two or more
7555 identifiers from the same subclass should not be used together, with the
7556 exception of identifiers from the display subclass. (Both display subclass
7557 identifiers may be used so that messages can be displayed to both standard
7558 error and the system console).7559 **Major Classifications**7560 Identifies the source of the condition. Identifiers are: MM_HARD
7561 (hardware), MM_SOFT (software), and MM_FIRM (firmware).7562 **Message Source Subclassifications**7563 Identifies the type of software in which the problem is detected.
7564 Identifiers are: MM_APPL (application), MM_UTIL (utility), and
7565 MM_OPSYS (operating system).7566 **Display Subclassifications**7567 Indicates where the message is to be displayed. Identifiers are:
7568 MM_PRINT to display the message on the standard error stream,
7569 MM_CONSOLE to display the message on the system console. One or
7570 both identifiers may be used.7571 **Status Subclassifications**7572 Indicates whether the application will recover from the condition.
7573 Identifiers are: MM_RECOVER (recoverable) and MM_NRECOV (non-
7574 recoverable).7575 An additional identifier, MM_NULLMC, indicates that no classification
7576 component is supplied for the message.7577 *label* Identifies the source of the message. The format is two fields separated by a
7578 colon. The first field is up to 10 bytes, the second is up to 14 bytes.7579 *severity* Indicates the seriousness of the condition. Identifiers for the levels of *severity*
7580 are:

7581		MM_HALT	Indicates that the application has encountered a severe fault and is halting. Produces the string "HALT".
7582			
7583		MM_ERROR	Indicates that the application has detected a fault. Produces the string "ERROR".
7584			
7585		MM_WARNING	Indicates a condition that is out of the ordinary, that might be a problem, and should be watched. Produces the string "WARNING".
7586			
7587			
7588		MM_INFO	Provides information about a condition that is not in error. Produces the string "INFO".
7589			
7590		MM_NOSEV	Indicates that no severity level is supplied for the message.
7591	<i>text</i>		Describes the error condition that produced the message. The character string is not limited to a specific size. If the character string is empty, then the text produced is unspecified.
7592			
7593			
7594	<i>action</i>		Describes the first step to be taken in the error-recovery process. The <i>fmtmsg()</i> function precedes the action string with the prefix: "TO FIX:". The <i>action</i> string is not limited to a specific size.
7595			
7596			
7597	<i>tag</i>		An identifier that references on-line documentation for the message. Suggested usage is that <i>tag</i> includes the <i>label</i> and a unique identifying number. A sample <i>tag</i> is "XSI:cat:146".
7598			
7599			

7600 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.

7610 *MSGVERB* affects only which components are selected for display to standard error. All message components are included in console messages.

7612 RETURN VALUE

7613 The *fmtmsg()* function returns one of the following values:

7614	MM_OK	The function succeeded.
7615	MM_NOTOK	The function failed completely.
7616	MM_NOMSG	The function was unable to generate a message on standard error, but otherwise succeeded.
7617		
7618	MM_NOCON	The function was unable to generate a console message, but otherwise succeeded.
7619		

7620 ERRORS

7621 None.

7622 **EXAMPLES**

7623 Example 1:

7624 The following example of *fmtmsg()*:

```
7625       fmtmsg(MM_PRINT, "XSI:cat", MM_ERROR, "illegal option",
7626       "refer to cat in user's reference manual", "XSI:cat:001")
```

7627 produces a complete message in the specified message format:

```
7628       XSI:cat: ERROR: illegal option
7629       TO FIX: refer to cat in user's reference manual XSI:cat:001
```

7630 Example 2:

7631 When the environment variable *MSGVERB* is set as follows:

7632 MSGVERB=severity:text:action

7633 and the Example 1 is used, *fmtmsg()* produces:

```
7634       ERROR: illegal option
7635       TO FIX: refer to cat in user's reference manual
```

7636 **APPLICATION USAGE**

7637 One or more message components may be systematically omitted from messages generated by
 7638 an application by using the null value of the argument for that component.

7639 **FUTURE DIRECTIONS**

7640 None.

7641 **SEE ALSO**7642 *printf()*, <fmtmsg.h>.7643 **CHANGE HISTORY**

7644 First released in Issue 4, Version 2.

7645 **Issue 5**

7646 Moved from X/OPEN UNIX extension to BASE.

7647 **NAME**

7648 fnmatch — match a filename or a pathname

7649 **SYNOPSIS**

7650 #include <fnmatch.h>

7651 int fnmatch(const char **pattern*, const char **string*, int *flags*);7652 **DESCRIPTION**

7653 The *fnmatch()* function matches patterns as described in the XCU specification, **Section 2.13.1, Patterns Matching a Single Character**, and **Section 2.13.2, Patterns Matching Multiple Characters**. It checks the string specified by the *string* argument to see if it matches the pattern specified by the *pattern* argument.

7657 The *flags* argument modifies the interpretation of *pattern* and *string*. It is the bitwise inclusive OR of zero or more of the flags defined in the header <fnmatch.h>. If the FNM_PATHNAME flag is set in *flags*, then a slash character in *string* will be explicitly matched by a slash in *pattern*; it will not be matched by either the asterisk or question-mark special characters, nor by a bracket expression. If the FNM_PATHNAME flag is not set, the slash character is treated as an ordinary character.

7663 If FNM_NOESCAPE is not set in *flags*, a backslash character (\) in *pattern* followed by any other character will match that second character in *string*. In particular, \\ will match a backslash in *string*. If FNM_NOESCAPE is set, a backslash character will be treated as an ordinary character.

7666 If FNM_PERIOD is set in *flags*, then a leading period in *string* will match a period in *pattern*; as described by rule 2 in the XCU specification, **Section 2.13.3, Patterns Used for Filename Expansion** where the location of “leading” is indicated by the value of FNM_PATHNAME:

- 7669 • If FNM_PATHNAME is set, a period is “leading” if it is the first character in *string* or if it immediately follows a slash.
- 7671 • If FNM_PATHNAME is not set, a period is “leading” only if it is the first character of *string*.

7672 If FNM_PERIOD is not set, then no special restrictions are placed on matching a period.

7673 **RETURN VALUE**

7674 If *string* matches the pattern specified by *pattern*, then *fnmatch()* returns 0. If there is no match, *fnmatch()* returns FNM_NOMATCH, which is defined in the header <fnmatch.h>. If an error occurs, *fnmatch()* returns another non-zero value.

7677 **ERRORS**

7678 No errors are defined.

7679 **EXAMPLES**

7680 None.

7681 **APPLICATION USAGE**

7682 The *fnmatch()* function has two major uses. It could be used by an application or utility that needs to read a directory and apply a pattern against each entry. The *find* utility is an example of this. It can also be used by the *pax* utility to process its *pattern* operands, or by applications that need to match strings in a similar manner.

7686 The name *fnmatch()* is intended to imply *filename* match, rather than *pathname* match. The default action of this function is to match filenames, rather than pathnames, since it gives no special significance to the slash character. With the FNM_PATHNAME flag, *fnmatch()* does match pathnames, but without tilde expansion, parameter expansion, or special treatment for period at the beginning of a filename.

7691 **FUTURE DIRECTIONS**

7692 None.

7693 **SEE ALSO**7694 *glob()*, *wordexp()*, *<fnmatch.h>*, the XCU specification.7695 **CHANGE HISTORY**

7696 First released in Issue 4.

7697 Derived from the ISO POSIX-2 standard.

7698 **Issue 5**

7699 Moved from POSIX2 C-language Binding to BASE.

7700 NAME

7701 fopen — open a stream

7702 SYNOPSIS

7703 #include <stdio.h>

7704 FILE *fopen(const char *filename, const char *mode);

7705 DESCRIPTION

7706 The *fopen()* function opens the file whose pathname is the string pointed to by *filename*, and
7707 associates a stream with it.7708 The argument *mode* points to a string beginning with one of the following sequences:7709 **r** or **rb** Open file for reading.7710 **w** or **wb** Truncate to zero length or create file for writing.7711 **a** or **ab** Append; open or create file for writing at end-of-file.7712 **r+** or **rb+** or **r+b** Open file for update (reading and writing).7713 **w+** or **wb+** or **w+b** Truncate to zero length or create file for update.7714 **a+** or **ab+** or **a+b** Append; open or create file for update, writing at end-of-file.7715 The character *b* has no effect, but is allowed for ISO C standard conformance. Opening a file
7716 with read mode (*r* as the first character in the *mode* argument) fails if the file does not exist or
7717 cannot be read.7718 Opening a file with append mode (*a* as the first character in the *mode* argument) causes all
7719 subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening
7720 calls to *fseek()*.7721 When a file is opened with update mode (*+* as the second or third character in the *mode*
7722 argument), both input and output may be performed on the associated stream. However,
7723 output must not be directly followed by input without an intervening call to *fflush()* or to a file
7724 positioning function (*fseek()*, *fsetpos()* or *rewind()*), and input must not be directly followed by
7725 output without an intervening call to a file positioning function, unless the input operation
7726 encounters end-of-file.7727 When opened, a stream is fully buffered if and only if it can be determined not to refer to an
7728 interactive device. The error and end-of-file indicators for the stream are cleared.7729 If *mode* is **w**, **a**, **w+** or **a+** and the file did not previously exist, upon successful completion,
7730 *fopen()* function will mark for update the *st_atime*, *st_ctime* and *st_mtime* fields of the file and the
7731 *st_ctime* and *st_mtime* fields of the parent directory.7732 If *mode* is **w** or **w+** and the file did previously exist, upon successful completion, *fopen()* will
7733 mark for update the *st_ctime* and *st_mtime* fields of the file. The *fopen()* function will allocate a
7734 file descriptor as *open()* does.7735 EX The largest value that can be represented correctly in an object of type **off_t** will be established
7736 as the offset maximum in the open file description.

7737 RETURN VALUE

7738 Upon successful completion, *fopen()* returns a pointer to the object controlling the stream.
7739 Otherwise, a null pointer is returned, and *errno* is set to indicate the error.

7740 **ERRORS**7741 The *fopen()* function will fail if:

7742 [EACCES] Search permission is denied on a component of the path prefix, or the file
 7743 exists and the permissions specified by *mode* are denied, or the file does not
 7744 exist and write permission is denied for the parent directory of the file to be
 7745 created.

7746 [EINTR] A signal was caught during *fopen()*.7747 [EISDIR] The named file is a directory and *mode* requires write access.7748 EX [ELOOP] Too many symbolic links were encountered in resolving *path*.

7749 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

7750 FIPS [ENAMETOOLONG]

7751 The length of the *filename* exceeds {PATH_MAX} or a pathname component is
 7752 longer than {NAME_MAX}.

7753 [ENFILE] The maximum allowable number of files is currently open in the system.

7754 [ENOENT] A component of *filename* does not name an existing file or *filename* is an empty
 7755 string.

7756 [ENOSPC] The directory or file system that would contain the new file cannot be
 7757 expanded, the file does not exist, and it was to be created.

7758 [ENOTDIR] A component of the path prefix is not a directory.

7759 [ENXIO] The named file is a character special or block special file, and the device
 7760 associated with this special file does not exist.

7761 EX [EOVERFLOW] The named file is a regular file and the size of the file cannot be represented
 7762 correctly in an object of type *off_t*.

7763 [EROFS] The named file resides on a read-only file system and *mode* requires write
 7764 access.

7765 The *fopen()* function may fail if:7766 EX [EINVAL] The value of the *mode* argument is not valid.

7767 EX [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

7768 EX [EMFILE] {STREAM_MAX} streams are currently open in the calling process.

7769 EX [ENAMETOOLONG]

7770 Pathname resolution of a symbolic link produced an intermediate result
 7771 whose length exceeds {PATH_MAX}.

7772 [ENOMEM] Insufficient storage space is available.

7773 [ETXTBSY] The file is a pure procedure (shared text) file that is being executed and *mode*
 7774 requires write access.

7775 **EXAMPLES**

7776 None.

7777 **APPLICATION USAGE**

7778 None.

7779 **FUTURE DIRECTIONS**

7780 None.

7781 **SEE ALSO**7782 *fclose()*, *fdopen()*, *freopen()*, *<stdio.h>*.7783 **CHANGE HISTORY**

7784 First released in Issue 1.

7785 Derived from Issue 1 of the SVID.

7786 **Issue 4**

7787 The following changes are incorporated for alignment with the ISO C standard:

- 7788 • The type of arguments *filename* and *mode* are changed from **char *** to **const char ***.
- 7789 • In the DESCRIPTION, (a) the use and settings of the *mode* argument are changed to support
- 7790 binary streams and (b) *setpos()* is added to the list of file positioning functions.

7791 The following change is incorporated for alignment with the FIPS requirements:

- 7792 • In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a
- 7793 pathname component is larger than {NAME_MAX} is now defined as mandatory and marked
- 7794 as an extension.

7795 Other changes are incorporated as follows:

- 7796 • In the DESCRIPTION the descriptions of input and output operations on update streams are
- 7797 changed to be requirements on the application.
- 7798 • The [EMFILE] error is added to the ERRORS section, and all the optional errors are marked
- 7799 as extensions.

7800 **Issue 4, Version 2**

7801 The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- 7802 • It states that [ELOOP] will be returned if too many symbolic links are encountered during
- 7803 pathname resolution.
- 7804 • A second [ENAMETOOLONG] condition is defined that may report excessive length of an
- 7805 intermediate result of pathname resolution of a symbolic link.

7806 **Issue 5**

7807 Large File Summit extensions added.

7808 **NAME**

7809 fork — create a new process

7810 **SYNOPSIS**

7811 OH #include <sys/types.h>

7812 #include <unistd.h>

7813 pid_t fork(void);

7814 **DESCRIPTION**

7815 The *fork()* function creates a new process. The new process (child process) is an exact copy of
 7816 the calling process (parent process) except as detailed below.

- 7817 • The child process has a unique process ID.
- 7818 • The child process ID also does not match any active process group ID.
- 7819 • The child process has a different parent process ID (that is, the process ID of the parent
 7820 process).
- 7821 • The child process has its own copy of the parent's file descriptors. Each of the child's file
 7822 descriptors refers to the same open file description with the corresponding file descriptor of
 7823 the parent.
- 7824 • The child process has its own copy of the parent's open directory streams. Each open
 7825 directory stream in the child process may share directory stream positioning with the
 7826 corresponding directory stream of the parent.
- 7827 EX • The child process may have its own copy of the parent's message catalogue descriptors.
- 7828 • The child process' values of *tms_utime*, *tms_stime*, *tms_cutime* and *tms_cstime* are set to 0.
- 7829 • The time left until an alarm clock signal is reset to 0.
- 7830 EX • All *semadj* values are cleared.
- 7831 • File locks set by the parent process are not inherited by the child process.
- 7832 • The set of signals pending for the child process is initialised to the empty set.
- 7833 EX • Interval timers are reset in the child process.
- 7834 RT • If the Semaphores option is supported, any semaphores that are open in the parent process
 7835 will also be open in the child process.
- 7836 • If the Process Memory Locking option is supported, the child process does not inherit any
 7837 address space memory locks established by the parent process via calls to *mlockall()* or
 7838 *mlock()*.
- 7839 • Memory mappings created in the parent are retained in the child process. MAP_PRIVATE
 7840 mappings inherited from the parent will also be MAP_PRIVATE mappings in the child, and
 7841 any modifications to the data in these mappings made by the parent prior to calling *fork()*
 7842 will be visible to the child. Any modifications to the data in MAP_PRIVATE mappings made
 7843 by the parent after *fork()* returns will be visible only to the parent. Modifications to the data
 7844 in MAP_PRIVATE mappings made by the child will be visible only to the child.
- 7845 RT • If the Process Scheduling option is supported, for the SCHED_FIFO and SCHED_RR
 7846 scheduling policies, the child process inherits the policy and priority settings of the parent
 7847 process during a *fork()* function. For other scheduling policies, the policy and priority
 7848 settings on *fork()* are implementation-dependent.

7849	<ul style="list-style-type: none"> • If the Timers option is supported, per-process timers created by the parent are not inherited by the child process. 	
7850		
7851	<ul style="list-style-type: none"> • If the Message Passing option is supported, the child process has its own copy of the message queue descriptors of the parent. Each of the message descriptors of the child refers to the same open message queue description as the corresponding message descriptor of the parent. 	
7852		
7853		
7854		
7855	<ul style="list-style-type: none"> • If the Asynchronous Input and Output option is supported, no asynchronous input or asynchronous output operations are inherited by the child process. 	
7856		
7857	The inheritance of process characteristics not defined by this document is implementation-dependent. After <i>fork()</i> , both the parent and the child processes are capable of executing independently before either one terminates.	
7858		
7859		
7860	A process is created with a single thread. If a multi-threaded process calls <i>fork()</i> , 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 <i>exec</i> functions is called. Fork handlers may be established by means of the <i>pthread_atfork()</i> function in order to maintain application invariants across <i>fork()</i> calls.	
7861		
7862		
7863		
7864		
7865		
7866	RETURN VALUE	
7867	Upon successful completion, <i>fork()</i> returns 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, -1 is returned to the parent process, no child process is created, and <i>errno</i> is set to indicate the error.	
7868		
7869		
7870	ERRORS	
7871	The <i>fork()</i> function will fail if:	
7872	[EAGAIN]	The system lacked the necessary resources to create another process, or the system-imposed limit on the total number of processes under execution system-wide or by a single user {CHILD_MAX} would be exceeded.
7873		
7874		
7875	The <i>fork()</i> function may fail if:	
7876	[ENOMEM]	Insufficient storage space is available.
7877	EXAMPLES	
7878	None.	
7879	APPLICATION USAGE	
7880	None.	
7881	FUTURE DIRECTIONS	
7882	None.	
7883	SEE ALSO	
7884	<i>alarm()</i> , <i>exec</i> , <i>fcntl()</i> , <i>semop()</i> , <i>signal()</i> , <i>times()</i> , <sys/types.h>, <unistd.h>.	
7885	CHANGE HISTORY	
7886	First released in Issue 1.	
7887	Derived from Issue 1 of the SVID.	
7888	Issue 4	
7889	The following changes are incorporated for alignment with the ISO POSIX-1 standard:	
7890	<ul style="list-style-type: none"> • The argument list is explicitly defined as void. 	

- 7891 • Though functionally identical to Issue 3, the DESCRIPTION has been reorganised to improve
7892 clarity and to align more closely with the ISO POSIX-1 standard.
- 7893 • The description of the [EAGAIN] error is updated to indicate that this error can also be
7894 returned if a system lacks the resources to create another process.
- 7895 Another change is incorporated as follows:
- 7896 • The `<sys/types.h>` header is now marked as optional (OH); this header need not be included
7897 on XSI-conformant systems.
- 7898 **Issue 4, Version 2**
- 7899 The DESCRIPTION is changed for X/OPEN UNIX conformance to identify that interval timers
7900 are reset in the child process.
- 7901 **Issue 5**
- 7902 The DESCRIPTION is changed for alignment with the POSIX Realtime Extension and the POSIX
7903 Threads Extension.

7904 **NAME**

7905 fpathconf, pathconf — get configurable pathname variables

7906 **SYNOPSIS**

7907 #include <unistd.h>

7908 long int fpathconf(int *fildes*, int *name*);7909 long int pathconf(const char **path*, int *name*);7910 **DESCRIPTION**7911 The *fpathconf()* and *pathconf()* functions provide a method for the application to determine the
7912 current value of a configurable limit or option (*variable*) that is associated with a file or directory.7913 For *pathconf()*, the *path* argument points to the pathname of a file or directory.7914 For *fpathconf()*, the *fildes* argument is an open file descriptor.7915 The *name* argument represents the variable to be queried relative to that file or directory.
7916 Implementations will support all of the variables listed in the following table and may support
7917 others. The variables in the following table come from <limits.h> or <unistd.h> and the
7918 symbolic constants, defined in <unistd.h>, are the corresponding values used for *name*:

7919

7920

7921 EX

7922

7923

7924

7925

7926

7927

7928

7929

7930

7931

7932

7933

Variable	Value of <i>name</i>	Notes
FILESIZEBITS	_PC_FILESIZEBITS	3, 4
LINK_MAX	_PC_LINK_MAX	1
MAX_CANON	_PC_MAX_CANON	2
MAX_INPUT	_PC_MAX_INPUT	2
NAME_MAX	_PC_NAME_MAX	3, 4
PATH_MAX	_PC_PATH_MAX	4, 5
PIPE_BUF	_PC_PIPE_BUF	6
_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED	7
_POSIX_NO_TRUNC	_PC_NO_TRUNC	3, 4
_POSIX_VDISABLE	_PC_VDISABLE	2
_POSIX_ASYNC_IO	_PC_ASYNC_IO	8
_POSIX_PRIO_IO	_PC_PRIO_IO	8
_POSIX_SYNC_IO	_PC_SYNC_IO	8

7934 **Notes:**

7935

7936

7937

7938

7939

7940

7941

7942

7943

7944

7945

7946

7947

7948

1. If *path* or *fildes* refers to a directory, the value returned applies to the directory itself.
2. If *path* or *fildes* does not refer to a terminal file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
3. If *path* or *fildes* refers to a directory, the value returned applies to filenames 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 pathname 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

7949 applies to any FIFO that exists or can be created within the directory. If *path* or
 7950 *filde*s refers to any other type of file, it is unspecified whether an implementation
 7951 supports an association of the variable name with the specified file.

7952 7. If *path* or *filde*s refers to a directory, the value returned applies to any files, other
 7953 than directories, that exist or can be created within the directory.

7954 8. If *path* or *filde*s refers to a directory, it is unspecified whether an implementation
 7955 supports an association of the variable name with the specified file.

7956 RETURN VALUE

7957 If *name* is an invalid value, both *pathconf()* and *fpathconf()* return `-1` and *errno* is set to indicate
 7958 the error.

7959 If the variable corresponding to *name* has no limit for the *path* or file descriptor, both *pathconf()*
 7960 and *fpathconf()* return `-1` without changing *errno*. If the implementation needs to use *path* to
 7961 determine the value of *name* and the implementation does not support the association of *name*
 7962 with the file specified by *path*, or if the process did not have appropriate privileges to query the
 7963 file specified by *path*, or *path* does not exist, *pathconf()* returns `-1` and *errno* is set to indicate the
 7964 error.

7965 If the implementation needs to use *filde*s to determine the value of *name* and the implementation
 7966 does not support the association of *name* with the file specified by *filde*s, or if *filde*s is an invalid
 7967 file descriptor, *fpathconf()* will return `-1` and *errno* is set to indicate the error.

7968 Otherwise *pathconf()* or *fpathconf()* returns the current variable value for the file or directory
 7969 without changing *errno*. The value returned will not be more restrictive than the corresponding
 7970 value available to the application when it was compiled with the implementation's `<limits.h>` or
 7971 `<unistd.h>`.

7972 ERRORS

7973 The *pathconf()* function will fail if:

7974 [EINVAL] The value of *name* is not valid.

7975 EX [ELOOP] Too many symbolic links were encountered in resolving *path*.

7976 The *pathconf()* function may fail if:

7977 [EACCES] Search permission is denied for a component of the path prefix.

7978 [EINVAL] The implementation does not support an association of the variable *name* with
 7979 the specified file.

7980 [ENAMETOOLONG]

7981 FIPS The length of the *path* argument exceeds {PATH_MAX} or a pathname
 7982 component is longer than {NAME_MAX}.

7983 [ENAMETOOLONG]

7984 EX Pathname resolution of a symbolic link produced an intermediate result
 7985 whose length exceeds {PATH_MAX}.

7986 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

7987 [ENOTDIR] A component of the path prefix is not a directory.

7988	The <i>fpathconf()</i> function will fail if:	
7989	[EINVAL] The value of <i>name</i> is not valid.	
7990	The <i>fpathconf()</i> function may fail if:	
7991	[EBADF] The <i>fildev</i> argument is not a valid file descriptor.	
7992	[EINVAL] The implementation does not support an association of the variable <i>name</i> with	
7993	the specified file.	
7994	EXAMPLES	
7995	None.	
7996	APPLICATION USAGE	
7997	None.	
7998	FUTURE DIRECTIONS	
7999	None.	
8000	SEE ALSO	
8001	<i>confstr()</i> , <i>sysconf()</i> , <i><limits.h></i> , <i><unistd.h></i> , the XCU specification of <i>getconf</i> .	
8002	CHANGE HISTORY	
8003	First released in Issue 3.	
8004	Entry included for alignment with the POSIX.1-1988 standard.	
8005	Issue 4	
8006	The <i>fpathconf()</i> function now has the full long int return type in the SYNOPSIS section.	
8007	The following changes have been made for alignment with the ISO POSIX-1 standard:	
8008	• The type of argument <i>path</i> is changed from char * to const char * . Also the return value of	
8009	both functions is changed from long to long int .	
8010	• In the DESCRIPTION, the words “The behaviour is undefined if” have been replaced by “it	
8011	is unspecified whether an implementation supports an association of the variable name with	
8012	the specified file” in notes 2, 4 and 6.	
8013	• In the RETURN VALUE section, errors associated with the use of <i>path</i> and <i>fildev</i> , when an	
8014	implementation does not support the requested association, are now specified separately.	
8015	• The requirement that <i>errno</i> be set to indicate the error is added.	
8016	The following change is incorporated for alignment with the FIPS requirements:	
8017	• In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a	
8018	pathname component is larger than {NAME_MAX} is now defined as mandatory and marked	
8019	as an extension.	
8020	Issue 4, Version 2	
8021	The ERRORS section is updated for X/OPEN UNIX conformance as follows:	
8022	• It states that [ELOOP] will be returned if too many symbolic links are encountered during	
8023	pathname resolution.	
8024	• A second [ENAMETOOLONG] condition is defined that may report excessive length of an	
8025	intermediate result of pathname resolution of a symbolic link.	

8026 **Issue 5**

8027 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

8028 Large File Summit extensions added.

8029 NAME

8030 fprintf, printf, snprintf, sprintf — print formatted output

8031 SYNOPSIS

8032 #include <stdio.h>

8033 int fprintf(FILE *stream, const char *format, ...);

8034 int printf(const char *format, ...);

8035 EX int snprintf(char *s, size_t n, const char *format, ...);

8036 int sprintf(char *s, const char *format, ...);

8037 DESCRIPTION

8038 The *fprintf()* function places output on the named output *stream*. The *printf()* function places
 8039 output on the standard output stream *stdout*. The *sprintf()* function places output followed by
 8040 the null byte, '\0', in consecutive bytes starting at *s*; it is the user's responsibility to ensure that
 8041 enough space is available.

8042 EX *snprintf()* is identical to *sprintf()* with the addition of the *n* argument, which states the size of the
 8043 buffer referred to by *s*.

8044 Each of these functions converts, formats and prints its arguments under control of the *format*.
 8045 The *format* is a character string, beginning and ending in its initial shift state, if any. The *format* is
 8046 composed of zero or more directives: *ordinary characters*, which are simply copied to the output
 8047 stream and *conversion specifications*, each of which results in the fetching of zero or more
 8048 arguments. The results are undefined if there are insufficient arguments for the *format*. If the
 8049 *format* is exhausted while arguments remain, the excess arguments are evaluated but are
 8050 otherwise ignored.

8051 EX Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
 8052 to the next unused argument. In this case, the conversion character % (see below) is replaced by
 8053 the sequence %*n*%, where *n* is a decimal integer in the range [1, {NL_ARGMAX}], giving the
 8054 position of the argument in the argument list. This feature provides for the definition of format
 8055 strings that select arguments in an order appropriate to specific languages (see the EXAMPLES
 8056 section).

8057 In format strings containing the %*n*% form of conversion specifications, numbered arguments in
 8058 the argument list can be referenced from the format string as many times as required.

8059 In format strings containing the % form of conversion specifications, each argument in the
 8060 argument list is used exactly once.

8061 All forms of the *fprintf()* functions allow for the insertion of a language-dependent radix
 8062 character in the output string. The radix character is defined in the program's locale (category
 8063 LC_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the
 8064 radix character defaults to a period (.).

8065 EX Each conversion specification is introduced by the % character or by the character sequence %*n*%,
 8066 after which the following appear in sequence:

- 8067 • Zero or more *flags* (in any order), which modify the meaning of the conversion specification.
- 8068 • An optional minimum *field width*. If the converted value has fewer bytes than the field
 8069 width, it will be padded with spaces by default on the left; it will be padded on the right, if
 8070 the left-adjustment flag (-), described below, is given to the field width. The field width
 8071 takes the form of an asterisk (*), described below, or a decimal integer.
- 8072 • An optional *precision* that gives the minimum number of digits to appear for the d, i, o, u, x
 8073 and X conversions; the number of digits to appear after the radix character for the e, E and f
 8074 conversions; the maximum number of significant digits for the g and G conversions; or the

8075 EX maximum number of bytes to be printed from a string in `s` and `S` conversions. The precision
 8076 takes the form of a period (.) followed either by an asterisk (*), described below, or an
 8077 optional decimal digit string, where a null digit string is treated as 0. If a precision appears
 8078 with any other conversion character, the behaviour is undefined.

8079 • An optional `h` specifying that a following `d`, `i`, `o`, `u`, `x` or `X` conversion character applies to a
 8080 type **short int** or type **unsigned short int** argument (the argument will have been promoted
 8081 according to the integral promotions, and its value will be converted to type **short int** or
 8082 **unsigned short int** before printing); an optional `h` specifying that a following `n` conversion
 8083 character applies to a pointer to a type **short int** argument; an optional `l` (ell) specifying that a
 8084 following `d`, `i`, `o`, `u`, `x` or `X` conversion character applies to a type **long int** or **unsigned long**
 8085 **int** argument; an optional `l` (ell) specifying that a following `n` conversion character applies to
 8086 a pointer to a type **long int** argument; or an optional `L` specifying that a following `e`, `E`, `f`, `g` or
 8087 `G` conversion character applies to a type **long double** argument. If an `h`, `l` or `L` appears with
 8088 any other conversion character, the behaviour is undefined.

8089 • An optional `l` specifying that a following `c` conversion character applies to a **wint_t**
 8090 argument; an optional `l` specifying that a following `s` conversion character applies to a
 8091 pointer to a **wchar_t** argument.

8092 • A *conversion character* that indicates the type of conversion to be applied.

8093 A field width, or precision, or both, may be indicated by an asterisk (*). In this case an argument
 8094 of type **int** supplies the field width or precision. Arguments specifying field width, or precision,
 8095 or both must appear in that order before the argument, if any, to be converted. A negative field
 8096 width is taken as a `-` flag followed by a positive field width. A negative precision is taken as if
 8097 EX the precision were omitted. In format strings containing the `%n$` form of a conversion
 8098 specification, a field width or precision may be indicated by the sequence `*m$`, where `m` is a
 8099 decimal integer in the range `[1, {NL_ARGMAX}]` giving the position in the argument list (after
 8100 the format argument) of an integer argument containing the field width or precision, for
 8101 example:

```
8102 printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

8103 The *format* can contain either numbered argument specifications (that is, `%n$` and `*m$`), or
 8104 unnumbered argument specifications (that is, `%` and `*`), but normally not both. The only
 8105 exception to this is that `%%` can be mixed with the `%n$` form. The results of mixing numbered
 8106 and unnumbered argument specifications in a *format* string are undefined. When numbered
 8107 argument specifications are used, specifying the *N*th argument requires that all the leading
 8108 arguments, from the first to the $(N-1)$ th, are specified in the format string.

8109 The flag characters and their meanings are:

8110 EX • The integer portion of the result of a decimal conversion (`%i`, `%d`, `%u`, `%f`, `%g` or `%G`)
 8111 will be formatted with thousands' grouping characters. For other conversions the
 8112 behaviour is undefined. The non-monetary grouping character is used.

8113 – The result of the conversion will be left-justified within the field. The conversion will
 8114 be right-justified if this flag is not specified.

8115 + The result of a signed conversion will always begin with a sign (`+` or `-`). The conversion
 8116 will begin with a sign only when a negative value is converted if this flag is not
 8117 specified.

8118	space	If the first character of a signed conversion is not a sign or if a signed conversion results in no characters, a space will be prefixed to the result. This means that if the space and + flags both appear, the space flag will be ignored.
8119		
8120		
8121	#	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 will have 0x (or 0X) prefixed to it. For e, E, f, g or G conversions, the result will always contain a radix character, even if no digits follow the radix character. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For g and G conversions, trailing zeros will <i>not</i> be removed from the result as they normally are. For other conversions, the behaviour is undefined.
8122		
8123		
8124		
8125		
8126		
8127		
8128		
8129	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 will be ignored. For d, i, o, u, x and X conversions, if a precision is specified, the 0 flag will be ignored. If the 0 and ' flags both appear, the grouping characters are inserted before zero padding. For other conversions, the behaviour is undefined.
8130		
8131		
8132		
8133		
8134		
8135	The conversion characters and their meanings are:	
8136	d, i	The int argument is converted to a signed decimal in the style <code>[-]dddd</code> . The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.
8137		
8138		
8139		
8140	o	The unsigned int argument is converted to unsigned octal format in the style <code>dddd</code> . The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.
8141		
8142		
8143		
8144		
8145	u	The unsigned int argument is converted to unsigned decimal format in the style <code>dddd</code> . The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.
8146		
8147		
8148		
8149		
8150	x	The unsigned int argument is converted to unsigned hexadecimal format in the style <code>dddd</code> ; 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 will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.
8151		
8152		
8153		
8154		
8155	X	Behaves the same as the x conversion character except that letters ABCDEF are used instead of abcdef.
8156		
8157	f	The double argument is converted to decimal notation in the style <code>[-]ddd.ddd</code> , 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.
8158		
8159		
8160		
8161		
8162	EX	The <i>fprintf()</i> family of functions may make available character string representations
8163		for infinity and NaN.

8164	e, E	The double argument is converted in the style <code>[-]d.ddde±dd</code> , 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 character will 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.
8171	EX	The <i>fprintf()</i> family of functions may make available character string representations for infinity and NaN.
8172		
8173	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 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) will 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.
8174		
8175		
8176		
8177		
8178		
8179		
8180	EX	The <i>fprintf()</i> family of functions may make available character string representations for infinity and NaN.
8181		
8182	c	The int argument is converted to an unsigned char , and the resulting byte is written.
8183		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.
8184		
8185		
8186		
8187		
8188	s	The argument must 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 are written. If the precision is not specified or is greater than the size of the array, the array must contain a null byte.
8189		
8190		
8191		
8192		If an l (ell) qualifier is present, the argument must 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 <i>wcrtomb()</i> function, with the conversion state described by an mbstate_t object initialised to zero before the first wide-character is converted) up to and including a terminating null wide-character. The resulting characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array must contain a null wide-character. If a precision is specified, no more than that many characters (bytes) are written (including shift sequences, if any), and the array must 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.
8193		
8194		
8195		
8196		
8197		
8198		
8199		
8200		
8201		
8202		
8203	p	The argument must be a pointer to void . The value of the pointer is converted to a sequence of printable characters, in an implementation-dependent manner.
8204		
8205	n	The argument must 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 <i>fprintf()</i> functions. No argument is converted.
8206		
8207		

8208	EX	C	Same as <code>lc</code> .	
8209	EX	S	Same as <code>ls</code> .	
8210		%	Print a %; no argument is converted. The entire conversion specification must be <code>%%</code> .	
8211			If a conversion specification does not match one of the above forms, the behaviour is undefined.	
8212			In no case does a non-existent 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 <code>fprintf()</code> and <code>printf()</code> are printed as if <code>fputc()</code> had been called.	
8213				
8214				
8215			The <code>st_ctime</code> and <code>st_mtime</code> fields of the file will be marked for update between the call to a successful execution of <code>fprintf()</code> or <code>printf()</code> and the next successful completion of a call to <code>fflush()</code> or <code>fclose()</code> on the same stream or a call to <code>exit()</code> or <code>abort()</code> .	
8216				
8217				
8218			RETURN VALUE	
8219			Upon successful completion, these functions return the number of bytes transmitted excluding the terminating null in the case of <code>sprintf()</code> or <code>snprintf()</code> or a negative value if an output error was encountered.	
8220				
8221				
8222			If the value of <i>n</i> is zero on a call to <code>snprintf()</code> , an unspecified value less than 1 is returned.	
8223			ERRORS	
8224			For the conditions under which <code>fprintf()</code> and <code>printf()</code> will fail and may fail, refer to <code>fputc()</code> or <code>fputwc()</code> .	
8225				
8226			In addition, all forms of <code>fprintf()</code> may fail if:	
8227	EX	[EILSEQ]	A wide-character code that does not correspond to a valid character has been detected.	
8228				
8229	EX	[EINVAL]	There are insufficient arguments.	
8230	EX		In addition, <code>printf()</code> and <code>fprintf()</code> may fail if:	
8231		[ENOMEM]	Insufficient storage space is available.	
8232			EXAMPLES	
8233			To print the language-independent date and time format, the following statement could be used:	
8234			<code>printf (format, weekday, month, day, hour, min);</code>	
8235			For American usage, <i>format</i> could be a pointer to the string:	
8236			<code>"%s, %s %d, %d:%.2d\n"</code>	
8237			producing the message:	
8238			<code>Sunday, July 3, 10:02</code>	
8239			whereas for German usage, <i>format</i> could be a pointer to the string:	
8240			<code>"%1\$s, %3\$d. %2\$s, %4\$d:%5\$.2d\n"</code>	
8241			producing the message:	
8242			<code>Sonntag, 3. Juli, 10:02</code>	
8243			APPLICATION USAGE	
8244			If the application calling <code>fprintf()</code> has any objects of type <code>wint_t</code> or <code>wchar_t</code> , it must also include the header <code><wchar.h></code> to have these objects defined.	
8245				

8246 **FUTURE DIRECTIONS**

8247 None.

8248 **SEE ALSO**8249 *fputc()*, *fscanf()*, *setlocale()*, *wcrtomb()*, *<stdio.h>*, *<wchar.h>*, the XBD specification, **Chapter 5**,
8250 **Locale**.8251 **CHANGE HISTORY**

8252 First released in Issue 1.

8253 Derived from Issue 1 of the SVID.

8254 **Issue 4**

8255 The following changes are incorporated for alignment with the ISO C standard:

- 8256 • The type of the *format* arguments is changed from **char *** to **const char ***.
- 8257 • The DESCRIPTION is reworded or presented differently in a number of places for alignment
- 8258 with the ISO C standard, and also for clarity. There are no functional changes, except as
- 8259 noted elsewhere in this CHANGE HISTORY section.

8260 The following changes are incorporated for alignment with the MSE working draft:

- 8261 • The C and S conversion characters are added, indicating respectively a wide-character of
- 8262 type **wchar_t** and pointer to a wide-character string of type **wchar_t*** in the argument list.

8263 Other changes are incorporated as follows:

- 8264 • In the DESCRIPTION, references to *langinfo* data are marked as extensions. The reference to
- 8265 *langinfo* data is removed from the description of the radix character.
- 8266 • The ' (single-quote) flag is added to the list of flag characters and marked as an extension.
- 8267 This flag directs that numeric conversion will be formatted with the decimal grouping
- 8268 character.
- 8269 • The detailed description of this function is provided here instead of under *printf()*.
- 8270 • The information in the APPLICATION USAGE section is moved to the DESCRIPTION. A
- 8271 new APPLICATION USAGE section is added.
- 8272 • The [EILSEQ] error is added to the ERRORS section and all errors are marked as extensions.

8273 **Issue 4, Version 2**

8274 The [ENOMEM] error is added to the ERRORS section as an optional error.

8275 **Issue 5**8276 Aligned with the ISO/IEC 9899:1990/Amendment 1:1994 (E). Specifically, the l (ell) qualifier can
8277 now be used with c and s conversion characters.

8278 NAME

8279 fputc — put a byte on a stream

8280 SYNOPSIS

8281 #include <stdio.h>

8282 int fputc(int *c*, FILE **stream*);

8283 DESCRIPTION

8284 The *fputc()* function writes the byte specified by *c* (converted to an **unsigned char**) to the output
 8285 stream pointed to by *stream*, at the position indicated by the associated file-position indicator for
 8286 the stream (if defined), and advances the indicator appropriately. If the file cannot support
 8287 positioning requests, or if the stream was opened with append mode, the byte is appended to
 8288 the output stream.

8289 The *st_ctime* and *st_mtime* fields of the file will be marked for update between the successful
 8290 execution of *fputc()* and the next successful completion of a call to *fflush()* or *fclose()* on the same
 8291 stream or a call to *exit()* or *abort()*.

8292 RETURN VALUE

8293 Upon successful completion, *fputc()* returns the value it has written. Otherwise, it returns EOF,
 8294 the error indicator for the stream is set, and *errno* is set to indicate the error.

8295 ERRORS

8296 The *fputc()* function will fail if either the *stream* is unbuffered or the *stream*'s buffer needs to be
 8297 flushed, and:

8298 [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
 8299 process would be delayed in the write operation.

8300 [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
 8301 writing.

8302 EX [EFBIG] An attempt was made to write to a file that exceeds the maximum file size or
 8303 the process' file size limit.

8304 EX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
 8305 offset maximum.

8306 [EINTR] The write operation was terminated due to the receipt of a signal, and no data
 8307 was transferred.

8308 EX [EIO] A physical I/O error has occurred, or the process is a member of a
 8309 background process group attempting to write to its controlling terminal,
 8310 TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU and the
 8311 process group of the process is orphaned. This error may also be returned
 8312 under implementation-dependent conditions.

8313 [ENOSPC] There was no free space remaining on the device containing the file.

8314 [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
 8315 any process. A SIGPIPE signal will also be sent to the thread.

8316 The *fputc()* function may fail if:

8317 EX [ENOMEM] Insufficient storage space is available.

8318 [ENXIO] A request was made of a non-existent device, or the request was outside the
 8319 capabilities of the device.

8320 **EXAMPLES**

8321 None.

8322 **APPLICATION USAGE**

8323 None.

8324 **FUTURE DIRECTIONS**

8325 None.

8326 **SEE ALSO**8327 *error()*, *fopen()*, *getrlimit()*, *putc()*, *puts()*, *setbuf()*, *ulimit()*, **<stdio.h>**.8328 **CHANGE HISTORY**

8329 First released in Issue 1.

8330 Derived from Issue 1 of the SVID.

8331 **Issue 4**

8332 The following changes are incorporated in this issue:

- 8333 • In the DESCRIPTION, the text is changed to make it clear that the function writes byte
8334 values, rather than (possibly multi-byte) character values.
- 8335 • In the ERRORS section, text is added to indicate that error returns will only be generated
8336 when either the stream is unbuffered, or if the stream buffer needs to be flushed.
- 8337 • Also in the ERRORS section, in previous issues generation of the [EIO] error depended on
8338 whether or not an implementation supported Job Control. This functionality is now defined
8339 as mandatory.
- 8340 • The [ENXIO] error is moved to the list of optional errors, and all the optional errors are
8341 marked as extensions.
- 8342 • The description of [EINTR] is amended.
- 8343 • The [EFBIG] error is marked to show extensions.

8344 **Issue 4, Version 2**

8345 In the ERRORS section, the description of [EIO] is updated to include the case where a physical
8346 I/O error occurs.

8347 **Issue 5**

8348 Large File Summit extensions added.

8349 **NAME**

8350 fputs — put a string on a stream

8351 **SYNOPSIS**

8352 #include <stdio.h>

8353 int fputs(const char *s, FILE *stream);

8354 **DESCRIPTION**8355 The *fputs()* function writes the null-terminated string pointed to by *s* to the stream pointed to by
8356 *stream*. The terminating null byte is not written.8357 The *st_ctime* and *st_mtime* fields of the file will be marked for update between the successful
8358 execution of *fputs()* and the next successful completion of a call to *fflush()* or *fclose()* on the same
8359 stream or a call to *exit()* or *abort()*.8360 **RETURN VALUE**8361 Upon successful completion, *fputs()* returns a non-negative number. Otherwise it returns EOF,
8362 sets an error indicator for the stream and *errno* is set to indicate the error.8363 **ERRORS**8364 Refer to *fputc()*.8365 **EXAMPLES**

8366 None.

8367 **APPLICATION USAGE**8368 The *puts()* function appends a newline character while *fputs()* does not.8369 **FUTURE DIRECTIONS**

8370 None.

8371 **SEE ALSO**8372 *fopen()*, *putc()*, *puts()*, <stdio.h>.8373 **CHANGE HISTORY**

8374 First released in Issue 1.

8375 Derived from Issue 1 of the SVID.

8376 **Issue 4**

8377 The following change is incorporated for alignment with the ISO C standard:

- 8378
- The type of argument *s* is changed from **char *** to **const char ***.

8379 Another change is incorporated as follows:

- 8380
- In the DESCRIPTION, the words “null character” are replaced by “null byte”, to make it
8381 clear that this interface deals solely in byte values.

8382 **NAME**

8383 fputc — put a wide-character code on a stream

8384 **SYNOPSIS**

8385 #include <stdio.h>

8386 #include <wchar.h>

8387 wint_t fputc(wchar_t *wc*, FILE **stream*);8388 **DESCRIPTION**

8389 The *fputc()* function writes the character corresponding to the wide-character code *wc* to the
 8390 output stream pointed to by *stream*, at the position indicated by the associated file-position
 8391 indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot
 8392 support positioning requests, or if the stream was opened with append mode, the character is
 8393 appended to the output stream. If an error occurs whilst writing the character, the shift state of
 8394 the output file is left in an undefined state.

8395 The *st_ctime* and *st_mtime* fields of the file will be marked for update between the successful
 8396 execution of *fputc()* and the next successful completion of a call to *fflush()* or *fclose()* on the
 8397 same stream or a call to *exit()* or *abort()*.

8398 **RETURN VALUE**

8399 Upon successful completion, *fputc()* returns *wc*. Otherwise, it returns WEOF, the error
 8400 indicator for the stream is set, and *errno* is set to indicate the error.

8401 **ERRORS**

8402 The *fputc()* function will fail if either the stream is unbuffered or data in the *stream*'s buffer
 8403 needs to be written, and:

8404 [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
 8405 process would be delayed in the write operation.

8406 [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
 8407 writing.

8408 [EFBIG] An attempt was made to write to a file that exceeds the maximum file size or
 8409 the process' file size limit.

8410 EX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
 8411 offset maximum associated with the corresponding stream.

8412 [EINTR] The write operation was terminated due to the receipt of a signal, and no data
 8413 was transferred.

8414 EX [EIO] A physical I/O error has occurred, or the process is a member of a background
 8415 process group attempting to write to its controlling terminal, TOSTOP is set,
 8416 the process is neither ignoring nor blocking SIGTTOU and the process group
 8417 of the process is orphaned. This error may also be returned under
 8418 implementation-dependent conditions.

8419 [ENOSPC] There was no free space remaining on the device containing the file.

8420 [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
 8421 any process. A SIGPIPE signal will also be sent to the thread.

8422 The *fputcwc()* function may fail if:

8423 [ENOMEM] Insufficient storage space is available.

8424 [ENXIO] A request was made of a non-existent device, or the request was outside the
8425 capabilities of the device.

8426 [EILSEQ] The wide-character code *wc* does not correspond to a valid character.

8427 EXAMPLES

8428 None.

8429 APPLICATION USAGE

8430 None.

8431 FUTURE DIRECTIONS

8432 None.

8433 SEE ALSO

8434 *ferror()*, *fopen()*, *setbuf()*, *ulimit()*, *<stdio.h>*, *<wchar.h>*.

8435 CHANGE HISTORY

8436 First released in Issue 4.

8437 Derived from the MSE working draft.

8438 Issue 4, Version 2

8439 In the ERRORS section, the description of [EIO] is updated to include the case where a physical
8440 I/O error occurs.

8441 Issue 5

8442 Aligned with ISO/IEC 9899:1990/Amendment 1:1994 (E). Specifically, the type of argument *wc*
8443 is changed from **wint_t** to **wchar_t**.

8444 The Optional Header (OH) marking is removed from *<stdio.h>*.

8445 Large File Summit extensions added.

8446 **NAME**

8447 fputws — put a wide-character string on a stream

8448 **SYNOPSIS**

8449 #include <stdio.h>

8450 #include <wchar.h>

8451 int fputws(const wchar_t *ws, FILE *stream);

8452 **DESCRIPTION**

8453 The *fputws()* function writes a character string corresponding to the (null-terminated) wide-character string pointed to by *ws* to the stream pointed to by *stream*. No character corresponding to the terminating null wide-character code is written.

8456 The *st_ctime* and *st_mtime* fields of the file will be marked for update between the successful execution of *fputws()* and the next successful completion of a call to *fflush()* or *fclose()* on the same stream or a call to *exit()* or *abort()*.

8459 **RETURN VALUE**

8460 Upon successful completion, *fputws()* returns a non-negative number. Otherwise it returns -1, sets an error indicator for the stream and *errno* is set to indicate the error.

8462 **ERRORS**8463 Refer to *fputwc()*.8464 **EXAMPLES**

8465 None.

8466 **APPLICATION USAGE**8467 The *fputws()* function does not append a newline character.8468 **FUTURE DIRECTIONS**

8469 None.

8470 **SEE ALSO**8471 *fopen()*, <stdio.h>, <wchar.h>.8472 **CHANGE HISTORY**

8473 First released in Issue 4.

8474 Derived from the MSE working draft.

8475 **Issue 5**

8476 The Optional Header (OH) marking is removed from <stdio.h>.

8477 **NAME**

8478 fread — binary input

8479 **SYNOPSIS**

8480 #include <stdio.h>

8481 size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);

8482 **DESCRIPTION**

8483 The *fread()* function reads, into the array pointed to by *ptr*, up to *nitems* members whose size is
 8484 specified by *size* in bytes, from the stream pointed to by *stream*. The file position indicator for the
 8485 stream (if defined) is advanced by the number of bytes successfully read. If an error occurs, the
 8486 resulting value of the file position indicator for the stream is indeterminate. If a partial member
 8487 is read, its value is indeterminate.

8488 The *fread()* function may mark the *st_atime* field of the file associated with *stream* for update.
 8489 The *st_atime* field will be marked for update by the first successful execution of *fgetc()*, *fgets()*,
 8490 *fgetwc()*, *fgetws()*, *fread()*, *fscanf()*, *getc()*, *getchar()*, *gets()* or *scanf()* using *stream* that returns
 8491 data not supplied by a prior call to *ungetc()* or *ungetwc()*.

8492 **RETURN VALUE**

8493 Upon successful completion, *fread()* returns the number of members successfully read which is
 8494 less than *nitems* only if a read error or end-of-file is encountered. If *size* or *nitems* is 0, *fread()*
 8495 returns 0 and the contents of the array and the state of the stream remain unchanged.
 8496 Otherwise, if a read error occurs, the error indicator for the stream is set and *errno* is set to
 8497 indicate the error.

8498 **ERRORS**8499 Refer to *fgetc()*.8500 **EXAMPLES**

8501 None.

8502 **APPLICATION USAGE**

8503 The *ferror()* or *feof()* functions must be used to distinguish between an error condition and an
 8504 end-of-file condition.

8505 Because of possible differences in member length and byte ordering, files written using *fwrite()*
 8506 are application-dependent, and possibly cannot be read using *fread()* by a different application
 8507 or by the same application on a different processor.

8508 **FUTURE DIRECTIONS**

8509 None.

8510 **SEE ALSO**8511 *feof()*, *ferror()*, *fopen()*, *getc()*, *gets()*, *scanf()*, <stdio.h>.8512 **CHANGE HISTORY**

8513 First released in Issue 1.

8514 Derived from Issue 1 of the SVID.

8515 **Issue 4**

8516 The following change is incorporated for alignment with the ISO C standard:

- 8517 • In the RETURN VALUE section, the behaviour if *size* or *nitems* is 0 is defined.

8518 Another change is incorporated as follows:

- 8519 • The list of functions that may cause the *st_atime* field to be updated is revised.

8520 **NAME**

8521 free — free allocated memory

8522 **SYNOPSIS**

8523 #include <stdlib.h>

8524 void free(void *ptr);

8525 **DESCRIPTION**

8526 The *free()* function causes the space pointed to by *ptr* to be deallocated; that is, made available
 8527 for further allocation. If *ptr* is a null pointer, no action occurs. Otherwise, if the argument does
 8528 not match a pointer earlier returned by the *calloc()*, *malloc()*, *realloc()* or *valloc()* function, or if
 8529 the space is deallocated by a call to *free()* or *realloc()*, the behaviour is undefined.

8530 Any use of a pointer that refers to freed space causes undefined behaviour.

8531 **RETURN VALUE**8532 The *free()* function returns no value.8533 **ERRORS**

8534 No errors are defined.

8535 **EXAMPLES**

8536 None.

8537 **APPLICATION USAGE**

8538 There is now no requirement for the implementation to support the inclusion of <malloc.h>.

8539 **FUTURE DIRECTIONS**

8540 None.

8541 **SEE ALSO**8542 *calloc()*, *malloc()*, *realloc()*, <stdlib.h>.8543 **CHANGE HISTORY**

8544 First released in Issue 1.

8545 Derived from Issue 1 of the SVID.

8546 **Issue 4**

8547 The following change is incorporated for alignment with the ISO C standard:

- 8548 • The DESCRIPTION now states that the behaviour is undefined if any use is made of a
 8549 pointer that refers to freed space. This was implied but not stated explicitly in Issue 3.

8550 Another change is incorporated as follows:

- 8551 • The APPLICATION USAGE section is changed to record that <malloc.h> need no longer be
 8552 supported on XSI-conformant systems.

8553 **Issue 4, Version 2**

8554 The DESCRIPTION is updated for X/OPEN UNIX conformance to indicate that the *free()*
 8555 function can also be used to free memory allocated by *valloc()*.

8556 **NAME**

8557 freopen — open a stream

8558 **SYNOPSIS**

8559 #include <stdio.h>

8560 FILE *freopen(const char *filename, const char *mode, FILE *stream);

8561 **DESCRIPTION**

8562 The *freopen()* function first attempts to flush the stream and close any file descriptor associated
 8563 with *stream*. Failure to flush or close the file successfully is ignored. The error and end-of-file
 8564 indicators for the stream are cleared.

8565 The *freopen()* function opens the file whose pathname is the string pointed to by *filename* and
 8566 associates the stream pointed to by *stream* with it. The *mode* argument is used just as in *fopen()*.

8567 The original stream is closed regardless of whether the subsequent open succeeds.

8568 After a successful call to the *freopen()* function, the orientation of the stream is cleared and the
 8569 associated **mbstate_t** object is set to describe an initial conversion state.

8570 **EX** The largest value that can be represented correctly in an object of type **off_t** will be established
 8571 as the offset maximum in the open file description.

8572 **RETURN VALUE**

8573 Upon successful completion, *freopen()* returns the value of *stream*. Otherwise a null pointer is
 8574 returned and *errno* is set to indicate the error.

8575 **ERRORS**8576 The *freopen()* function will fail if:

8577 [EACCES] Search permission is denied on a component of the path prefix, or the file
 8578 exists and the permissions specified by *mode* are denied, or the file does not
 8579 exist and write permission is denied for the parent directory of the file to be
 8580 created.

8581 [EINTR] A signal was caught during *freopen()*.

8582 [EISDIR] The named file is a directory and *mode* requires write access.

8583 **EX** [ELOOP] Too many symbolic links were encountered in resolving *path*.

8584 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

8585 **FIPS** [ENAMETOOLONG]

8586 The length of the *filename* exceeds {PATH_MAX} or a pathname component is
 8587 longer than {NAME_MAX}.

8588 [ENFILE] The maximum allowable number of files is currently open in the system.

8589 [ENOENT] A component of *filename* does not name an existing file or *filename* is an empty
 8590 string.

8591 [ENOSPC] The directory or file system that would contain the new file cannot be
 8592 expanded, the file does not exist, and it was to be created.

8593 [ENOTDIR] A component of the path prefix is not a directory.

8594 [ENXIO] The named file is a character special or block special file, and the device
 8595 associated with this special file does not exist.

8596	EX	[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 .
8597			
8598		[EROFS]	The named file resides on a read-only file system and <i>mode</i> requires write access.
8599			
8600			The <i>freopen()</i> function may fail if:
8601	EX	[EINVAL]	The value of the <i>mode</i> argument is not valid.
8602	EX	[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.
8603			
8604			
8605		[ENOMEM]	Insufficient storage space is available.
8606		[ENXIO]	A request was made of a non-existent device, or the request was outside the capabilities of the device.
8607			
8608		[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and <i>mode</i> requires write access.
8609			
8610	EXAMPLES		
8611	None.		
8612	APPLICATION USAGE		
8613	The <i>freopen()</i> function is typically used to attach the preopened <i>streams</i> associated with <i>stdin</i> , <i>stdout</i> and <i>stderr</i> to other files.		
8614			
8615	FUTURE DIRECTIONS		
8616	None.		
8617	SEE ALSO		
8618	<i>fclose()</i> , <i>fopen()</i> , <i>fdopen()</i> , <i>mbsinit()</i> , <stdio.h>.		
8619	CHANGE HISTORY		
8620	First released in Issue 1.		
8621	Derived from Issue 1 of the SVID.		
8622	Issue 4		
8623	The following change is incorporated for alignment with the ISO C standard:		
8624	<ul style="list-style-type: none">The type of arguments <i>filename</i> and <i>mode</i> are changed from char * to const char *.		
8625	The following change is incorporated for alignment with the FIPS requirements:		
8626	<ul style="list-style-type: none">In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a pathname component is larger than {NAME_MAX} is now defined as mandatory and marked as an extension.		
8627			
8628			
8629	Other changes are incorporated as follows:		
8630	<ul style="list-style-type: none">In the DESCRIPTION, the word “name” is replaced by “pathname”, to make it clear that the interface is not limited to accepting filenames only.		
8631			
8632	<ul style="list-style-type: none">In the ERRORS section, (a) the description of the [EMFILE] error has been changed to refer to {OPEN_MAX} file descriptors rather than {FOPEN_MAX} file descriptors, directories and message catalogues, (b) the errors [EINVAL], [ENOMEM] and [ETXTBSY] are marked as extensions, and (c) the [ENXIO] error is added in the “may fail” section and marked as an extension.		
8633			
8634			
8635			
8636			

8637 **Issue 4, Version 2**

8638 The ERRORS section is updated for X/OPEN UNIX conformance as follows: |

8639 • It states that [ELOOP] will be returned if too many symbolic links are encountered during
8640 pathname resolution.8641 • A second [ENAMETOOLONG] condition is defined that may report excessive length of an
8642 intermediate result of pathname resolution of a symbolic link.8643 **Issue 5**8644 The DESCRIPTION is updated to indicate that the orientation of the stream is cleared and the
8645 conversion state of the stream is set to an initial conversion state by a successful call to the
8646 *freopen()* function. |

8647 Large File Summit extensions added.

8648 **NAME**

8649 frexp — extract mantissa and exponent from a double precision number

8650 **SYNOPSIS**

8651 #include <math.h>

8652 double frexp(double num, int *exp);

8653 **DESCRIPTION**8654 The *frexp()* function breaks a floating-point number into a normalised fraction and an integral
8655 power of 2. It stores the integer exponent in the **int** object pointed to by *exp*.8656 An application wishing to check for error situations should set *errno* to 0 before calling *frexp()*. If
8657 *errno* is non-zero on return, or the return value is NaN, an error has occurred.8658 **RETURN VALUE**8659 The *frexp()* function returns the value *x*, such that *x* is a **double** with magnitude in the interval
8660 [$\frac{1}{2}$, 1) or 0, and *num* equals *x* times 2 raised to the power **exp*.8661 If *num* is 0, both parts of the result are 0.8662 EX If *num* is NaN, NaN is returned, *errno* may be set to [EDOM] and the value of **exp* is unspecified.8663 If *num* is $\pm\text{Inf}$, *num* is returned, *errno* may be set to [EDOM] and the value of **exp* is unspecified.8664 **ERRORS**8665 The *frexp()* function may fail if:8666 EX [EDOM] The value of *num* is NaN or $\pm\text{Inf}$.

8667 EX No other errors will occur.

8668 **EXAMPLES**

8669 None.

8670 **APPLICATION USAGE**

8671 None.

8672 **FUTURE DIRECTIONS**

8673 None.

8674 **SEE ALSO**8675 *isnan()*, *ldexp()*, *modf()*, <math.h>.8676 **CHANGE HISTORY**

8677 First released in Issue 1.

8678 Derived from Issue 1 of the SVID.

8679 **Issue 4**

8680 The following changes are incorporated in this issue:

- 8681 • Removed references to *matherr()*.
- 8682 • The name of the first argument is changed from *value* to *num*.
- 8683 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
- 8684 the ISO C standard and to rationalise error handling in the mathematics functions.
- 8685 • The return value specified for [EDOM] is marked as an extension.

8686 **Issue 5**

8687 The DESCRIPTION is updated to indicate how an application should check for an error. This
8688 text was previously published in the APPLICATION USAGE section.

8689 NAME

8690 fscanf, scanf, sscanf — convert formatted input

8691 SYNOPSIS

8692 #include <stdio.h>

8693 int fscanf(FILE *stream, const char *format, ...);

8694 int scanf(const char *format, ...);

8695 int sscanf(const char *s, const char *format, ...);

8696 DESCRIPTION

8697 The *fscanf()* function reads from the named input *stream*. The *scanf()* function reads from the
 8698 standard input stream *stdin*. The *sscanf()* function reads from the string *s*. Each function reads
 8699 bytes, interprets them according to a format, and stores the results in its arguments. Each
 8700 expects, as arguments, a control string *format* described below, and a set of *pointer* arguments
 8701 indicating where the converted input should be stored. The result is undefined if there are
 8702 insufficient arguments for the format. If the format is exhausted while arguments remain, the
 8703 excess arguments are evaluated but are otherwise ignored.

8704 EX Conversions can be applied to the *nth* argument after the *format* in the argument list, rather than
 8705 to the next unused argument. In this case, the conversion character % (see below) is replaced by
 8706 the sequence %*n*\$, where *n* is a decimal integer in the range [1, {NL_ARGMAX}]. This feature
 8707 provides for the definition of format strings that select arguments in an order appropriate to
 8708 specific languages. In format strings containing the %*n*\$ form of conversion specifications, it is
 8709 unspecified whether numbered arguments in the argument list can be referenced from the
 8710 format string more than once.

8711 The *format* can contain either form of a conversion specification, that is, % or %*n*\$, but the two
 8712 forms cannot normally be mixed within a single *format* string. The only exception to this is that
 8713 %% or %* can be mixed with the %*n*\$ form.

8714 The *fscanf()* function in all its forms allows for detection of a language-dependent radix
 8715 character in the input string. The radix character is defined in the program's locale (category
 8716 LC_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the
 8717 radix character defaults to a period (.).

8718 The format is a character string, beginning and ending in its initial shift state, if any, composed
 8719 of zero or more directives. Each directive is composed of one of the following: one or more
 8720 white-space characters (space, tab, newline, vertical-tab or form-feed characters); an ordinary
 8721 character (neither % nor a white-space character); or a conversion specification. Each conversion
 8722 EX specification is introduced by the character % or the character sequence %*n*\$ after which the
 8723 following appear in sequence:

- 8724 • An optional assignment-suppressing character *.
- 8725 • An optional non-zero decimal integer that specifies the maximum field width.
- 8726 • An optional size modifier h, l (ell) or L indicating the size of the receiving object. The
 8727 conversion characters d, i and n must be preceded by h if the corresponding argument is a
 8728 pointer to **short int** rather than a pointer to **int**, or by l (ell) if it is a pointer to **long int**.
 8729 Similarly, the conversion characters o, u and x must be preceded by h if the corresponding
 8730 argument is a pointer to **unsigned short int** rather than a pointer to **unsigned int**, or by l (ell)
 8731 if it is a pointer to **unsigned long int**. The conversion characters e, f and g must be preceded
 8732 by l (ell) if the corresponding argument is a pointer to **double** rather than a pointer to **float**,
 8733 or by L if it is a pointer to **long double**. Finally, the conversion characters c, s and [must be
 8734 preceded by l (ell) if the corresponding argument is a pointer to **wchar_t** rather than a pointer
 8735 to a character type. If an h, l (ell) or L appears with any other conversion character, the

behaviour is undefined.

- A conversion character that specifies the type of conversion to be applied. The valid conversion characters are described below.

The *fscanf()* functions execute each directive of the format in turn. If a directive fails, as detailed below, the function returns. 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 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.

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 *[%cCn]* 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 behaviour is undefined.

The following conversion characters are valid:

d	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of <i>strtol()</i> with the value 10 for the <i>base</i> argument. In the absence of a size modifier, the corresponding argument must be a pointer to int .
i	Matches an optionally signed integer, whose format is the same as expected for the subject sequence of <i>strtol()</i> with 0 for the <i>base</i> argument. In the absence of a size modifier, the corresponding argument must be a pointer to int .
o	Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of <i>strtoul()</i> with the value 8 for the <i>base</i> argument. In the absence of a size modifier, the corresponding argument must be a pointer to unsigned int .
u	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of <i>strtoul()</i> with the value 10 for the <i>base</i> argument. In the absence

8782		of a size modifier, the corresponding argument must be a pointer to unsigned int .
8783	x	Matches an optionally signed hexadecimal integer, whose format is the same as
8784		expected for the subject sequence of <i>strtoul()</i> with the value 16 for the <i>base</i> argument.
8785		In the absence of a size modifier, the corresponding argument must be a pointer to
8786		unsigned int .
8787	e, f, g	Matches an optionally signed floating-point number, whose format is the same as
8788		expected for the subject sequence of <i>strtod()</i> . In the absence of a size modifier, the
8789		corresponding argument must be a pointer to float .
8790		If the <i>fprintf()</i> family of functions generates character string representations for infinity
8791		and NaN (a 7858 symbolic entity encoded in floating-point format) to support the
8792		ANSI/IEEE Std 754:1985 standard, the <i>fscanf()</i> family of functions will recognise them
8793		as input.
8794	s	Matches a sequence of bytes that are not white-space characters. The corresponding
8795		argument must be a pointer to the initial byte of an array of char , signed char or
8796		unsigned char large enough to accept the sequence and a terminating null character
8797		code, which will be added automatically.
8798		If an l (ell) qualifier is present, the input is a sequence of characters that begins in the
8799		initial shift state. Each character is converted to a wide-character as if by a call to the
8800		<i>mbrtowc()</i> function, with the conversion state described by an mbstate_t object
8801		initialised to zero before the first character is converted. The corresponding argument
8802		must be a pointer to an array of wchar_t large enough to accept the sequence and the
8803		terminating null wide-character, which will be added automatically.
8804	[Matches a non-empty sequence of characters from a set of expected characters (the
8805		<i>scanset</i>). The normal skip over white-space characters is suppressed in this case. The
8806		corresponding argument must be a pointer to the initial byte of an array of char , signed
8807		char or unsigned char large enough to accept the sequence and a terminating null byte,
8808		which will be added automatically.
8809		If an l (ell) qualifier is present, the input is a sequence of characters that begins in the
8810		initial shift state. Each character in the sequence is converted to a wide-character as if
8811		by a call to the <i>mbrtowc()</i> function, with the conversion state described by an mbstate_t
8812		object initialised to zero before the first character is converted. The corresponding
8813		argument must be a pointer to an array of wchar_t large enough to accept the sequence
8814		and the terminating null wide-character, which will be added automatically.
8815		The conversion specification includes all subsequent characters in the <i>format</i> string up
8816		to and including the matching right square bracket (]). The characters between the
8817		square brackets (the <i>scanlist</i>) comprise the scanset, unless the character after the left
8818		square bracket is a circumflex (^), in which case the scanset contains all characters that
8819		do not appear in the scanlist between the circumflex and the right square bracket. If the
8820		conversion specification begins with [] or [^], the right square bracket is included in the
8821		scanlist and the next right square bracket is the matching right square bracket that ends
8822		the conversion specification; otherwise the first right square bracket is the one that ends
8823		the conversion specification. If a – is in the scanlist and is not the first character, nor the
8824		second where the first character is a ^, nor the last character, the behaviour is
8825		implementation-dependent.
8826	c	Matches a sequence of characters of the number specified by the field width (1 if no
8827		field width is present in the conversion specification). The corresponding argument
8828		must be a pointer to the initial byte of an array of char , signed char or unsigned char
8829		large enough to accept the sequence. No null byte is added. The normal skip over

8830		white-space characters is suppressed in this case.	
8831		If an l (ell) qualifier is present, the input is a sequence of characters that begins in the	
8832		initial shift state. Each character in the sequence is converted to a wide-character as if	
8833		by a call to the <i>mbrtowc()</i> function, with the conversion state described by an mbstate_t	
8834		object initialised to zero before the first character is converted. The corresponding	
8835		argument must be a pointer to an array of wchar_t large enough to accept the resulting	
8836		sequence of wide-characters. No null wide-character is added.	
8837	p	Matches an implementation-dependent set of sequences, which must be the same as	
8838		the set of sequences that is produced by the %p conversion of the corresponding	
8839		<i>fprintf()</i> functions. The corresponding argument must be a pointer to a pointer to void .	
8840		The interpretation of the input item is implementation-dependent. If the input item is a	
8841		value converted earlier during the same program execution, the pointer that results will	
8842		compare equal to that value; otherwise the behaviour of the %p conversion is	
8843		undefined.	
8844	n	No input is consumed. The corresponding argument must be a pointer to the integer	
8845		into which is to be written the number of bytes read from the input so far by this call to	
8846		the <i>fscanf()</i> functions. Execution of a %n conversion specification does not increment	
8847		the assignment count returned at the completion of execution of the function.	
8848	EX	C Same as lc.	
8849	EX	S Same as ls.	
8850	%	Matches a single %; no conversion or assignment occurs. The complete conversion	
8851		specification must be %%. If a conversion specification is invalid, the behaviour is undefined.	
8852			
8853		The conversion characters E, G and X are also valid and behave the same as, respectively, e, g	
8854		and x. If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before	
8855		any bytes matching the current conversion specification (except for %n) have been read (other	
8856		than leading white-space characters, where permitted), execution of the current conversion	
8857		specification terminates with an input failure. Otherwise, unless execution of the current	
8858		conversion specification is terminated with a matching failure, execution of the following	
8859		conversion specification (if any) is terminated with an input failure.	
8860			
8861		Reaching the end of the string in <i>sscanf()</i> is equivalent to encountering end-of-file for <i>fscanf()</i> .	
8862		If conversion terminates on a conflicting input, the offending input is left unread in the input.	
8863		Any trailing white space (including newline characters) is left unread unless matched by a	
8864		conversion specification. The success of literal matches and suppressed assignments is only	
8865		directly determinable via the %n conversion specification. The <i>fscanf()</i> and <i>scanf()</i> functions may mark the <i>st_atime</i> field of the file associated with <i>stream</i>	
8866		for update. The <i>st_atime</i> field will be marked for update by the first successful execution of	
8867		<i>fgetc()</i> , <i>fgets()</i> , <i>fread()</i> , <i>getc()</i> , <i>getchar()</i> , <i>gets()</i> , <i>fscanf()</i> or <i>scanf()</i> using <i>stream</i> that returns data	
8868		not supplied by a prior call to <i>ungetc()</i> .	
8869			
8870	RETURN VALUE		
8871		Upon successful completion, these functions return the number of successfully matched and	
8872		assigned input items; this number can be 0 in the event of an early matching failure. If the input	
8873		ends before the first matching failure or conversion, EOF is returned. If a read error occurs the	
8874		error indicator for the stream is set, EOF is returned, and <i>errno</i> is set to indicate the error.	

8875 **ERRORS**

8876 For the conditions under which the *fscanf()* functions will fail and may fail, refer to *fgetc()* or
8877 *fgetwc()*.

8878 In addition, *fscanf()* may fail if:

8879 EX **[EILSEQ]** Input byte sequence does not form a valid character.

8880 EX **[EINVAL]** There are insufficient arguments.

8881 **EXAMPLES**

8882 The call:

```
8883     int i, n; float x; char name[50];
8884     n = scanf("%d%f%s", &i, &x, name);
```

8885 with the input line:

```
8886     25 54.32E-1 Hamster
```

8887 will assign to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* will contain the string
8888 Hamster.

8889 The call:

```
8890     int i; float x; char name[50];
8891     (void) scanf("%2d%f*d %[0123456789]", &i, &x, name);
```

8892 with input:

```
8893     56789 0123 56a72
```

8894 will assign 56 to *i*, 789.0 to *x*, skip 0123, and place the string 56\0 in *name*. The next call to
8895 *getchar()* will return the character a.

8896 **APPLICATION USAGE**

8897 If the application calling *fscanf()* has any objects of type **wint_t** or **wchar_t**, it must also include
8898 the header **<wchar.h>** to have these objects defined.

8899 **FUTURE DIRECTIONS**

8900 None.

8901 **SEE ALSO**

8902 *getc()*, *printf()*, *setlocale()*, *strtod()*, *strtol()*, *strtoul()*, *wcrtomb()*, **<langinfo.h>**, **<stdio.h>**,
8903 **<wchar.h>**, the XBD specification, **Chapter 5, Locale**.

8904 **CHANGE HISTORY**

8905 First released in Issue 1.

8906 Derived from Issue 1 of the SVID.

8907 **Issue 4**

8908 The following changes are incorporated for alignment with the ISO C standard:

- 8909 • The type of the argument *format* for all functions, and the type of argument *s* for *sscanf()*, are
8910 changed from **char *** to **const char ***.
- 8911 • The description is updated in various places to align more closely with the text of the ISO C
8912 standard. In particular, this issue fully defines the L conversion character, allows for the
8913 support of multi-byte coded character sets (although these are not mandated by X/Open),
8914 and fills in a number of gaps in the definition (for example, by defining termination
8915 conditions for *sscanf()*).

- 8916 • Following an ANSI interpretation, the effect of conversion specifications that consume no
8917 input is better defined, and is no longer marked as an extension.
- 8918 The following change is incorporated for alignment with the MSE working draft.
- 8919 • The C and S conversion characters are added, indicating a pointer in the argument list to the
8920 initial wide-character code of an array large enough to accept the input sequence.
- 8921 Other changes are incorporated as follows:
- 8922 • Use of the terms “byte” and “character” is rationalised to make it clear when single-byte and
8923 multi-byte values can be used. Similarly, use of the terms “conversion specification” and
8924 “conversion character” is now more precise.
- 8925 • Various errors are corrected. For example, the description of the d conversion character
8926 contained an erroneous reference to *strtod()* in Issue 3. This is replaced in this issue by
8927 reference to *strtol()*.
- 8928 • The DESCRIPTION is updated in a number of places to indicate further implications of the
8929 %n\$ form of a conversion. All references to this functionality, which is not specified in the
8930 ISO C standard, are marked as extensions.
- 8931 • The ERRORS section is changed to refer to the entries for *fgetc()* and *fgetwc()*; the [EINVAL]
8932 error is marked as an extension; and the [EILSEQ] error is added and marked as an extension.
- 8933 • The detailed description of this function including the CHANGE HISTORY section for
8934 *scanf()* is provided here instead of under *scanf()*.
- 8935 • The APPLICATION USAGE section is amended to record the need for <sys/types.h> or
8936 <stddef.h> if type *wchar_t* is required.
- 8937 **Issue 5**
- 8938 Aligned with the ISO/IEC 9899:1990/Amendment 1:1994 (E). Specifically, the l (ell) qualifier is
8939 now defined for c, s and [conversion characters.
- 8940 The DESCRIPTION is updated to indicate that if infinity and Nan can be generated by the
8941 *fprintf()* family of functions, then they will be recognised by the *fscanf()* family.

8942 **NAME**

8943 fseek, fseeko — reposition a file-position indicator in a stream

8944 **SYNOPSIS**

8945 #include <stdio.h>

8946 int fseek(FILE *stream, long int offset, int whence);

8947 EX int fseeko(FILE *stream, off_t offset, int whence);

8948

8949 **DESCRIPTION**8950 The *fseek()* function sets the file-position indicator for the stream pointed to by *stream*.

8951 The new position, measured in bytes from the beginning of the file, is obtained by adding *offset* to the position specified by *whence*. The specified point is the beginning of the file for SEEK_SET, the current value of the file-position indicator for SEEK_CUR, or end-of-file for SEEK_END.

8954 If the stream is to be used with wide-character input/output functions, *offset* must either be 0 or a value returned by an earlier call to *ftell()* on the same stream and *whence* must be SEEK_SET.

8956 A successful call to *fseek()* clears the end-of-file indicator for the stream and undoes any effects of *ungetc()* and *ungetwc()* on the same stream. After an *fseek()* call, the next operation on an update stream may be either input or output.

8959 If the most recent operation, other than *ftell()*, on a given stream is *fflush()*, the file offset in the underlying open file description will be adjusted to reflect the location specified by *fseek()*.

8961 The *fseek()* function allows the file-position indicator to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads of data in the gap will return bytes with the value 0 until data is actually written into the gap.

8964 The behaviour of *fseek()* on devices which are incapable of seeking is implementation-dependent. The value of the file offset associated with such a device is undefined.

8966 If the stream is writable and buffered data had not been written to the underlying file, *fseek()* will cause the unwritten data to be written to the file and mark the *st_ctime* and *st_mtime* fields of the file for update.

8969 In a locale with state-dependent encoding, whether *fseek()* restores the stream's shift state is implementation-dependent.

8971 EX The *fseeko()* function is identical to the *fseek()* function except that the *offset* argument is of type *off_t*.

8973 **RETURN VALUE**

8974 EX The *fseek()* and *fseeko()* functions return 0 if they succeed; otherwise they return -1 and set *errno* to indicate the error.

8976 **ERRORS**

8977 EX The *fseek()* and *fseeko()* functions will fail if, either the *stream* is unbuffered or the *stream*'s buffer needed to be flushed, and the call to *fseek()* or *fseeko()* causes an underlying *lseek()* or *write()* to be invoked:

8980 [EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the process would be delayed in the write operation.

8982 [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.

8984	EX	[EFBIG]	An attempt was made to write a file that exceeds the maximum file size or the process' file size limit.
8985			
8986	EX	[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.
8987			
8988		[EINTR]	The write operation was terminated due to the receipt of a signal, and no data was transferred.
8989			
8990		[EINVAL]	The <i>whence</i> argument is invalid. The resulting file-position indicator would be set to a negative value.
8991			
8992	EX	[EIO]	A physical I/O error has occurred, or the process is a member of a background process group attempting to perform a <i>write()</i> 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-dependent conditions.
8993			
8994			
8995			
8996			
8997		[ENOSPC]	There was no free space remaining on the device containing the file.
8998	EX	[EOVERFLOW]	For <i>fseek()</i> , the resulting file offset would be a value which cannot be represented correctly in an object of type long .
8999			
9000	EX	[EOVERFLOW]	For <i>fseeko()</i> , the resulting file offset would be a value which cannot be represented correctly in an object of type off_t .
9001			
9002		[EPIPE]	The file descriptor underlying <i>stream</i> is associated with a pipe or FIFO.
9003		[EPIPE]	An attempt was made to write to a pipe or FIFO that is not open for reading by any process; a SIGPIPE signal will also be sent to the thread.
9004			
9005	EX	[ENXIO]	A request was made of a non-existent device, or the request was outside the capabilities of the device.
9006			

9007 EXAMPLES

9008 None.

9009 APPLICATION USAGE

9010 None.

9011 FUTURE DIRECTIONS

9012 None.

9013 SEE ALSO

9014 *fopen()*, *fsetpos()*, *ftell()*, *getrlimit()*, *rewind()*, *ulimit()*, *ungetc()*, **<stdio.h>**.

9015 CHANGE HISTORY

9016 First released in Issue 1.

9017 Derived from Issue 1 of the SVID.

9018 Issue 4

9019 The following change is incorporated for alignment with the ISO C standard:

- 9020 • The type of argument *offset* is now defined in full as **long int** instead of **long**.

9021 The following change is incorporated for alignment with the FIPS requirements:

- 9022 • The [EINTR] error is no longer an indication that the implementation does not report partial transfers.

9024 Other changes are incorporated as follows:

9025	• In the DESCRIPTION, the words “The <i>seek()</i> function does not, by itself, extend the size of a	
9026	file” are deleted.	
9027	• In the RETURN VALUE section, the value <code>-1</code> is marked as an extension. This is because the	
9028	ISO POSIX-1 standard only requires that a non-zero value is returned.	
9029	• In the ERRORS section, text is added to indicate that error returns will only be generated	
9030	when either the stream is unbuffered, or if the stream buffer needs to be flushed.	
9031	• The “will fail” and “may fail” parts of the ERRORS section are revised for consistency with	
9032	<i>lseek()</i> and <i>write()</i> .	
9033	• Text associated with the [EIO] error is expanded and the [ENXIO] error is added.	
9034	• Text is added to explain how <i>fseek()</i> is used with wide-character input/output; this is marked	
9035	as a WP extension.	
9036	• The [EFBIG] error is marked to show extensions.	
9037	• The APPLICATION USAGE section is added.	
9038	Issue 4, Version 2	
9039	In the ERRORS section, the description of [EIO] is updated to include the case where a physical	
9040	I/O error occurs.	
9041	Issue 5	
9042	Normative text previously in the APPLICATION USAGE section is moved to the	
9043	DESCRIPTION.	
9044	Large File Summit extensions added.	

9045 **NAME**

9046 fsetpos — set current file position

9047 **SYNOPSIS**

9048 #include <stdio.h>

9049 int fsetpos(FILE **stream*, const fpos_t **pos*);9050 **DESCRIPTION**

9051 The *fsetpos()* function sets the file position and state indicators for the stream pointed to by *stream* according to the value of the object pointed to by *pos*, which must be a value obtained from an earlier call to *fgetpos()* on the same stream.

9054 A successful call to *fsetpos()* function clears the end-of-file indicator for the stream and undoes any effects of *ungetc()* on the same stream. After an *fsetpos()* call, the next operation on an update stream may be either input or output.

9057 **RETURN VALUE**

9058 The *fsetpos()* function returns 0 if it succeeds; otherwise it returns a non-zero value and sets *errno* to indicate the error.

9060 **ERRORS**9061 The *fsetpos()* function may fail if:9062 EX [EBADF] The file descriptor underlying *stream* is not valid.9063 [ESPIPE] The file descriptor underlying *stream* is associated with a pipe or FIFO.9064 **EXAMPLES**

9065 None.

9066 **APPLICATION USAGE**

9067 None.

9068 **FUTURE DIRECTIONS**

9069 None.

9070 **SEE ALSO**9071 *fopen()*, *ftell()*, *rewind()*, *ungetc()*, <stdio.h>.9072 **CHANGE HISTORY**

9073 First released in Issue 4.

9074 Derived from the ISO C standard.

9075 NAME

9076 fstat — get file status

9077 SYNOPSIS

9078 OH #include <sys/types.h>

9079 #include <sys/stat.h>

9080 int fstat(int *fildev*, struct stat **buf*);

9081 DESCRIPTION

9082 The *fstat()* function obtains information about an open file associated with the file descriptor
 9083 *fildev*, and writes it to the area pointed to by *buf*.

9084 RT If `_XOPEN_REALTIME` is defined and has a value other than `-1`, and *fildev* references a shared
 9085 memory object, the implementation need update in the **stat** structure pointed to by the *buf*
 9086 argument only the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the `S_IRUSR`, `S_IWUSR`,
 9087 `S_IRGRP`, `S_IWGRP`, `S_IROTH`, and `S_IWOTH` file permission bits need be valid.

9088 The *buf* argument is a pointer to a **stat** structure, as defined in `<sys/stat.h>`, into which
 9089 information is placed concerning the file.

9090 The structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime* and *st_mtime*
 9091 will have meaningful values for all file types defined in this document. The value of the member
 9092 *st_nlink* will be set to the number of links to the file.

9093 An implementation that provides additional or alternative file access control mechanisms may,
 9094 under implementation-dependent conditions, cause *fstat()* to fail.

9095 The *fstat()* function updates any time-related fields as described in **File Times Update** (see the
 9096 **XBD** specification, **Chapter 4, Character Set**), before writing into the *stat* structure.

9097 RETURN VALUE

9098 Upon successful completion, 0 is returned. Otherwise, `-1` is returned and *errno* is set to indicate
 9099 the error.

9100 ERRORS

9101 The *fstat()* function will fail if:9102 [EBADF] The *fildev* argument is not a valid file descriptor.

9103 EX [EIO] An I/O error occurred while reading from the file system.

9104 EX [EOVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file
 9105 serial number cannot be represented correctly in the structure pointed to by
 9106 *buf*.

9107 EX The *fstat()* function may fail if:

9108 [EOVERFLOW] One of the values is too large to store into the structure pointed to by the *buf*
 9109 argument.

9110 EXAMPLES

9111 None.

9112 APPLICATION USAGE

9113 None.

9114 FUTURE DIRECTIONS

9115 None.

9116 **SEE ALSO**9117 *lstat()*, *stat()*, `<sys/stat.h>`, `<sys/types.h>`.9118 **CHANGE HISTORY**

9119 First released in Issue 1.

9120 Derived from Issue 1 of the SVID.

9121 **Issue 4**9122 The following changes are incorporated in the DESCRIPTION for alignment with the
9123 ISO POSIX-1 standard:

- 9124 • A paragraph defining the contents of *stat* structure members is added.
- 9125 • The words “extended security controls” are replaced by “additional or alternative file access
- 9126 control mechanisms”.

9127 Another change is incorporated as follows:

- 9128 • The `<sys/types.h>` header is now marked as optional (OH); this header need not be included
- 9129 on XSI-conformant systems.

9130 **Issue 4, Version 2**

9131 The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- 9132 • The [EIO] error is added as a mandatory error indicated the occurrence of an I/O error.
- 9133 • The [EOVERFLOW] error is added as an optional error indicating that one of the values is
- 9134 too large to store in the area pointed to by *buf*.

9135 **Issue 5**

9136 The DESCRIPTION is updated for alignment with POSIX Realtime Extension.

9137 Large File Summit extensions added.

9138 **NAME**

9139 fstatvfs, statvfs — get file system information

9140 **SYNOPSIS**

9141 EX #include <sys/statvfs.h>

9142 int fstatvfs(int *fildev*, struct statvfs **buf*);9143 int statvfs(const char **path*, struct statvfs **buf*);

9144

9145 **DESCRIPTION**9146 The *fstatvfs()* function obtains information about the file system containing the file referenced by
9147 *fildev*.9148 The following flags can be returned in the **f_flag** member:

9149 ST_RDONLY Read-only file system.

9150 ST_NOSUID Setuid/setgid bits ignored by exec.

9151 The *statvfs()* function obtains descriptive information about the file system containing the file
9152 named by *path*.9153 For both functions, the *buf* argument is a pointer to a **statvfs** structure that will be filled. Read,
9154 write, or execute permission of the named file is not required, but all directories listed in the
9155 pathname leading to the file must be searchable.9156 It is unspecified whether all members of the **statvfs** structure have meaningful values on all file
9157 systems.9158 **RETURN VALUE**9159 Upon successful completion, *statvfs()* returns 0. Otherwise, it returns -1 and sets *errno* to
9160 indicate the error.9161 **ERRORS**9162 The *fstatvfs()* and *statvfs()* functions will fail if:

9163 [EIO] An I/O error occurred while reading the file system.

9164 [EINTR] A signal was caught during execution of the function.

9165 EX [EOVERFLOW] One of the values to be returned cannot be represented correctly in the
9166 structure pointed to by **buf**.9167 The *fstatvfs()* function will fail if:9168 [EBADF] The *fildev* argument is not an open file descriptor.9169 The *statvfs()* function will fail if:9170 [EACCES] Search permission is denied on a component of the *path* prefix.9171 [ELOOP] Too many symbolic links were encountered in resolving *path*.

9172 [ENAMETOOLONG] The length of a pathname exceeds {PATH_MAX}, or a pathname component is

9173 longer than {NAME_MAX}.
91749175 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.9176 [ENOTDIR] A component of the path prefix of *path* is not a directory.

9177 The *statvfs()* function may fail if: |

9178 [ENAMETOOLONG] |

9179 Pathname resolution of a symbolic link produced an intermediate result |

9180 whose length exceeds {PATH_MAX}. |

9181 **EXAMPLES** |

9182 None. |

9183 **APPLICATION USAGE** |

9184 None. |

9185 **FUTURE DIRECTIONS** |

9186 None. |

9187 **SEE ALSO** |

9188 *chmod()*, *chown()*, *creat()*, *dup()*, *exec*, *fcntl()*, *link()*, *mknod()*, *open()*, *pipe()*, *read()*, *time()*, |

9189 *unlink()*, *ustat()*, *utime()*, *write()*, <sys/statvfs.h>. |

9190 **CHANGE HISTORY** |

9191 First released in Issue 4, Version 2. |

9192 **Issue 5** |

9193 Moved from X/OPEN UNIX extension to BASE. |

9194 Large File Summit extensions added. |

9195 **NAME**

9196 fsync — synchronise changes to a file

9197 **SYNOPSIS**

9198 #include <unistd.h>

9199 int fsync(int *fil*des);9200 **DESCRIPTION**

9201 The *fsync()* function can be used by an application to indicate that all data for the open file
 9202 description named by *fil*des is to be transferred to the storage device associated with the file
 9203 described by *fil*des in an implementation-dependent manner. The *fsync()* function does not
 9204 return until the system has completed that action or until an error is detected.

9205 **RT** The *fsync()* function forces all currently queued I/O operations associated with the file indicated
 9206 by file descriptor *fil*des to the synchronised I/O completion state. All I/O operations are
 9207 completed as defined for synchronised I/O file integrity completion.

9208 **RETURN VALUE**

9209 Upon successful completion, *fsync()* returns 0. Otherwise, -1 is returned and *errno* is set to
 9210 indicate the error. If the *fsync()* function fails, outstanding I/O operations are not guaranteed to
 9211 have been completed.

9212 **ERRORS**9213 The *fsync()* function will fail if:9214 [EBADF] The *fil*des argument is not a valid descriptor.9215 [EINTR] The *fsync()* function was interrupted by a signal.9216 [EINVAL] The *fil*des argument does not refer to a file on which this operation is possible.

9217 [EIO] An I/O error occurred while reading from or writing to the file system.

9218 In the event that any of the queued I/O operations fail, *fsync()* returns the error conditions
 9219 defined for *read()* and *write()*.

9220 **EXAMPLES**

9221 None.

9222 **APPLICATION USAGE**

9223 The *fsync()* function should be used by programs which require modifications to a file to be
 9224 completed before continuing; for example, a program which contains a simple transaction
 9225 facility might use it to ensure that all modifications to a file or files caused by a transaction are
 9226 recorded.

9227 **FUTURE DIRECTIONS**

9228 None.

9229 **SEE ALSO**9230 *sync()*, <unistd.h>.9231 **CHANGE HISTORY**

9232 First released in Issue 3.

9233 **Issue 4**

9234 The following changes are incorporated in this issue:

- 9235 • The <unistd.h> header is added to the SYNOPSIS section.
- 9236 • In the APPLICATION USAGE section, the words “require a file to be in a known state” are
 9237 replaced by “require modifications to a file to be completed before continuing”.

9238 Issue 5

9239 Aligned with *fsync()* in the POSIX Realtime Extension. Specifically, the DESCRIPTION and
9240 RETURN VALUE sections are much expanded, and the ERRORS section is updated to indicate
9241 that *fsync()* can return the error conditions defined for *read()* and *write()*.

9242 **NAME**

9243 ftell, ftello — return a file offset in a stream

9244 **SYNOPSIS**

9245 #include <stdio.h>

9246 long int ftell(FILE *stream);

9247 EX off_t ftello(FILE *stream);

9248

9249 **DESCRIPTION**9250 The *ftell()* function obtains the current value of the file-position indicator for the stream pointed
9251 to by *stream*.9252 EX The *ftello()* function is identical to *ftell()* except that the return value is of type **off_t**.9253 **RETURN VALUE**9254 EX Upon successful completion, *ftell()* and *ftello()* return the current value of the file-position
9255 indicator for the stream measured in bytes from the beginning of the file.9256 EX Otherwise, *ftell()* and *ftello()* return **-1**, cast to **long** and **off_t** respectively, and set *errno* to
9257 indicate the error.9258 **ERRORS**9259 EX The *ftell()* and *ftello()* functions will fail if:9260 [EBADF] The file descriptor underlying *stream* is not an open file descriptor.9261 EX [EOVERFLOW] For *ftell()*, the current file offset cannot be represented correctly in an object of
9262 type **long**.9263 EX [EOVERFLOW] For *ftello()*, the current file offset cannot be represented correctly in an object
9264 of type **off_t**.9265 [ESPIPE] The file descriptor underlying *stream* is associated with a pipe or FIFO.9266 **EXAMPLES**

9267 None.

9268 **APPLICATION USAGE**

9269 None.

9270 **FUTURE DIRECTIONS**

9271 None.

9272 **SEE ALSO**9273 *fgetpos()*, *fopen()*, *fseek()*, *ftello()*, *lseek()*, <stdio.h>.9274 **CHANGE HISTORY**

9275 First released in Issue 1.

9276 Derived from Issue 1 of the SVID.

9277 **Issue 4**

9278 The following change is incorporated for alignment with the ISO C standard:

- 9279 • The function return value is now defined in full as
- long int**
- . It was previously defined as
-
- 9280
- long**
- .

9281 **Issue 5**

9282 Large File Summit extensions added.

9283 NAME

9284 ftime — get date and time

9285 SYNOPSIS

9286 EX `#include <sys/timeb.h>`

9287 `int ftime(struct timeb *tp);`

9288

9289 DESCRIPTION

9290 The *ftime()* function sets the **time** and **millitm** members of the **timeb** structure pointed to by *tp*
 9291 to contain the seconds and milliseconds portions, respectively, of the current time in seconds
 9292 since 00:00:00 UTC (Coordinated Universal Time), January 1, 1970. The contents of the **timezone**
 9293 and **dstflag** members of *tp* after a call to *ftime()* are unspecified.

9294 The system clock need not have millisecond granularity. Depending on any granularity
 9295 (particularly a granularity of one) renders code non-portable.

9296 RETURN VALUE

9297 Upon successful completion, the *ftime()* function returns 0. Otherwise -1 is returned.

9298 ERRORS

9299 No errors are defined.

9300 EXAMPLES

9301 None.

9302 APPLICATION USAGE

9303 For portability to implementations conforming to earlier versions of this specification, *time()* is
 9304 preferred over this function.

9305 FUTURE DIRECTIONS

9306 None.

9307 SEE ALSO

9308 *ctime()*, *gettimeofday()*, *time()*, `<sys/timeb.h>`.

9309 CHANGE HISTORY

9310 First released in Issue 4, Version 2.

9311 Issue 5

9312 Moved from X/OPEN UNIX extension to BASE.

9313 Normative text previously in the APPLICATION USAGE section is moved to the
 9314 DESCRIPTION.

9315 **NAME**

9316 ftok — generate an IPC key

9317 **SYNOPSIS**

9318 EX #include <sys/ipc.h>

9319 key_t ftok(const char *path, int id);

9320

9321 **DESCRIPTION**

9322 The *ftok()* function returns a key based on *path* and *id* that is usable in subsequent calls to *msgget()*, *semget()* and *shmget()*. The *path* argument must be the pathname of an existing file that the process is able to *stat()*.

9325 The *ftok()* function will return the same key value for all paths that name the same file, when called with the same *id* value, and will return different key values when called with different *id* values or with paths that name different files existing on the same file system at the same time. It is unspecified whether *ftok()* returns the same key value when called again after the file named by *path* is removed and recreated with the same name.

9330 Only the low order 8-bits of *id* are significant. The behaviour of *ftok()* is unspecified if these bits are 0.

9332 **RETURN VALUE**

9333 Upon successful completion, *ftok()* returns a key. Otherwise, *ftok()* returns (**key_t**)−1 and sets *errno* to indicate the error.

9335 **ERRORS**9336 The *ftok()* function will fail if:

9337 [EACCES] Search permission is denied for a component of the path prefix.

9338 [ELOOP] Too many symbolic links were encountered in resolving *path*.

9339 [ENAMETOOLONG]

9340 The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

9342 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

9343 [ENOTDIR] A component of the path prefix is not a directory.

9344 The *ftok()* function may fail if:

9345 [ENAMETOOLONG]

9346 Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

9348 **EXAMPLES**

9349 None.

9350 **APPLICATION USAGE**9351 For maximum portability, *id* should be a single-byte character.9352 **FUTURE DIRECTIONS**

9353 None.

9354 **SEE ALSO**9355 *msgget()*, *semget()*, *shmget()*, <sys/ipc.h>.

9356 **CHANGE HISTORY**

9357 First released in Issue 4, Version 2.

9358 **Issue 5**

9359 Moved from X/OPEN UNIX extension to BASE.

|
|

9360 **NAME**

9361 ftruncate, truncate — truncate a file to a specified length

9362 **SYNOPSIS**

9363 #include <unistd.h>

9364 int ftruncate(int *filides*, off_t *length*);9365 EX int truncate(const char **path*, off_t *length*);

9366

9367 **DESCRIPTION**9368 The *ftruncate()* function causes the regular file referenced by *filides* to have a size of *length* bytes.9369 EX The *truncate()* function causes the regular file named by *path* to have a size of *length* bytes.

9370 If the file previously was larger than *length*, the extra data is discarded. If it was previously
 9371 shorter than *length*, it is unspecified whether the file is changed or its size increased. If the file is
 9372 RT extended, the extended area appears as if it were zero-filled. If *filides* references a shared memory
 9373 object, *ftruncate()* sets the size of the shared memory object to *length*. If the file is not a regular
 9374 file or a shared memory object, the result is unspecified.

9375 EX With *ftruncate()*, the file must be open for writing; for *truncate()*, the process must have write
 9376 permission for the file.

9377 RT If the effect of truncation is to decrease the size of a file or shared memory object and whole
 9378 pages beyond the new end were previously mapped, then the whole pages beyond the new end
 9379 will be discarded. References to the discarded pages result in generation of a *SIGBUS* signal.

9380 EX If the request would cause the file size to exceed the soft file size limit for the process, the
 9381 request will fail and the implementation will generate the *SIGXFSZ* signal for the process.

9382 These functions do not modify the file offset for any open file descriptions associated with the
 9383 file. On successful completion, if the file size is changed, these functions will mark for update
 9384 the *st_ctime* and *st_mtime* fields of the file, and if the file is a regular file, the *S_ISUID* and
 9385 *S_ISGID* bits of the file mode may be cleared.

9386 **RETURN VALUE**

9387 EX Upon successful completion, *ftruncate()* and *truncate()* return 0. Otherwise a *-1* is returned, and
 9388 *errno* is set to indicate the error.

9389 **ERRORS**9390 EX The *ftruncate()* and *truncate()* functions will fail if:

9391 [EINTR] A signal was caught during execution.

9392 [EINVAL] The *length* argument was less than 0.

9393 [EFBIG] or [EINVAL]

9394 The *length* argument was greater than the maximum file size.

9395 [EIO] An I/O error occurred while reading from or writing to a file system.

9396 The *ftruncate()* function will fail if:

9397 [EBADF] or [EINVAL]

9398 The *filides* argument is not a file descriptor open for writing.

9399 EX [EFBIG] The file is a regular file and *length* is greater than the offset maximum
 9400 established in the open file description associated with *filides*.

9401 [EINVAL] The *filides* argument references a file that was opened without write
 9402 permission.

9403	[EROFS]	The named file resides on a read-only file system.
9404	EX	The <i>truncate()</i> function will fail if:
9405	[EACCES]	A component of the path prefix denies search permission, or write permission
9406		is denied on the file.
9407	[EISDIR]	The named file is a directory.
9408	[ELOOP]	Too many symbolic links were encountered in resolving <i>path</i> .
9409	[ENAMETOOLONG]	
9410		The length of the specified pathname exceeds PATH_MAX bytes, or the length
9411		of a component of the pathname exceeds NAME_MAX bytes.
9412	[ENOENT]	A component of <i>path</i> does not name an existing file or <i>path</i> is an empty string.
9413	[ENOTDIR]	A component of the path prefix of <i>path</i> is not a directory.
9414	[EROFS]	The named file resides on a read-only file system.
9415		The <i>truncate()</i> function may fail if:
9416	[ENAMETOOLONG]	
9417		Pathname resolution of a symbolic link produced an intermediate result
9418		whose length exceeds {PATH_MAX}.
9419		
9420	EXAMPLES	
9421	None.	
9422	APPLICATION USAGE	
9423	None.	
9424	FUTURE DIRECTIONS	
9425	None.	
9426	SEE ALSO	
9427	<i>open()</i> , <unistd.h>.	
9428	CHANGE HISTORY	
9429	First released in Issue 4, Version 2.	
9430	Issue 5	
9431	Moved from X/OPEN UNIX extension to BASE and aligned with <i>ftruncate()</i> in the POSIX	
9432	Realtime Extension. Specifically, the DESCRIPTION is extensively reworded and [EROFS] is	
9433	added to the list of mandatory errors that can be returned by <i>ftruncate()</i> .	
9434	Large File Summit extensions added.	

9435 **NAME**

9436 ftrylockfile — stdio locking functions

9437 **SYNOPSIS**

9438 #include <stdio.h>

9439 int ftrylockfile(FILE **file*);9440 **DESCRIPTION**9441 Refer to *flockfile()*.9442 **CHANGE HISTORY**

9443 First released in Issue 5.

9444 Included for alignment with the POSIX Threads Extension.

9445 **NAME**

9446 ftw — traverse (walk) a file tree

9447 **SYNOPSIS**9448 EX `#include <ftw.h>`

```
9449 int ftw(const char *path, int (*fn)(const char *,
9450      const struct stat *ptr, int flag), int ndirs);
9451
```

9452 **DESCRIPTION**

9453 The *ftw()* function recursively descends the directory hierarchy rooted in *path*. For each object in
 9454 the hierarchy, *ftw()* calls the function pointed to by *fn*, passing it a pointer to a null-terminated
 9455 character string containing the name of the object, a pointer to a **stat** structure containing
 9456 information about the object, and an integer. Possible values of the integer, defined in the
 9457 **<ftw.h>** header, are:

9458 FTW_D For a directory.

9459 FTW_DNR For a directory that cannot be read.

9460 FTW_F For a file.

9461 EX FTW_SL For a symbolic link (but see also FTW_NS below).

9462 EX FTW_NS For an object other than a symbolic link on which *stat()* could not successfully be
 9463 EX executed. If the object is a symbolic link and *stat()* failed, it is unspecified whether
 9464 *ftw()* passes FTW_SL or FTW_NS to the user-supplied function.

9465 If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is
 9466 FTW_NS, the **stat** structure will contain undefined values. An example of an object that would
 9467 cause FTW_NS to be passed to the function pointed to by *fn* would be a file in a directory with
 9468 read but without execute (search) permission.

9469 The *ftw()* function visits a directory before visiting any of its descendants.9470 EX The *ftw()* function uses at most one file descriptor for each level in the tree.9471 The argument *ndirs* should be in the range of 1 to {OPEN_MAX}.

9472 The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a non-zero
 9473 value, or some error, other than [EACCES], is detected within *ftw()*.

9474 The *ndirs* argument specifies the maximum number of directory streams or file descriptors or
 9475 both available for use by *ftw()* while traversing the tree. When *ftw()* returns it closes any
 9476 directory streams and file descriptors it uses not counting any opened by the application-
 9477 supplied *fn()* function.

9478 **RETURN VALUE**

9479 If the tree is exhausted, *ftw()* returns 0. If the function pointed to by *fn* returns a non-zero value,
 9480 *ftw()* stops its tree traversal and returns whatever value was returned by the function pointed to
 9481 by *fn()*. If *ftw()* detects an error, it returns -1 and sets *errno* to indicate the error.

9482 EX If *ftw()* encounters an error other than [EACCES] (see FTW_DNR and FTW_NS above), it
 9483 returns -1 and *errno* is set to indicate the error. The external variable *errno* may contain any error
 9484 value that is possible when a directory is opened or when one of the *stat* functions is executed on
 9485 a directory or file.

9486 **ERRORS**9487 The *ftw()* function will fail if:9488 [EACCES] Search permission is denied for any component of *path* or read permission is
9489 denied for *path*.

9490 [ELOOP] Too many symbolic links were encountered.

9491 [ENAMETOOLONG]

9492 The length of the *path* exceeds {PATH_MAX}, or a pathname component is
9493 longer than {NAME_MAX}.9494 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.9495 [ENOTDIR] A component of *path* is not a directory.9496 The *ftw()* function may fail if:9497 [EINVAL] The value of the *ndirs* argument is invalid.

9498 EX [ENAMETOOLONG]

9499 Pathname resolution of a symbolic link produced an intermediate result
9500 whose length exceeds {PATH_MAX}.9501 In addition, if the function pointed to by *fn* encounters system errors, *errno* may be set
9502 accordingly.9503 **EXAMPLES**

9504 None.

9505 **APPLICATION USAGE**9506 The *ftw()* may allocate dynamic storage during its operation. If *ftw()* is forcibly terminated,
9507 such as by *longjmp()* or *siglongjmp()* being executed by the function pointed to by *fn* or an
9508 interrupt routine, *ftw()* will not have a chance to free that storage, so it will remain permanently
9509 allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and
9510 arrange to have the function pointed to by *fn* return a non-zero value at its next invocation.9511 **FUTURE DIRECTIONS**

9512 None.

9513 **SEE ALSO**9514 *longjmp()*, *lstat()*, *malloc()*, *nftw()*, *opendir()*, *siglongjmp()*, *stat()*, <ftw.h>, <sys/stat.h>.9515 **CHANGE HISTORY**

9516 First released in Issue 1.

9517 Derived from Issue 1 of the SVID.

9518 **Issue 4**

9519 The following change is incorporated for alignment with the FIPS requirements:

- 9520
- In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a
9521 pathname component is larger than {NAME_MAX} is now defined as mandatory and marked
9522 as an extension.

9523 Other changes are incorporated as follows:

- 9524
- The type of argument *path* is changed from **char *** to **const char ***. The argument list for *fn()*
9525 has also been defined.
 - In the DESCRIPTION, the words “other than [EACCES]” are added to the paragraph
9526 describing termination conditions for tree traversal.
9527

9528 **Issue 4, Version 2**

9529 The following changes are incorporated for X/OPEN UNIX conformance:

- 9530 • The DESCRIPTION is updated to describe the use of the FTW_SL and FTW_NS values for a
9531 symbolic link. |
- 9532 • The DESCRIPTION states that *ftw()* uses at most one file descriptor for each level in the tree. |
- 9533 • The DESCRIPTION constrains *ndirs* to the range from 1 to {OPEN_MAX}. |
- 9534 • The RETURN VALUE section is updated to describe the case where *ftw()* encounters an error
9535 other than [EACCES]. |
- 9536 • In the ERRORS section, a second [ENAMETOOLONG] condition is defined that may report
9537 excessive length of an intermediate result of pathname resolution of a symbolic link. |

9538 **Issue 5**9539 UX codings in the DESCRIPTION, RETURN VALUE and ERRORS sections have been changed
9540 to EX. |

9541 **NAME**

9542 funlockfile — stdio locking functions

9543 **SYNOPSIS**

9544 #include <stdio.h>

9545 void funlockfile(FILE **file*);9546 **DESCRIPTION**9547 Refer to *flockfile()*.9548 **CHANGE HISTORY**

9549 First released in Issue 5.

9550 Included for alignment with the POSIX Threads Extension.

9551 **NAME**

9552 fwide — set stream orientation

9553 **SYNOPSIS**

9554 #include <stdio.h>

9555 #include <wchar.h>

9556 int fwide(FILE **stream*, int *mode*);9557 **DESCRIPTION**

9558 The *fwide()* function determines the orientation of the stream pointed to by *stream*. If *mode* is
9559 greater than zero, the function first attempts to make the stream wide-orientated. If *mode* is less
9560 than zero, the function first attempts to make the stream byte-orientated. Otherwise, *mode* is
9561 zero and the function does not alter the orientation of the stream.

9562 If the orientation of the stream has already been determined, *fwide()* does not change it.

9563 Because no return value is reserved to indicate an error, an application wishing to check for error
9564 situations should set *errno* to 0, then call *fwide()*, then check *errno* and if it is non-zero, assume an
9565 error has occurred.

9566 **RETURN VALUE**

9567 The *fwide()* function returns a value greater than zero if, after the call, the stream has wide-
9568 orientation, a value less than zero if the stream has byte-orientation, or zero if the stream has no
9569 orientation.

9570 **ERRORS**9571 The *fwide()* function may fail if:

9572 EX [EBADF] The *stream* argument is not a valid stream.

9573 **EXAMPLES**

9574 None.

9575 **APPLICATION USAGE**

9576 A call to *fwide()* with *mode* set to zero can be used to determine the current orientation of a
9577 stream.

9578 **FUTURE DIRECTIONS**

9579 None.

9580 **SEE ALSO**

9581 <wchar.h>.

9582 **CHANGE HISTORY**

9583 First released in Issue 5.

9584 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

9585 **NAME**

9586 fwprintf, wprintf, swprintf — print formatted wide-character output

9587 **SYNOPSIS**

9588 #include <stdio.h>

9589 #include <wchar.h>

9590 int fwprintf(FILE **stream*, const wchar_t **format*, ...);9591 int wprintf(const wchar_t **format*, ...);9592 int swprintf(wchar_t **s*, size_t *n*, const wchar_t **format*, ...);9593 **DESCRIPTION**

9594 The *fwprintf()* function places output on the named output *stream*. The *wprintf()* function places
 9595 output on the standard output stream *stdout*. The *swprintf()* function places output followed by
 9596 the null wide-character in consecutive wide-characters starting at **s*; no more than *n* wide-
 9597 characters are written, including a terminating null wide-character, which is always added
 9598 (unless *n* is zero).

9599 Each of these functions converts, formats and prints its arguments under control of the *format*
 9600 wide-character string. The *format* is composed of zero or more directives: *ordinary wide-*
 9601 *characters*, which are simply copied to the output stream and *conversion specifications*, each of
 9602 which results in the fetching of zero or more arguments. The results are undefined if there are
 9603 insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the
 9604 excess arguments are evaluated but are otherwise ignored.

9605 EX Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
 9606 to the next unused argument. In this case, the conversion wide-character % (see below) is
 9607 replaced by the sequence %*n*\$, where *n* is a decimal integer in the range [1, {NL_ARGMAX}],
 9608 giving the position of the argument in the argument list. This feature provides for the definition
 9609 of format wide-character strings that select arguments in an order appropriate to specific
 9610 languages (see the EXAMPLES section).

9611 In format wide-character strings containing the %*n*\$ form of conversion specifications,
 9612 numbered arguments in the argument list can be referenced from the format wide-character
 9613 string as many times as required.

9614 In format wide-character strings containing the % form of conversion specifications, each
 9615 argument in the argument list is used exactly once.

9616 All forms of the *fwprintf()* functions allow for the insertion of a language-dependent radix
 9617 character in the output string, output as a wide-character value. The radix character is defined
 9618 in the program's locale (category LC_NUMERIC). In the POSIX locale, or in a locale where the
 9619 radix character is not defined, the radix character defaults to a period (.).

9620 EX Each conversion specification is introduced by the % wide-character or by the wide-character
 9621 sequence %*n*\$, after which the following appear in sequence:

- 9622 • Zero or more *flags* (in any order), which modify the meaning of the conversion specification.
- 9623 • An optional minimum *field width*. If the converted value has fewer wide-characters than the
 9624 field width, it will be padded with spaces by default on the left; it will be padded on the right,
 9625 if the left-adjustment flag (–), described below, is given to the field width. The field width
 9626 takes the form of an asterisk (*), described below, or a decimal integer.
- 9627 • An optional *precision* that gives the minimum number of digits to appear for the d, i, o, u, x
 9628 and X conversions; the number of digits to appear after the radix character for the e, E and f
 9629 conversions; the maximum number of significant digits for the g and G conversions; or the
 9630 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 behaviour is undefined.

- An optional l (ell) specifying that a following c conversion wide-character applies to a **wint_t** argument; an optional l specifying that a following s conversion wide-character applies to a **wchar_t** argument; an optional h specifying that a following d, i, o, u, x or X conversion wide-character applies to a type **short int** or type **unsigned short int** argument (the argument will have been promoted according to the integral promotions, and its value will be converted to type **short int** or **unsigned short int** before printing); an optional h specifying that a following n conversion wide-character applies to a pointer to a type **short int** argument; an optional l (ell) specifying that a following d, i, o, u, x or X conversion wide-character applies to a type **long int** or **unsigned long int** argument; an optional l (ell) specifying that a following n conversion wide-character applies to a pointer to a type **long int** argument; or an optional L specifying that a following e, E, f, g or G conversion wide-character applies to a type **long double** argument. If an h, l or L appears with any other conversion wide-character, the behaviour is undefined.
- 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. Arguments specifying field width, or precision, or both must 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) will be formatted with thousands' grouping wide-characters. For other conversions the behaviour is undefined. The non-monetary grouping wide-character is used.
- The result of the conversion will be left-justified within the field. The conversion will be right-justified if this flag is not specified.
- The result of a signed conversion will always begin with a sign (+ or –). The conversion will 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 will be prefixed to the result. This means that if the space and + flags both appear, the space flag will be ignored.

9677	#	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 will have 0x (or 0X) prefixed to it. For e, E, f, g or G conversions, the result will 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 will <i>not</i> be removed from the result as they normally are. For other conversions, the behaviour is undefined.
9678		
9679		
9680		
9681		
9682		
9683		
9684		
9685	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 will be ignored. For d, i, o, u, x and X conversions, if a precision is specified, the 0 flag will be ignored. If the 0 and ' flags both appear, the grouping wide-characters are inserted before zero padding. For other conversions, the behaviour is undefined.
9686		
9687		
9688		
9689		
9690		
9691	The conversion wide-characters and their meanings are:	
9692	d, i	The int argument is converted to a signed decimal in the style <code>[-]dddd</code> . The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will 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.
9693		
9694		
9695		
9696		
9697	o	The unsigned int argument is converted to unsigned octal format in the style <code>dddd</code> . The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will 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.
9698		
9699		
9700		
9701		
9702	u	The unsigned int argument is converted to unsigned decimal format in the style <code>dddd</code> . The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will 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.
9703		
9704		
9705		
9706		
9707	x	The unsigned int argument is converted to unsigned hexadecimal format in the style <code>dddd</code> ; 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 will 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.
9708		
9709		
9710		
9711		
9712	X	Behaves the same as the x conversion wide-character except that letters ABCDEF are used instead of abcdef.
9713		
9714	f	The double argument is converted to decimal notation in the style <code>[-]ddd.ddd</code> , 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.
9715		
9716		
9717		
9718		
9719	EX	The <code>fwprintf()</code> family of functions may make available wide-character string representations for infinity and NaN.
9720		
9721	e, E	The double argument is converted in the style <code>[-]d.ddde±dd</code> , 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
9722		
9723		

9724		as 6; if the precision is 0 and no # flag is present, no radix character appears. The value
9725		is rounded to the appropriate number of digits. The E conversion wide-character will
9726		produce a number with E instead of e introducing the exponent. The exponent always
9727		contains at least two digits. If the value is 0, the exponent is 0.
9728	EX	The <i>fwprintf()</i> family of functions may make available wide-character string
9729		representations for infinity and NaN.
9730	g, G	The double argument is converted in the style f or e (or in the style E in the case of a G
9731		conversion wide-character), with the precision specifying the number of significant
9732		digits. If an explicit precision is 0, it is taken as 1. The style used depends on the value
9733		converted; style e (or E) will be used only if the exponent resulting from such a
9734		conversion is less than -4 or greater than or equal to the precision. Trailing zeros are
9735		removed from the fractional portion of the result; a radix character appears only if it is
9736		followed by a digit.
9737	EX	The <i>fwprintf()</i> family of functions may make available wide-character string
9738		representations for infinity and NaN.
9739	c	If no l (ell) qualifier is present, the int argument is converted to a wide-character as if by
9740		calling the <i>btowc()</i> function and the resulting wide-character is written. Otherwise the
9741		wint_t argument is converted to wchar_t , and written.
9742	s	If no l (ell) qualifier is present, the argument must be a pointer to a character array
9743		containing a character sequence beginning in the initial shift state. Characters from the
9744		array are converted as if by repeated calls to the <i>mbrtowc()</i> function, with the
9745		conversion state described by an mbstate_t object initialised to zero before the first
9746		character is converted, and written up to (but not including) the terminating null wide-
9747		character. If the precision is specified, no more than that many wide-characters are
9748		written. If the precision is not specified or is greater than the size of the array, the array
9749		must contain a null wide-character.
9750		If an l (ell) qualifier is present, the argument must be a pointer to an array of type
9751		wchar_t . Wide characters from the array are written up to (but not including) a
9752		terminating null wide-character. If no precision is specified or is greater than the size
9753		of the array, the array must contain a null wide-character. If a precision is specified, no
9754		more than that many wide-characters are written.
9755	p	The argument must be a pointer to void . The value of the pointer is converted to a
9756		sequence of printable wide-characters, in an implementation-dependent manner.
9757	n	The argument must be a pointer to an integer into which is written the number of
9758		wide-characters written to the output so far by this call to one of the <i>fwprintf()</i>
9759		functions. No argument is converted.
9760	EX	C Same as lc .
9761		S Same as ls .
9762	%	Output a % wide-character; no argument is converted. The entire conversion
9763		specification must be %%. If a conversion specification does not match one of the above forms, the behaviour is undefined.
9764		
9765		In no case does a non-existent or small field width cause truncation of a field; if the result of a
9766		conversion is wider than the field width, the field is simply expanded to contain the conversion
9767		result. Characters generated by <i>fwprintf()</i> and <i>wprintf()</i> are printed as if <i>fputwc()</i> had been
9768		called.

9769 The *st_ctime* and *st_mtime* fields of the file will be marked for update between the call to a
 9770 successful execution of *fwprintf()* or *wprintf()* and the next successful completion of a call to
 9771 *fflush()* or *fclose()* on the same stream or a call to *exit()* or *abort()*.

9772 RETURN VALUE

9773 Upon successful completion, these functions return the number of wide-characters transmitted
 9774 excluding the terminating null wide-character in the case of *swprintf()* or a negative value if an
 9775 output error was encountered.

9776 ERRORS

9777 For the conditions under which *fwprintf()* and *wprintf()* will fail and may fail, refer to *fputwc()*.

9778 In addition, all forms of *fwprintf()* may fail if:

9779 EX [EILSEQ] A wide-character code that does not correspond to a valid character has been
 9780 detected.

9781 EX [EINVAL] There are insufficient arguments.

9782 In addition, *wprintf()* and *fwprintf()* may fail if:

9783 [ENOMEM] Insufficient storage space is available.

9784 EXAMPLES

9785 To print the language-independent date and time format, the following statement could be used:

```
9786 wprintf (format, weekday, month, day, hour, min);
```

9787 For American usage, *format* could be a pointer to the wide-character string:

```
9788 L"%s, %s %d, %d:%.2d\n"
```

9789 producing the message:

```
9790 Sunday, July 3, 10:02
```

9791 whereas for German usage, *format* could be a pointer to the wide-character string:

```
9792 L"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

9793 producing the message:

```
9794 Sonntag, 3. Juli, 10:02
```

9795 APPLICATION USAGE

9796 None.

9797 FUTURE DIRECTIONS

9798 None.

9799 SEE ALSO

9800 *btowc()*, *fputwc()*, *fwscanf()*, *setlocale()*, *mbrtowc()*, *<stdio.h>*, *<wchar.h>*, the XBD specification,
 9801 Chapter 5, Locale.

9802 CHANGE HISTORY

9803 First released in Issue 5.

9804 Include for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

9805 **NAME**

9806 fwrite — binary output

9807 **SYNOPSIS**

9808 #include <stdio.h>

```
9809     size_t fwrite(const void *ptr, size_t size, size_t nitems,
9810                  FILE *stream);
```

9811 **DESCRIPTION**

9812 The *fwrite()* function writes, from the array pointed to by *ptr*, up to *nitems* members whose size
 9813 is specified by *size*, to the stream pointed to by *stream*. The file-position indicator for the stream
 9814 (if defined) is advanced by the number of bytes successfully written. If an error occurs, the
 9815 resulting value of the file-position indicator for the stream is indeterminate.

9816 The *st_ctime* and *st_mtime* fields of the file will be marked for update between the successful
 9817 execution of *fwrite()* and the next successful completion of a call to *fflush()* or *fclose()* on the
 9818 same stream or a call to *exit()* or *abort()*.

9819 **RETURN VALUE**

9820 The *fwrite()* function returns the number of members successfully written, which may be less
 9821 than *nitems* if a write error is encountered. If *size* or *nitems* is 0, *fwrite()* returns 0 and the state of
 9822 the stream remains unchanged. Otherwise, if a write error occurs, the error indicator for the
 9823 stream is set and *errno* is set to indicate the error.

9824 **ERRORS**9825 Refer to *fputc()*.9826 **EXAMPLES**

9827 None.

9828 **APPLICATION USAGE**

9829 Because of possible differences in member length and byte ordering, files written using *fwrite()*
 9830 are application-dependent, and possibly cannot be read using *fread()* by a different application
 9831 or by the same application on a different processor.

9832 **FUTURE DIRECTIONS**

9833 None.

9834 **SEE ALSO**9835 *ferror()*, *fopen()*, *printf()*, *putc()*, *puts()*, *write()*, <stdio.h>.9836 **CHANGE HISTORY**

9837 First released in Issue 1.

9838 Derived from Issue 1 of the SVID.

9839 **Issue 4**

9840 The following change is incorporated for alignment with the ISO C standard:

- 9841 • The type of argument *ptr* is changed from **void*** to **const void***.

9842 Another change is incorporated as follows:

- 9843 • In the DESCRIPTION, the text is changed to make it clear that the function advances the
 9844 file-position indicator by the number of bytes successfully written rather than the number of
 9845 characters, which could include multi-byte sequences.

9846 NAME

9847 fwscanf, wscanf, swscanf — convert formatted wide-character input

9848 SYNOPSIS

9849 #include <stdio.h>

9850 #include <wchar.h>

9851 int fwscanf(FILE *stream, const wchar_t *format, ...);

9852 int wscanf(const wchar_t *format, ...);

9853 int swscanf(const wchar_t *s, const wchar_t *format, ...);

9854 DESCRIPTION

9855 The *fwscanf()* function reads from the named input *stream*. The *wscanf()* function reads from the
 9856 standard input stream *stdin*. The *swscanf()* function reads from the wide-character string *s*.
 9857 Each function reads wide-characters, interprets them according to a format, and stores the
 9858 results in its arguments. Each expects, as arguments, a control wide-character string *format*
 9859 described below, and a set of *pointer* arguments indicating where the converted input should be
 9860 stored. The result is undefined if there are insufficient arguments for the format. If the format is
 9861 exhausted while arguments remain, the excess arguments are evaluated but are otherwise
 9862 ignored.

9863 EX Conversions can be applied to the *nth* argument after the *format* in the argument list, rather than
 9864 to the next unused argument. In this case, the conversion wide-character % (see below) is
 9865 replaced by the sequence %*n*\$, where *n* is a decimal integer in the range [1, {NL_ARGMAX}].
 9866 This feature provides for the definition of format wide-character strings that select arguments in
 9867 an order appropriate to specific languages. In format wide-character strings containing the %*n*\$
 9868 form of conversion specifications, it is unspecified whether numbered arguments in the
 9869 argument list can be referenced from the format wide-character string more than once.

9870 The *format* can contain either form of a conversion specification, that is, % or %*n*\$, but the two
 9871 forms cannot normally be mixed within a single *format* wide-character string. The only
 9872 exception to this is that %% or %* can be mixed with the %*n*\$ form.

9873 The *fwscanf()* function in all its forms allows for detection of a language-dependent radix
 9874 character in the input string, encoded as a wide-character value. The radix character is defined
 9875 in the program's locale (category LC_NUMERIC). In the POSIX locale, or in a locale where the
 9876 radix character is not defined, the radix character defaults to a period (.).

9877 The format is a wide-character string composed of zero or more directives. Each directive is
 9878 composed of one of the following: one or more white-space wide-characters (space, tab,
 9879 newline, vertical-tab or form-feed characters); an ordinary wide-character (neither % nor a
 9880 white-space character); or a conversion specification. Each conversion specification is
 9881 EX introduced by a % or the sequence %*n*\$ after which the following appear in sequence:

- 9882 • An optional assignment-suppressing character *.
- 9883 • An optional non-zero decimal integer that specifies the maximum field width.
- 9884 • An optional size modifier h, l (ell) or L indicating the size of the receiving object. The
 9885 conversion wide-characters c, s and [must be precede by l (ell) if the corresponding
 9886 argument is a pointer to **wchar_t** rather than a pointer to a character type. The conversion
 9887 wide-characters d, i and n must be preceded by h if the corresponding argument is a pointer
 9888 to **short int** rather than a pointer to **int**, or by l (ell) if it is a pointer to **long int**. Similarly, the
 9889 conversion wide-characters o, u and x must be preceded by h if the corresponding argument
 9890 is a pointer to **unsigned short int** rather than a pointer to **unsigned int**, or by l (ell) if it is a
 9891 pointer to **unsigned long int**. The conversion wide-characters e, f and g must be preceded by
 9892 l (ell) if the corresponding argument is a pointer to **double** rather than a pointer to **float**, or

by L if it is a pointer to **long double**. If an h, l (ell) or L appears with any other conversion wide-character, the behaviour is undefined.

- A conversion wide-character that specifies the type of conversion to be applied. The valid 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 returns. 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.

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 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 behaviour is undefined.

The following conversion wide-characters are valid:

d	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of <i>wcstol()</i> with the value 10 for the <i>base</i> argument. In the absence of a size modifier, the corresponding argument must be a pointer to int .
i	Matches an optionally signed integer, whose format is the same as expected for the subject sequence of <i>wcstol()</i> with 0 for the <i>base</i> argument. In the absence of a size modifier, the corresponding argument must be a pointer to int .
o	Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of <i>wcstoul()</i> with the value 8 for the <i>base</i> argument. In the absence of a size modifier, the corresponding argument must be a pointer to unsigned int .

9939	u	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of <i>wcstoul()</i> with the value 10 for the <i>base</i> argument. In the absence of a size modifier, the corresponding argument must be a pointer to unsigned int .
9940		
9941		
9942		
9943	x	Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of <i>wcstoul()</i> with the value 16 for the <i>base</i> argument. In the absence of a size modifier, the corresponding argument must be a pointer to unsigned int .
9944		
9945		
9946		
9947	e, f, g	Matches an optionally signed floating-point number, whose format is the same as expected for the subject sequence of <i>wcstod()</i> . In the absence of a size modifier, the corresponding argument must be a pointer to float .
9948		
9949		
9950		If the <i>fwprintf()</i> family of functions generates character string representations for infinity and NaN (a 7858 symbolic entity encoded in floating-point format) to support the ANSI/IEEE Std 754:1985 standard, the <i>fwscanf()</i> family of functions will recognise them as input.
9951		
9952		
9953		
9954	s	Matches a sequence of non white-space wide-characters. If no <i>l</i> (ell) qualifier is present, characters from the input field are converted as if by repeated calls to the <i>wcrtomb()</i> function, with the conversion state described by an mbstate_t object initialised to zero before the first wide-character is converted. The corresponding argument must be a pointer to a character array large enough to accept the sequence and the terminating null character, which will be added automatically.
9955		
9956		
9957		
9958		
9959		
9960		Otherwise, the corresponding argument must be a pointer to an array of wchar_t large enough to accept the sequence and the terminating null wide-character, which will be added automatically.
9961		
9962		
9963	[Matches a non-empty sequence of wide-characters from a set of expected wide-characters (the <i>scanset</i>). If no <i>l</i> (ell) qualifier is present, wide-characters from the input field are converted as if by repeated calls to the <i>wcrtomb()</i> function, with the conversion state described by an mbstate_t object initialised to zero before the first wide-character is converted. The corresponding argument must be a pointer to a character array large enough to accept the sequence and the terminating null character, which will be added automatically.
9964		
9965		
9966		
9967		
9968		
9969		
9970		If an <i>l</i> (ell) qualifier is present, the corresponding argument must be a pointer to an array of wchar_t large enough to accept the sequence and the terminating null wide-character, which will be added automatically.
9971		
9972		
9973		The conversion specification includes all subsequent <i>widw</i> characters in the <i>format</i> string up to and including the matching right square bracket (]). The wide-characters between the square brackets (the <i>scanlist</i>) comprise the <i>scanset</i> , unless the wide-character after the left square bracket is a circumflex (^), in which case the <i>scanset</i> contains all wide-characters that do not appear in the <i>scanlist</i> between the circumflex and the right square bracket. If the conversion specification begins with [] or [^], the right square bracket is included in the <i>scanlist</i> 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 <i>scanlist</i> and is not the first wide-character, nor the second where the first wide-character is a ^, nor the last wide-character, the behaviour is implementation-dependent.
9974		
9975		
9976		
9977		
9978		
9979		
9980		
9981		
9982		
9983		
9984		
9985	c	Matches a sequence of wide-characters of the number specified by the field width (1 if no field width is present in the conversion specification). If no <i>l</i> (ell) qualifier is present,
9986		

9987		wide-characters from the input field are converted as if by repeated calls to the
9988		<i>wcrtomb()</i> function, with the conversion state described by an mbstate_t object
9989		initialised to zero before the first wide-character is converted. The corresponding
9990		argument must be a pointer to a character array large enough to accept the sequence.
9991		No null character is added.
9992		Otherwise, the corresponding argument must be a pointer to an array of wchar_t large
9993		enough to accept the sequence. No null wide-character is added.
9994	p	Matches an implementation-dependent set of sequences, which must be the same as
9995		the set of sequences that is produced by the %p conversion of the corresponding
9996		<i>fwprintf()</i> functions. The corresponding argument must be a pointer to a pointer to
9997		void . The interpretation of the input item is implementation-dependent. If the input
9998		item is a value converted earlier during the same program execution, the pointer that
9999		results will compare equal to that value; otherwise the behaviour of the %p conversion
10000		is undefined.
10001	n	No input is consumed. The corresponding argument must be a pointer to the integer
10002		into which is to be written the number of wide-characters read from the input so far by
10003		this call to the <i>fwscanf()</i> functions. Execution of a %n conversion specification does not
10004		increment the assignment count returned at the completion of execution of the
10005		function.
10006	EX	C Same as lc .
10007		S Same as ls .
10008	%	Matches a single %; no conversion or assignment occurs. The complete conversion
10009		specification must be %%. If a conversion specification is invalid, the behaviour is undefined.
10010		
10011		The conversion characters E, G and X are also valid and behave the same as, respectively, e, g
10012		and x.
10013		If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before
10014		any wide-characters matching the current conversion specification (except for %n) have been
10015		read (other than leading white-space, where permitted), execution of the current conversion
10016		specification terminates with an input failure. Otherwise, unless execution of the current
10017		conversion specification is terminated with a matching failure, execution of the following
10018		conversion specification (if any) is terminated with an input failure.
10019		Reaching the end of the string in <i>swscanf()</i> is equivalent to encountering end-of-file for <i>fwscanf()</i> .
10020		If conversion terminates on a conflicting input, the offending input is left unread in the input.
10021		Any trailing white space (including newline) is left unread unless matched by a conversion
10022		specification. The success of literal matches and suppressed assignments is only directly
10023		determinable via the %n conversion specification.
10024		The <i>fwscanf()</i> and <i>wscanf()</i> functions may mark the <i>st_atime</i> field of the file associated with
10025		<i>stream</i> for update. The <i>st_atime</i> field will be marked for update by the first successful execution
10026		of <i>fgetc()</i> , <i>fgetwc()</i> , <i>fgets()</i> , <i>fgetws()</i> , <i>fread()</i> , <i>getc()</i> , <i>getwc()</i> , <i>getchar()</i> , <i>getwchar()</i> , <i>gets()</i> , <i>fscanf()</i>
10027		or <i>fwscanf()</i> using <i>stream</i> that returns data not supplied by a prior call to <i>ungetc()</i> .
10028	RETURN VALUE	
10029		Upon successful completion, these functions return the number of successfully matched and
10030		assigned input items; this number can be 0 in the event of an early matching failure. If the input
10031		ends before the first matching failure or conversion, EOF is returned. If a read error occurs the
10032		error indicator for the stream is set, EOF is returned, and <i>errno</i> is set to indicate the error.

10033 **ERRORS**

10034 For the conditions under which the *fwscanf()* functions will fail and may fail, refer to *fgetwc()*.

10035 In addition, *fwscanf()* may fail if:

10036 EX **[EILSEQ]** Input byte sequence does not form a valid character.

10037 EX **[EINVAL]** There are insufficient arguments.

10038 **EXAMPLES**

10039 The call:

```
10040     int i, n; float x; char name[50];
10041     n = wscanf(L"%d%f%s", &i, &x, name);
```

10042 with the input line:

```
10043     25 54.32E-1 Hamster
```

10044 will assign to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* will contain the string
10045 Hamster.

10046 The call:

```
10047     int i; float x; char name[50];
10048     (void) wscanf(L"%2d%f*d %[0123456789]", &i, &x, name);
```

10049 with input:

```
10050     56789 0123 56a72
```

10051 will assign 56 to *i*, 789.0 to *x*, skip 0123, and place the string 56\0 in *name*. The next call to
10052 *getchar()* will return the character a.

10053 **APPLICATION USAGE**

10054 In format strings containing the % form of conversion specifications, each argument in the
10055 argument list is used exactly once.

10056 **FUTURE DIRECTIONS**

10057 None.

10058 **SEE ALSO**

10059 *getwc()*, *fwprintf()*, *setlocale()*, *wcstod()*, *wcstol()*, *wcstoul()*, *wcrtomb()*, **<langinfo.h>**, **<stdio.h>**,
10060 **<wchar.h>**, the XBD specification, **Chapter 5, Locale**.

10061 **CHANGE HISTORY**

10062 First released in Issue 5.

10063 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

10064 **NAME**10065 gamma, signgam — log gamma function (**LEGACY**)10066 **SYNOPSIS**

10067 EX #include <math.h>

10068 double gamma(double x);

10069 extern int signgam;

10070

10071 **DESCRIPTION**10072 The *gamma()* function performs identically to *lgamma()*, including the use of *signgam*.

10073 This interface need not be reentrant.

10074 **RETURN VALUE**10075 Return to *lgamma()*.10076 **ERRORS**

10077 None.

10078 **EXAMPLES**

10079 None.

10080 **APPLICATION USAGE**10081 This interface is functionally equivalent to *lgamma()*.10082 **FUTURE DIRECTIONS**

10083 None.

10084 **SEE ALSO**10085 Return to *lgamma()*.10086 **CHANGE HISTORY**

10087 First released in Issue 1.

10088 Derived from Issue 1 of the SVID.

10089 **Issue 4**

10090 The following changes are incorporated in this issue:

- 10091 • This interface is marked TO BE WITHDRAWN, as it is functionally equivalent to *lgamma()*.
- 10092 • The DESCRIPTION is changed to refer to *lgamma()*.
- 10093 • The APPLICATION USAGE section is added.

10094 **Issue 5**

10095 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

10096 Marked LEGACY.

10097 **NAME**

10098 gcvt — convert a floating-point number to a string

10099 **SYNOPSIS**

10100 EX #include <stdlib.h>

10101 char *gcvt(double *value*, int *ndigit*, char **buf*);

10102

10103 **DESCRIPTION**10104 Refer to *ecvt()*.10105 **CHANGE HISTORY**

10106 First released in Issue 4, Version 2.

10107 **Issue 5**

10108 Moved from X/OPEN UNIX extension to BASE.

10109 **NAME**

10110 getc — get a byte from a stream

10111 **SYNOPSIS**

10112 #include <stdio.h>

10113 int getc(FILE **stream*);10114 **DESCRIPTION**

10115 The *getc()* function is equivalent to *fgetc()*, except that if it is implemented as a macro it may
10116 evaluate *stream* more than once, so the argument should never be an expression with side effects.

10117 **RETURN VALUE**10118 Refer to *fgetc()*.10119 **ERRORS**10120 Refer to *fgetc()*.10121 **EXAMPLES**

10122 None.

10123 **APPLICATION USAGE**

10124 If the integer value returned by *getc()* is stored into a variable of type **char** and then compared
10125 against the integer constant EOF, the comparison may never succeed, because sign-extension of
10126 a variable of type **char** on widening to integer is implementation-dependent.

10127 Because it may be implemented as a macro, *getc()* may treat incorrectly a *stream* argument with
10128 side effects. In particular, *getc(*f++)* will not necessarily work as expected. Therefore, use of this
10129 function should be preceded by “#undef getc” in such situations; *fgetc()* could also be used.

10130 **FUTURE DIRECTIONS**

10131 None.

10132 **SEE ALSO**10133 *fgetc()*, <stdio.h>.10134 **CHANGE HISTORY**

10135 First released in Issue 1.

10136 Derived from Issue 1 of the SVID.

10137 **Issue 4**

10138 The following changes are incorporated in this issue:

- 10139 • The words “a character variable” are replaced by “a variable of type **char**”, to emphasise the
10140 fact that this interface deals with byte values.
- 10141 • The APPLICATION USAGE section now states that the use of this function is not
10142 recommended.

10143 **NAME**10144 `getchar` — get a byte from a *stdin* stream10145 **SYNOPSIS**10146 `#include <stdio.h>`10147 `int getchar(void);`10148 **DESCRIPTION**10149 The *getchar()* function is equivalent to *getc(stdin)*.10150 **RETURN VALUE**10151 Refer to *fgetc()*.10152 **ERRORS**10153 Refer to *fgetc()*.10154 **EXAMPLES**

10155 None.

10156 **APPLICATION USAGE**

10157 If the integer value returned by *getchar()* is stored into a variable of type **char** and then
10158 compared against the integer constant EOF, the comparison may never succeed, because sign-
10159 extension of a variable of type **char** on widening to integer is implementation-dependent.

10160 **FUTURE DIRECTIONS**

10161 None.

10162 **SEE ALSO**10163 *getc()*, **<stdio.h>**.10164 **CHANGE HISTORY**

10165 First released in Issue 1.

10166 Derived from Issue 1 of the SVID.

10167 **Issue 4**

10168 The following change is incorporated for alignment with the ISO C standard:

- 10169 • The argument list is explicitly defined as **void**.

10170 Another change is incorporated as follows:

- 10171 • The words “a character variable” are replaced by “a variable of type **char**”, to emphasise the
10172 fact that this interface deals in byte values.

10173 **NAME**

10174 `getc_unlocked`, `getchar_unlocked`, `putc_unlocked`, `putchar_unlocked` — stdio with explicit client
10175 locking

10176 **SYNOPSIS**

10177 `#include <stdio.h>`

10178 `int getc_unlocked(FILE *stream);`
10179 `int getchar_unlocked(void);`
10180 `int putc_unlocked(int c, FILE *stream);`
10181 `int putchar_unlocked(int c);`

10182 **DESCRIPTION**

10183 Versions of the functions `getc()`, `getchar()`, `putc()`, and `putchar()` respectively named
10184 `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`, and `putchar_unlocked()` are provided which
10185 are functionally identical to the original versions with the exception that they are not required to
10186 be implemented in a thread-safe manner. They may only safely be used within a scope
10187 protected by `flockfile()` (or `ftrylockfile()`) and `funlockfile()`. These functions may safely be used in
10188 a multi-threaded program if and only if they are called while the invoking thread owns the
10189 (**FILE***) object, as is the case after a successful call of the `flockfile()` or `ftrylockfile()` functions.

10190 **RETURN VALUE**

10191 See `getc()`, `getchar()`, `putc()`, and `putchar()`.

10192 **ERRORS**

10193 No errors are defined.

10194 **EXAMPLES**

10195 None.

10196 **APPLICATION USAGE**

10197 None.

10198 **FUTURE DIRECTIONS**

10199 None.

10200 **SEE ALSO**

10201 `getc()`, `getchar()`, `putc()`, `putchar()`, `<stdio.h>`.

10202 **CHANGE HISTORY**

10203 First released in Issue 5.

10204 Included for alignment with the POSIX Threads Extension.

10205 **NAME**

10206 getcontext, setcontext — get and set current user context

10207 **SYNOPSIS**

10208 EX #include <ucontext.h>

10209 int getcontext(ucontext_t *ucp);

10210 int setcontext(const ucontext_t *ucp);

10211

10212 **DESCRIPTION**

10213 The *getcontext()* function initialises the structure pointed to by *ucp* to the current user context of
 10214 the calling thread. The **ucontext_t** type that *ucp* points to defines the user context and includes
 10215 the contents of the calling thread's machine registers, the signal mask, and the current execution
 10216 stack.

10217 The *setcontext()* function restores the user context pointed to by *ucp*. A successful call to
 10218 *setcontext()* does not return; program execution resumes at the point specified by the *ucp*
 10219 argument passed to *setcontext()*. The *ucp* argument should be created either by a prior call to
 10220 *getcontext()* or *makecontext()*, or by being passed as an argument to a signal handler. If the *ucp*
 10221 argument was created with *getcontext()*, program execution continues as if the corresponding
 10222 call of *getcontext()* had just returned. If the *ucp* argument was created with *makecontext()*,
 10223 program execution continues with the function passed to *makecontext()*. When that function
 10224 returns, the thread continues as if after a call to *setcontext()* with the *ucp* argument that was
 10225 input to *makecontext()*. If the **uc_link** member of the **ucontext_t** structure pointed to by the *ucp*
 10226 argument is equal to 0, then this context is the main context, and the thread will exit when this
 10227 context returns. The effects of passing a *ucp* argument obtained from any other source are
 10228 unspecified.

10229 **RETURN VALUE**

10230 On successful completion, *setcontext()* does not return and *getcontext()* returns 0. Otherwise, a
 10231 value of -1 is returned.

10232 **ERRORS**

10233 No errors are defined.

10234 **EXAMPLES**

10235 None.

10236 **APPLICATION USAGE**

10237 When a signal handler is executed, the current user context is saved and a new context is
 10238 created. If the thread leaves the signal handler via *longjmp()*, then it is unspecified whether the
 10239 context at the time of the corresponding *setjmp()* call is restored and thus whether future calls to
 10240 *getcontext()* will provide an accurate representation of the current context, since the context
 10241 restored by *longjmp()* does not necessarily contain all the information that *setcontext()* requires.
 10242 Signal handlers should use *siglongjmp()* or *setcontext()* instead.

10243 Portable applications should not modify or access the **uc_mcontext** member of **ucontext_t**. A
 10244 portable application cannot assume that context includes any process-wide static data, possibly
 10245 including *errno*. Users manipulating contexts should take care to handle these explicitly when
 10246 required.

10247 Use of contexts to create alternate stacks is not defined by this specification.

10248 **FUTURE DIRECTIONS**

10249 None.

10250 **SEE ALSO**

10251 *bsd_signal()*, *makecontext()*, *setjmp()*, *sigaction()*, *sigaltstack()*, *sigprocmask()*, *sigsetjmp()*,
10252 ucontext.h.

10253 **CHANGE HISTORY**

10254 First released in Issue 4, Version 2.

10255 **Issue 5**

10256 Moved from X/OPEN UNIX extension to BASE.

10257 The following sentence was removed from the DESCRIPTION: “If the *ucp* argument was passed
10258 to a signal handler, program execution continues with the program instruction following the
10259 instruction interrupted by the signal.”

10260 **NAME**

10261 getcwd — get the pathname of the current working directory

10262 **SYNOPSIS**

10263 #include <unistd.h>

10264 char *getcwd(char *buf, size_t size);

10265 **DESCRIPTION**

10266 The *getcwd()* function places an absolute pathname of the current working directory in the array pointed to by *buf*, and returns *buf*. The *size* argument is the size in bytes of the character array pointed to by the *buf* argument. If *buf* is a null pointer, the behaviour of *getcwd()* is undefined.

10269 **RETURN VALUE**

10270 Upon successful completion, *getcwd()* returns the *buf* argument. Otherwise, *getcwd()* returns a null pointer and sets *errno* to indicate the error. The contents of the array pointed to by *buf* is then undefined.

10273 **ERRORS**10274 The *getcwd()* function will fail if:10275 [EINVAL] The *size* argument is 0.

10276 [ERANGE] The size argument is greater than 0, but is smaller than the length of the
10277 pathname + 1.

10278 The *getcwd()* function may fail if:

10279 [EACCES] Read or search permission was denied for a component of the pathname.

10280 EX [ENOMEM] Insufficient storage space is available.

10281 **EXAMPLES**

10282 None.

10283 **APPLICATION USAGE**

10284 If *buf* is a null pointer, *getcwd()* may obtain *size* bytes of memory using *malloc()*. In this case, the pointer returned by *getcwd()* may be used as the argument in a subsequent call to *free()*.
10286 Invoking *getcwd()* with *buf* as a null pointer is not recommended.

10287 **FUTURE DIRECTIONS**

10288 None.

10289 **SEE ALSO**10290 *malloc()*, <unistd.h>.10291 **CHANGE HISTORY**

10292 First released in Issue 1.

10293 Derived from Issue 1 of the SVID.

10294 **Issue 4**

10295 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 10296 • The DESCRIPTION is changed to indicate that the effects of passing a null pointer in *buf* are
10297 undefined.

10298 Other changes are incorporated as follows:

- 10299 • The <**unistd.h**> header is added to the SYNOPSIS section.
10300 • The [ENOMEM] error is marked as an extension.
10301 • The words “as this functionality may be subject to withdrawal” have been deleted from the
10302 end of the last sentence in the APPLICATION USAGE section.

10303 **NAME**

10304 getdate — convert user format date and time

10305 **SYNOPSIS**

10306 EX #include <time.h>

10307 struct tm *getdate(const char *string);

10308

10309 **DESCRIPTION**10310 The *getdate()* function converts a string representation of a date or time into a broken-down
10311 time.10312 The external variable or macro *getdate_err* is used by *getdate()* to return error values.10313 Templates are used to parse and interpret the input string. The templates are contained in a text
10314 file identified by the environment variable *DATMSK*. The *DATMSK* variable should be set to
10315 indicate the full pathname of the file that contains the templates. The first line in the template
10316 that matches the input specification is used for interpretation and conversion into the internal
10317 time format.

10318 The following field descriptors are supported:

10319 %% same as %

10320 %a abbreviated weekday name

10321 %A full weekday name

10322 %b abbreviated month name

10323 %B full month name

10324 %c locale's appropriate date and time representation

10325 %C century number (00-99; leading zeros are permitted but not required)

10326 %d day of month (01-31; the leading 0 is optional)

10327 %D date as %m/%d/%y

10328 %e same as %d

10329 %h abbreviated month name

10330 %H hour (00-23)

10331 %I hour (01-12)

10332 %m month number (01-12)

10333 %M minute (00-59)

10334 %n same as new line

10335 %p locale's equivalent of either AM or PM

10336 %r The locale's appropriate representation of time in AM and PM notation. In the POSIX
10337 locale, this is equivalent to %I:%M:%S %p

10338 %R time as %H:%M

10339 %S seconds (00-61). Leap seconds are allowed but are not predictable through use of
10340 algorithms.

10341	%t	same as tab
10342	%T	time as %H:%M:%S
10343	%w	weekday number (Sunday = 0 - 6)
10344	%x	locale's appropriate date representation
10345	%X	locale's appropriate time representation
10346	%y	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).
10347		
10348		
10349	%Y	year as ccyy (for example, 1994)
10350	%Z	time zone name or no characters if no time zone exists. If the time zone supplied by %Z is not the time zone that <i>getdate()</i> expects, an invalid input specification error will result. The <i>getdate()</i> function calculates an expected time zone based on information supplied to the function (such as the hour, day, and month).
10351		
10352		
10353		
10354	The match between the template and input specification performed by <i>getdate()</i> is case insensitive.	
10355		
10356	The month and weekday names can consist of any combination of upper and lower case letters.	
10357	The process can request that the input date or time specification be in a specific language by setting the LC_TIME category (see <i>setlocale()</i>).	
10358		
10359	Leading 0's are not necessary for the descriptors that allow leading 0's. However, at most two digits are allowed for those descriptors, including leading 0's. Extra whitespace in either the template file or in <i>string</i> is ignored.	
10360		
10361		
10362	The field descriptors %c, %x, and %X will not be supported if they include unsupported field descriptors.	
10363		
10364	The following rules apply for converting the input specification into the internal format:	
10365	• If %Z is being scanned, then <i>getdate()</i> initialises the broken-down time to be the current time in the scanned time zone. Otherwise it initialises the broken-down time based on the current local time as if <i>localtime()</i> had been called.	
10366		
10367		
10368	• If only the weekday is given, today is assumed if the given day is equal to the current day and next week if it is less,	
10369		
10370	• If only the month is given, the current month is assumed if the given month is equal to the current month and next year if it is less and no year is given (the first day of month is assumed if no day is given),	
10371		
10372		
10373	• If no hour, minute and second are given the current hour, minute and second are assumed,	
10374	• If no date is given, today is assumed if the given hour is greater than the current hour and tomorrow is assumed if it is less.	
10375		
10376	If a field descriptor specification in the DATEMSK file does not correspond to one of the field descriptors above, the behaviour is unspecified.	
10377		
10378	This interface need not be reentrant.	
10379	RETURN VALUE	
10380	Upon successful completion, <i>getdate()</i> returns a pointer to a struct tm . Otherwise, it returns a null pointer and <i>getdate_err</i> is set to indicate the error.	
10381		

10382 **ERRORS**

10383 The *getdate()* function will fail in the following cases, setting *getdate_err* to the value shown in
 10384 the list below. Any changes to *errno* are unspecified.

- 10385 1 The *DATETIME* environment variable is null or undefined.
- 10386 2 The template file cannot be opened for reading.
- 10387 3 Failed to get file status information.
- 10388 4 The template file is not a regular file.
- 10389 5 An I/O error is encountered while reading the template file.
- 10390 6 Memory allocation failed (not enough memory available).
- 10391 7 There is no line in the template that matches the input.
- 10392 8 Invalid input specification. For example, February 31; or a time is specified that can not be
 10393 represented in a **time_t** (representing the time in seconds since 00:00:00 UTC, January 1,
 10394 1970).

10395 **EXAMPLES**

10396 Example 1:

10397 The following example shows the possible contents of a template:

```
10398 %m
10399 %A %B %d, %Y, %H:%M:%S
10400 %A
10401 %B
10402 %m/%d/%y %I %p
10403 %d,%m,%Y %H:%M
10404 at %A the %dst of %B in %Y
10405 run job at %I %p,%B %dnd
10406 %A den %d. %B %Y %H.%M Uhr
```

10407 Example 2:

10408 The following are examples of valid input specifications for the template in Example 1:

```
10409 getdate("10/1/87 4 PM");
10410 getdate("Friday");
10411 getdate("Friday September 18, 1987, 10:30:30");
10412 getdate("24,9,1986 10:30");
10413 getdate("at monday the 1st of december in 1986");
10414 getdate("run job at 3 PM, december 2nd");
```

10415 If the *LC_TIME* category is set to a German locale that includes *freitag* as a weekday name
 10416 and *oktober* as a month name, the following would be valid:

```
10417 getdate("freitag den 10. oktober 1986 10.30 Uhr");
```

Example 3:

The following examples 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

Example 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 specification 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).

FUTURE DIRECTIONS

None.

SEE ALSO

ctime(), *ctype()*, *localtime()*, *setlocale()*, *strftime()*, *times()*, **<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.

- 10463 A note indicating that this interface need not be reentrant is added to the DESCRIPTION. |
- 10464 The %R specifier is changed to follow historical practise.

10465 **NAME**10466 getdtablesize — get the file descriptor table size (**LEGACY**)10467 **SYNOPSIS**10468 EX `#include <unistd.h>`10469 `int getdtablesize(void);`

10470

10471 **DESCRIPTION**10472 The *getdtablesize()* function is equivalent to *getrlimit()* with the `RLIMIT_NOFILE` option.

10473 This interface need not be reentrant.

10474 **RETURN VALUE**10475 The *getdtablesize()* function returns the current soft limit as if obtained from a call to *getrlimit()*10476 with the `RLIMIT_NOFILE` option.10477 **ERRORS**

10478 No errors are defined.

10479 **EXAMPLES**

10480 None.

10481 **APPLICATION USAGE**10482 There is no direct relationship between the value returned by *getdtablesize()* and `{OPEN_MAX}`
10483 defined in `<limits.h>`.10484 The *getrlimit()* function returns a value of type `rlim_t`. This interface, returning an `int`, may have
10485 problems representing appropriate values in the future. Applications should use the *getrlimit()*
10486 function instead.10487 **FUTURE DIRECTIONS**

10488 None.

10489 **SEE ALSO**10490 *close()*, *getrlimit()*, *open()*, *select()*, *setrlimit()*, `<limits.h>`, `<unistd.h>`.10491 **CHANGE HISTORY**

10492 First released in Issue 4, Version 2.

10493 **Issue 5**

10494 Moved from X/OPEN UNIX extension to BASE

10495 A new paragraph is added to the APPLICATION USAGE section giving reasons why the
10496 interface may be withdrawn in a future issue.

10497 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

10498 Marked LEGACY.

10499 **NAME**

10500 getegid — get the effective group ID

10501 **SYNOPSIS**

10502 OH #include <sys/types.h>

10503 #include <unistd.h>

10504 gid_t getegid(void);

10505 **DESCRIPTION**10506 The *getegid()* function returns the effective group ID of the calling process.10507 **RETURN VALUE**10508 The *getegid()* function is always successful and no return value is reserved to indicate an error.10509 **ERRORS**

10510 No errors are defined.

10511 **EXAMPLES**

10512 None.

10513 **APPLICATION USAGE**

10514 None.

10515 **FUTURE DIRECTIONS**

10516 None.

10517 **SEE ALSO**10518 *getgid()*, *setgid()*, <sys/types.h>, <unistd.h>.10519 **CHANGE HISTORY**

10520 First released in Issue 1.

10521 Derived from Issue 1 of the SVID.

10522 **Issue 4**

10523 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 10524
- The argument list is explicitly defined as **void**.

10525 Other changes are incorporated as follows:

- 10526
- 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.

10529 **NAME**

10530 getenv — get value of an environment variable

10531 **SYNOPSIS**

10532 #include <stdlib.h>

10533 char *getenv(const char *name);

10534 **DESCRIPTION**

10535 The *getenv()* function searches the environment list for a string of the form "*name=value*", and
 10536 returns a pointer to a string containing the *value* for the specified name. If the specified name
 10537 cannot be found, a null pointer is returned. The string pointed to must not be modified by the
 10538 EX application, but may be overwritten by a subsequent call to *getenv()* or *putenv()* but will not be
 10539 overwritten by a call to any other function in this document.

10540 This interface need not be reentrant.

10541 **RETURN VALUE**

10542 Upon successful completion, *getenv()* returns a pointer to a string containing the *value* for the
 10543 specified *name*. If the specified name cannot be found a null pointer is returned.

10544 The return value from *getenv()* may point to static data which may be overwritten by
 10545 subsequent calls to *getenv()* or *putenv()*.

10546 **ERRORS**

10547 No errors are defined.

10548 **EXAMPLES**

10549 None.

10550 **APPLICATION USAGE**

10551 None.

10552 **FUTURE DIRECTIONS**

10553 None.

10554 **SEE ALSO**10555 *exec*, *putenv()*, <stdlib.h>, the XBD specification, **Chapter 6, Environment Variables**.10556 **CHANGE HISTORY**

10557 First released in Issue 1.

10558 Derived from Issue 1 of the SVID.

10559 **Issue 4**

10560 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 10561 • The type of argument *name* is changed from **char *** to **const char ***.

10562 Other changes are incorporated as follows:

- 10563 • The DESCRIPTION is updated to indicate that the return string (a) must not be modified by
 10564 an application, (b) may be overwritten by subsequent calls to *getenv()* or *putenv()*, and (c)
 10565 will not be overwritten by calls to other XSI system interfaces. A reference to *putenv()* has
 10566 also been added to the APPLICATION USAGE section.

10567 **Issue 5**

10568 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
 10569 VALUE section.

10570 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

10571 **NAME**

10572 geteuid — get the effective user ID

10573 **SYNOPSIS**

10574 OH #include <sys/types.h>

10575 #include <unistd.h>

10576 uid_t geteuid(void);

10577 **DESCRIPTION**10578 The *geteuid()* function returns the effective user ID of the calling process.10579 **RETURN VALUE**10580 The *geteuid()* function is always successful and no return value is reserved to indicate an error.10581 **ERRORS**

10582 No errors are defined.

10583 **EXAMPLES**

10584 None.

10585 **APPLICATION USAGE**

10586 None.

10587 **FUTURE DIRECTIONS**

10588 None.

10589 **SEE ALSO**10590 *getuid()*, *setuid()*, <sys/types.h>, <unistd.h>.10591 **CHANGE HISTORY**

10592 First released in Issue 1.

10593 Derived from Issue 1 of the SVID.

10594 **Issue 4**

10595 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 10596
- The argument list is explicitly defined as **void**.

10597 Other changes are incorporated as follows:

- 10598
- 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.

10601 **NAME**

10602 getgid — get the real group ID

10603 **SYNOPSIS**

10604 OH #include <sys/types.h>

10605 #include <unistd.h>

10606 gid_t getgid(void);

10607 **DESCRIPTION**10608 The *getgid()* function returns the real group ID of the calling process.10609 **RETURN VALUE**10610 The *getgid()* function is always successful and no return value is reserved to indicate an error.10611 **ERRORS**

10612 No errors are defined.

10613 **EXAMPLES**

10614 None.

10615 **APPLICATION USAGE**

10616 None.

10617 **FUTURE DIRECTIONS**

10618 None.

10619 **SEE ALSO**10620 *getuid()*, *setgid()*, <sys/types.h>, <unistd.h>.10621 **CHANGE HISTORY**

10622 First released in Issue 1.

10623 Derived from Issue 1 of the SVID.

10624 **Issue 4**

10625 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 10626
- The argument list is explicitly defined as **void**.

10627 Other changes are incorporated as follows:

- 10628
- 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.

10631 **NAME**

10632 getgrent — get the group database entry

10633 **SYNOPSIS**

10634 EX #include <grp.h>

10635 struct group *getgrent(void);

10636

10637 **DESCRIPTION**10638 Refer to *endgrent()*.10639 **CHANGE HISTORY**

10640 First released in Issue 4, Version 2.

10641 **Issue 5**

10642 Moved from X/OPEN UNIX extension to BASE.

10643 NAME

10644 getgrgid, getgrgid_r — get group database entry for a group ID

10645 SYNOPSIS

10646 OH #include <sys/types.h>

10647 #include <grp.h>

10648 struct group *getgrgid(gid_t gid);

10649 int getgrgid_r(gid_t gid, struct group *grp, char *buffer,

10650 size_t bufsize, struct group **result);

10651 DESCRIPTION

10652 The *getgrgid()* function searches the group database for an entry with a matching *gid*.10653 The *getgrgid()* interface need not be reentrant.

10654 The *getgrgid_r()* function updates the **group** structure pointed to by *grp* and stores a pointer to
 10655 that structure at the location pointed to by *result*. The structure contains an entry from the group
 10656 database with a matching *gid* or *name*. Storage referenced by the group structure is allocated
 10657 from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. The
 10658 maximum size needed for this buffer can be determined with the {_SC_GETGR_R_SIZE_MAX}
 10659 *sysconf()* parameter. A NULL pointer is returned at the location pointed to by *result* on error or
 10660 if the requested entry is not found.

10661 RETURN VALUE

10662 Upon successful completion, *getgrgid()* returns a pointer to a **struct group** with the structure
 10663 defined in <grp.h> with a matching entry if one is found. The *getgrgid()* function returns a null
 10664 EX pointer if either the requested entry was not found, or an error occurred. On error, *errno* will be
 10665 set to indicate the error.

10666 The return value may point to a static area which is overwritten by a subsequent call to
 10667 *getgrent()*, *getgrgid()* or *getgrnam()*.

10668 If successful, the *getgrgid_r()* function returns zero. Otherwise, an error number is returned to
 10669 indicate the error.

10670 ERRORS

10671 The *getgrgid()* function may fail if:

10672 EX [EIO] An I/O error has occurred.

10673 [EINTR] A signal was caught during *getgrgid()*.

10674 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

10675 [ENFILE] The maximum allowable number of files is currently open in the system.

10676 The *getgrgid_r()* function may fail if:

10677 [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to
 10678 be referenced by the resulting **group** structure.

10679 EXAMPLES

10680 None.

10681 APPLICATION USAGE

10682 Applications wishing to check for error situations should set *errno* to 0 before calling *getgrgid()*.10683 If *errno* is set on return, an error occurred.

10684 FUTURE DIRECTIONS

10685 None.

10686 **SEE ALSO**10687 *endgrent()*, *getgrnam()*, *<grp.h>*, *<limits.h>*, *<sys/types.h>*.10688 **CHANGE HISTORY**

10689 First released in Issue 1.

10690 Derived from System V Release 2.0.

10691 **Issue 4**

10692 The following changes are incorporated in this issue:

- 10693 • The DESCRIPTION is clarified. |
- 10694 • In the RETURN VALUE section, the reference to the setting of *errno* is marked as an |
- 10695 extension. |
- 10696 • The errors [EIO], [EINTR], [EMFILE] and [ENFILE] are marked as extensions. |
- 10697 • A note is added to the APPLICATION USAGE section advising how applications should |
- 10698 check for errors. |
- 10699 • The *<sys/types.h>* header is added as optional (OH); this header need not be included on |
- 10700 XSI-conformant systems. |

10701 **Issue 5**10702 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
10703 VALUE section. |10704 The *getgrgid_r()* function is included for alignment with the POSIX Threads Extension. |10705 A note indicating that the *getgrgid()* interface need not be reentrant is added to the
10706 DESCRIPTION. |

10707 NAME

10708 getgrnam, getgrnam_r — search group database for a name

10709 SYNOPSIS

10710 OH #include <sys/types.h>

10711 #include <grp.h>

10712 struct group *getgrnam(const char *name);

10713 int getgrnam_r(const char *name, struct group *grp, char *buffer,

10714 size_t bufsize, struct group **result);

10715 DESCRIPTION

10716 The *getgrnam()* function searches the group database for an entry with a matching *name*.

10717 The *getgrnam()* interface need not be reentrant.

10718 The *getgrnam_r()* function updates the **group** structure pointed to by *grp* and stores a pointer to
 10719 that structure at the location pointed to by *result*. The structure contains an entry from the group
 10720 database with a matching *gid* or *name*. Storage referenced by the group structure is allocated
 10721 from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. The
 10722 maximum size needed for this buffer can be determined with the {_SC_GETGR_R_SIZE_MAX}
 10723 *sysconf()* parameter. A NULL pointer is returned at the location pointed to by *result* on error or
 10724 if the requested entry is not found.

10725 RETURN VALUE

10726 The *getgrnam()* function returns a pointer to a **struct group** with the structure defined in <grp.h>
 10727 with a matching entry if one is found. The *getgrnam()* function returns a null pointer if either the
 10728 EX requested entry was not found, or an error occurred. On error, *errno* will be set to indicate the
 10729 error.

10730 The return value may point to a static area which is overwritten by a subsequent call to
 10731 *getgrent()*, *getgrgid()* or *getgrnam()*.

10732 If successful, the *getgrnam_r()* function returns zero. Otherwise, an error number is returned to
 10733 indicate the error.

10734 ERRORS

10735 The *getgrnam()* function may fail if:

10736 EX [EIO] An I/O error has occurred.

10737 [EINTR] A signal was caught during *getgrnam()*.

10738 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

10739 [ENFILE] The maximum allowable number of files is currently open in the system.

10740 The *getgrnam_r()* function may fail if:

10741 [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to
 10742 be referenced by the resulting **group** structure.

10743 EXAMPLES

10744 None.

10745 APPLICATION USAGE

10746 Applications wishing to check for error situations should set *errno* to 0 before calling *getgrnam()*.

10747 If *errno* is set on return, an error occurred.

10748 FUTURE DIRECTIONS

10749 None.

10750 **SEE ALSO**10751 `endgrent()`, `getgrgid()`, `<grp.h>`, `<limits.h>`, `<sys/types.h>`.10752 **CHANGE HISTORY**

10753 First released in Issue 1.

10754 Derived from System V Release 2.0. |

10755 **Issue 4**

10756 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 10757
- The type of argument *name* is changed from `char *` to `const char *`.

10758 Other changes are incorporated as follows:

- 10759
- The DESCRIPTION is clarified. |

- 10760
- The `<sys/types.h>` header is added as optional (OH); this header need not be included on |
-
- 10761 XSI-conformant systems.

- 10762
- In the RETURN VALUE section, reference to the setting of *errno* is marked as an extension. |

- 10763
- The errors [EIO], [EINTR], [EMFILE] and [ENFILE] are marked as extensions.

- 10764
- A note is added to the APPLICATION USAGE section advising how applications should |
-
- 10765 check for errors.

10766 **Issue 5**10767 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
10768 VALUE section. |10769 The `getgrnam_r()` function is included for alignment with the POSIX Threads Extension. |10770 A note indicating that the `getgrnam()` interface need not be reentrant is added to the
10771 DESCRIPTION. |

10772 **NAME**

10773 getgroups — get supplementary group IDs

10774 **SYNOPSIS**

10775 OH #include <sys/types.h>

10776 #include <unistd.h>

10777 int getgroups(int *gidsetsize*, gid_t *grouplist*[]);10778 **DESCRIPTION**10779 The *getgroups()* function fills in the array *grouplist* with the current supplementary group IDs of
10780 the calling process.10781 The *gidsetsize* argument specifies the number of elements in the array *grouplist*. The actual
10782 number of supplementary group IDs stored in the array is returned. The values of array entries
10783 with indices greater than or equal to the value returned are undefined.10784 If *gidsetsize* is 0, *getgroups()* returns the number of supplementary group IDs associated with the
10785 calling process without modifying the array pointed to by *grouplist*.10786 It is unspecified whether the effective group ID of the calling process is included in, or omitted
10787 from, the returned list of supplementary group IDs.10788 **RETURN VALUE**10789 Upon successful completion, the number of supplementary group IDs is returned. A return
10790 value of -1 indicates failure and *errno* is set to indicate the error.10791 **ERRORS**10792 The *getgroups()* function will fail if:10793 [EINVAL] The *gidsetsize* argument is non-zero and is less than the number of
10794 supplementary group IDs.10795 **EXAMPLES**

10796 None.

10797 **APPLICATION USAGE**

10798 None.

10799 **FUTURE DIRECTIONS**

10800 None.

10801 **SEE ALSO**10802 *getegid()*, *setgid()*, <sys/types.h>, <unistd.h>.10803 **CHANGE HISTORY**

10804 First released in Issue 3.

10805 Entry included for alignment with the POSIX.1-1988 standard.

10806 **Issue 4**

10807 The following change is incorporated for alignment with the FIPS requirements:

- 10808
- A return value of 0 is no longer permitted, because {NGROUPS_MAX} cannot be 0.

10809	Other changes are incorporated as follows:	
10810	• The < sys/types.h > header is now marked as optional (OH); this header need not be included	
10811	on XSI-conformant systems.	
10812	• The < unistd.h > header is added to the SYNOPSIS section.	
10813	Issue 5	
10814	Normative text previously in the APPLICATION USAGE section is moved to the	
10815	DESCRIPTION.	

10816 **NAME**

10817 gethostid — get an identifier for the current host |

10818 **SYNOPSIS**

10819 EX #include <unistd.h> |

10820 long gethostid(void); |

10821 |

10822 **DESCRIPTION**10823 The *gethostid()* function retrieves a 32-bit identifier for the current host.10824 **RETURN VALUE**10825 Upon successful completion, *gethostid()* returns an identifier for the current host.10826 **ERRORS**

10827 No errors are defined. |

10828 **EXAMPLES** |

10829 None.

10830 **APPLICATION USAGE**

10831 The Open Group does not define the domain in which the return value is unique. |

10832 **FUTURE DIRECTIONS** |

10833 None.

10834 **SEE ALSO**10835 *random()*, <unistd.h>.10836 **CHANGE HISTORY**

10837 First released in Issue 4, Version 2. |

10838 **Issue 5** |

10839 Moved from X/OPEN UNIX extension to BASE.

10840 **NAME**

10841 getitimer, setitimer — get or set value of interval timer

10842 **SYNOPSIS**10843 EX

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

```
10844 int getitimer(int which, struct itimerval *value);
10845 int setitimer(int which, const struct itimerval *value,
10846              struct itimerval *ovalue);
10847
```

10848 **DESCRIPTION**

10849 The *getitimer()* function stores the current value of the timer specified by *which* into the structure
 10850 pointed to by *value*. The *setitimer()* function sets the timer specified by *which* to the value
 10851 specified in the structure pointed to by *value*, and if *ovalue* is not a null pointer, stores the
 10852 previous value of the timer in the structure pointed to by *ovalue*.

10853 A timer value is defined by the **itimerval** structure. If *it_value* is non-zero, it indicates the time to
 10854 the next timer expiration. If *it_interval* is non-zero, it specifies a value to be used in reloading
 10855 *it_value* when the timer expires. Setting *it_value* to 0 disables a timer, regardless of the value of
 10856 *it_interval*. Setting *it_interval* to 0 disables a timer after its next expiration (assuming *it_value* is
 10857 non-zero).

10858 Implementations may place limitations on the granularity of timer values. For each interval
 10859 timer, if the requested timer value requires a finer granularity than the implementation supports,
 10860 the actual timer value will be rounded up to the next supported value.

10861 An XSI-conforming implementation provides each process with at least three interval timers,
 10862 which are indicated by the *which* argument:

10863 **ITIMER_REAL**

10864 Decrements in real time. A SIGALRM signal is delivered when this timer expires.

10865 **ITIMER_VIRTUAL**

10866 Decrements in process virtual time. It runs only when the process is executing. A
 10867 SIGVTALRM signal is delivered when it expires.

10868 **ITIMER_PROF**

10869 Decrements both in process virtual time and when the system is running on behalf of the
 10870 process. It is designed to be used by interpreters in statistically profiling the execution of
 10871 interpreted programs. Each time the ITIMER_PROF timer expires, the SIGPROF signal is
 10872 delivered.

10873 The interaction between *setitimer()* and any of *alarm()*, *sleep()* or *usleep()* is unspecified.

10874 **RETURN VALUE**

10875 Upon successful completion, *getitimer()* or *setitimer()* returns 0. Otherwise, -1 is returned and
 10876 *errno* is set to indicate the error.

10877 **ERRORS**

10878 The *setitimer()* function will fail if:

10879 [EINVAL] The *value* argument is not in canonical form. (In canonical form, the number
 10880 of microseconds is a non-negative integer less than 1,000,000 and the number
 10881 of seconds is a non-negative integer.)

10882 The *getitimer()* and *setitimer()* functions may fail if:

10883 [EINVAL] The *which* argument is not recognised.

10884 **EXAMPLES**

10885 None.

10886 **APPLICATION USAGE**

10887 None.

10888 **FUTURE DIRECTIONS**

10889 None.

10890 **SEE ALSO**10891 *alarm()*, *sleep()*, *timer_gettime()*, *timer_settime()*, *ualarm()*, *usleep()*, **<signal.h>**, **<sys/time.h>**.10892 **CHANGE HISTORY**

10893 First released in Issue 4, Version 2.

10894 **Issue 5**

10895 Moved from X/OPEN UNIX extension to BASE.

10896 **NAME**

10897 getlogin, getlogin_r — get login name

10898 **SYNOPSIS**

10899 #include <unistd.h>

10900 char *getlogin(void);

10901 int getlogin_r(char *name, size_t namesize);

10902 **DESCRIPTION**

10903 The *getlogin()* function returns a pointer to a string giving a user name associated with the
 10904 calling process, which is the login name associated with the calling process. If *getlogin()* returns
 10905 a non-null pointer, then that pointer points to the name that the user logged in under, even if
 10906 there are several login names with the same user ID.

10907 The *getlogin()* interface need not be reentrant.

10908 The *getlogin_r()* function puts the name associated by the login activity with the control terminal
 10909 of the current process in the character array pointed to by *name*. The array is *namesize* characters
 10910 long and should have space for the name and the terminating null character. The maximum size
 10911 of the login name is {LOGIN_NAME_MAX}.

10912 If *getlogin_r()* is successful, *name* points to the name the user used at login, even if there are
 10913 several login names with the same user ID.

10914 **RETURN VALUE**

10915 Upon successful completion, *getlogin()* returns a pointer to the login name or a null pointer if the
 10916 EX user's login name cannot be found. Otherwise it returns a null pointer and sets *errno* to indicate
 10917 the error.

10918 The return value may point to static data whose content is overwritten by each call.

10919 If successful, the *getlogin_r()* function returns zero. Otherwise, an error number is returned to
 10920 indicate the error.

10921 **ERRORS**

10922 The *getlogin()* function may fail if:

10923 EX [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

10924 [ENFILE] The maximum allowable number of files is currently open in the system.

10925 [ENXIO] The calling process has no controlling terminal.

10926 The *getlogin_r()* function may fail if:

10927 [ERANGE] The value of *namesize* is smaller than the length of the string to be returned
 10928 including the terminating null character.

10929 **EXAMPLES**

10930 None.

10931 **APPLICATION USAGE**

10932 Three names associated with the current process can be determined: *getpwuid(geteuid())* returns
 10933 the name associated with the effective user ID of the process; *getlogin()* returns the name
 10934 associated with the current login activity; and *getpwuid(getuid())* returns the name associated
 10935 with the real user ID of the process.

10936 **FUTURE DIRECTIONS**

10937 None.

10938 **SEE ALSO**10939 `getpwnam()`, `getpwuid()`, `geteuid()`, `getuid()`, `<limits.h>`, `<unistd.h>`.10940 **CHANGE HISTORY**

10941 First released in Issue 1.

10942 Derived from System V Release 2.0.

10943 **Issue 4**

10944 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- 10945 • The argument list is explicitly defined as **void**.
- 10946 • The DESCRIPTION is updated to state explicitly that the return value is a pointer to a string
- 10947 giving the user name, rather than simply a pointer to the user name as stated in previous
- 10948 issues.

10949 Other changes are incorporated as follows:

- 10950 • The `<unistd.h>` header is added to the SYNOPSIS section.
- 10951 • In the RETURN VALUE section, reference to the setting of *errno* is marked as an extension.
- 10952 • The behaviour of the function when the login name cannot be found is included in the
- 10953 RETURN VALUE section instead of the DESCRIPTION.
- 10954 • The errors [EMFILE], [ENFILE] and [ENXIO] are marked as extensions.
- 10955 • The APPLICATION USAGE section is changed to refer to `getpwuid()` rather than `cuserid()`,
- 10956 which may be withdrawn in a future issue.

10957 **Issue 5**10958 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
10959 VALUE section.10960 The `getlogin_r()` function is included for alignment with the POSIX Threads Extension.10961 A note indicating that the `getlogin()` interface need not be reentrant is added to the
10962 DESCRIPTION.

10963 NAME

10964 getmsg, getpmsg — receive next message from a STREAMS file

10965 SYNOPSIS

10966 EX

```
#include <stropts.h>
```

```
10967 int getmsg(int fildes, struct strbuf *ctlptr, struct strbuf *dataptr,
10968           int *flagsp);
10969 int getpmsg(int fildes, struct strbuf *ctlptr, struct strbuf *dataptr,
10970            int *bandp, int *flagsp);
10971
```

10972 DESCRIPTION

10973 The *getmsg()* function retrieves the contents of a message located at the head of the STREAM
 10974 head read queue associated with a STREAMS file and places the contents into one or more
 10975 buffers. The message contains either a data part, a control part, or both. The data and control
 10976 parts of the message are placed into separate buffers, as described below. The semantics of each
 10977 part is defined by the originator of the message.

10978 The *getpmsg()* function does the same thing as *getmsg()*, but provides finer control over the
 10979 priority of the messages received. Except where noted, all requirements on *getmsg()* also pertain
 10980 to *getpmsg()*.

10981 The *fildes* argument specifies a file descriptor referencing a STREAMS-based file.

10982 The *ctlptr* and *dataptr* arguments each point to a **strbuf** structure, in which the **buf** member
 10983 points to a buffer in which the data or control information is to be placed, and the **maxlen**
 10984 member indicates the maximum number of bytes this buffer can hold. On return, the **len**
 10985 member contains the number of bytes of data or control information actually received. The **len**
 10986 member is set to 0 if there is a zero-length control or data part and **len** is set to -1 if no data or
 10987 control information is present in the message.

10988 When *getmsg()* is called, *flagsp* should point to an integer that indicates the type of message the
 10989 process is able to receive. This is described further below.

10990 The *ctlptr* argument is used to hold the control part of the message, and *dataptr* is used to hold
 10991 the data part of the message. If *ctlptr* (or *dataptr*) is a null pointer or the **maxlen** member is -1,
 10992 the control (or data) part of the message is not processed and is left on the STREAM head read
 10993 queue, and if the *ctlptr* (or *dataptr*) is not a null pointer, **len** is set to -1. If the **maxlen** member is
 10994 set to 0 and there is a zero-length control (or data) part, that zero-length part is removed from
 10995 the read queue and **len** is set to 0. If the **maxlen** member is set to 0 and there are more than 0
 10996 bytes of control (or data) information, that information is left on the read queue and **len** is set to
 10997 0. If the **maxlen** member in *ctlptr* (or *dataptr*) is less than the control (or data) part of the
 10998 message, **maxlen** bytes are retrieved. In this case, the remainder of the message is left on the
 10999 STREAM head read queue and a non-zero return value is provided.

11000 By default, *getmsg()* processes the first available message on the STREAM head read queue.
 11001 However, a process may choose to retrieve only high-priority messages by setting the integer
 11002 pointed to by *flagsp* to RS_HIPRI. In this case, *getmsg()* will only process the next message if it is
 11003 a high-priority message. When the integer pointed to by *flagsp* is 0, any message will be
 11004 retrieved. In this case, on return, the integer pointed to by *flagsp* will be set to RS_HIPRI if a
 11005 high-priority message was retrieved, or 0 otherwise.

11006 For *getpmsg()*, the flags are different. The *flagsp* argument points to a bitmask with the following
 11007 mutually-exclusive flags defined: MSG_HIPRI, MSG_BAND and MSG_ANY. Like *getmsg()*,
 11008 *getpmsg()* processes the first available message on the STREAM head read queue. A process
 11009 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()* will 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()* will 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* will be set to MSG_HIPRI and the integer pointed to by *bandp* will be set to 0. Otherwise, the integer pointed to by *flagsp* will be set to MSG_BAND and the integer pointed to by *bandp* will be set to the priority band of the message.

If O_NONBLOCK is not set, *getmsg()* and *getpmsg()* will 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()* fail and set *errno* to [EAGAIN].

If a hangup occurs on the STREAM from which messages are to be 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*.

11028 RETURN VALUE

Upon successful completion, *getmsg()* and *getpmsg()* 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 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()* retrieves 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 will be placed back on the queue as a normal message of band 0. Subsequent *getmsg()* and *getpmsg()* calls 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()* return -1 and set *errno* to indicate the error.

11043 ERRORS

The *getmsg()* and *getpmsg()* functions will fail if:

- | | | |
|-------|-----------|--|
| 11045 | [EAGAIN] | The O_NONBLOCK flag is set and no messages are available. |
| 11046 | [EBADF] | The <i>fildev</i> argument is not a valid file descriptor open for reading. |
| 11047 | [EBADMSG] | The queued message to be read is not valid for <i>getmsg()</i> or <i>getpmsg()</i> or a pending file descriptor is at the STREAM head. |
| 11049 | [EINTR] | A signal was caught during <i>getmsg()</i> or <i>getpmsg()</i> . |
| 11050 | [EINVAL] | An illegal value was specified by <i>flagsp</i> , or the STREAM or multiplexer referenced by <i>fildev</i> is linked (directly or indirectly) downstream from a multiplexer. |
| 11053 | [ENOSTR] | A STREAM is not associated with <i>fildev</i> . |

In addition, *getmsg()* and *getpmsg()* will 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

11056 *getmsg()* or *getpmsg()* but reflects the prior error.

11057 **EXAMPLES**

11058 None.

11059 **APPLICATION USAGE**

11060 None.

11061 **FUTURE DIRECTIONS**

11062 None.

11063 **SEE ALSO**

11064 *poll()*, *putmsg()*, *read()*, *write()*, **<stropts.h>**, Section 2.5 on page 34.

11065 **CHANGE HISTORY**

11066 First released in Issue 4, Version 2.

11067 **Issue 5**

11068 Moved from X/OPEN UNIX extension to BASE.

11069 A paragraph regarding “high-priority control parts of messages” is added to the RETURN
11070 VALUE section.

11071 **NAME**

11072 getopt, optarg, optind, opterr, optopt — command option parsing

11073 **SYNOPSIS**

11074 #include <unistd.h>

11075 int getopt(int argc, char * const argv[], const char *optstring);

11076 extern char *optarg;

11077 extern int optind, opterr, optopt;

11078 **DESCRIPTION**

11079 The *getopt()* function is a command-line parser that can be used by applications that follow
 11080 Utility Syntax Guidelines 3, 4, 5, 6, 7, 9 and 10 in the **XBD** specification, **Section 10.2, Utility**
 11081 **Syntax Guidelines**. The remaining guidelines are not addressed by *getopt()* and are the
 11082 responsibility of the application.

11083 The parameters *argc* and *argv* are the argument count and argument array as passed to *main()*
 11084 (see *exec*). The argument *optstring* is a string of recognised option characters; if a character is
 11085 followed by a colon, the option takes an argument. All option characters allowed by Utility
 11086 Syntax Guideline 3 are allowed in *optstring*. The implementation may accept other characters as
 11087 an extension.

11088 The variable *optind* is the index of the next element of the *argv[]* vector to be processed. It is
 11089 initialised to 1 by the system, and *getopt()* updates it when it finishes with each element of
 11090 *argv[]*. When an element of *argv[]* contains multiple option characters, it is unspecified how
 11091 *getopt()* determines which options have already been processed.

11092 The *getopt()* function returns the next option character (if one is found) from *argv* that matches a
 11093 character in *optstring*, if there is one that matches. If the option takes an argument, *getopt()* sets
 11094 the variable *optarg* to point to the option-argument as follows:

- 11095 1. If the option was the last character in the string pointed to by an element of *argv*, then
 11096 *optarg* contains the next element of *argv*, and *optind* is incremented by 2. If the resulting
 11097 value of *optind* is not less than *argc*, this indicates a missing option-argument, and *getopt()*
 11098 returns an error indication.
- 11099 2. Otherwise, *optarg* points to the string following the option character in that element of
 11100 *argv*, and *optind* is incremented by 1.

11101 If, when *getopt()* is called:

11102 <i>argv[optind]</i>	is a null pointer
11103 * <i>argv[optind]</i>	is not the character –
11104 <i>argv[optind]</i>	points to the string "--"

11105 *getopt()* returns –1 without changing *optind*. If:11106 *argv[optind]* points to the string "--"11107 *getopt()* returns –1 after incrementing *optind*.

11108 If *getopt()* encounters an option character that is not contained in *optstring*, it returns the
 11109 question-mark (?) character. If it detects a missing option-argument, it returns the colon
 11110 character (:). If the first character of *optstring* was a colon, or a question-mark character (?)
 11111 otherwise. In either case, *getopt()* will set the variable *optopt* to the option character that caused
 11112 the error. If the application has not set the variable *opterr* to 0 and the first character of *optstring*
 11113 is not a colon, *getopt()* also prints a diagnostic message to *stderr* in the format specified for the
 11114 *getopts* utility.

11115 RETURN VALUE

11116 The *getopt()* function returns the next option character specified on the command line.

11117 A colon (:) is returned if *getopt()* detects a missing argument and the first character of *optstring*
11118 was a colon (:).

11119 A question mark (?) is returned if *getopt()* encounters an option character not in *optstring* or
11120 detects a missing argument and the first character of *optstring* was not a colon (:).

11121 Otherwise *getopt()* returns -1 when all command line options are parsed.

11122 ERRORS

11123 No errors are defined.

11124 EXAMPLES

11125 The following code fragment shows how one might process the arguments for a utility that can
11126 take the mutually exclusive options *a* and *b* and the options *f* and *o*, both of which require
11127 arguments:

```
11128 #include <unistd.h>
11129 int
11130 main (int argc, char *argv[ ])
11131 {
11132     int c;
11133     int bflg, aflag, errflag;
11134     char *ifile;
11135     char *ofile;
11136     extern char *optarg;
11137     extern int optind, optopt;
11138     . . .
11139     while ((c = getopt(argc, argv, ":abf:o:")) != -1) {
11140         switch (c) {
11141             case 'a':
11142                 if (bflg)
11143                     errflag++;
11144                 else
11145                     aflag++;
11146                 break;
11147             case 'b':
11148                 if (aflag)
11149                     errflag++;
11150                 else {
11151                     bflg++;
11152                     bproc();
11153                 }
11154                 break;
11155             case 'f':
11156                 ifile = optarg;
11157                 break;
11158             case 'o':
11159                 ofile = optarg;
11160                 break;
11161             case '': /* -f or -o without operand */
11162                 fprintf(stderr,
11163                     "Option -%c requires an operand\n", optopt);
11164                 errflag++;
11165                 break;
11166             case '?':
11167                 fprintf(stderr,
```

```

11168                                     "Unrecognised option: -%c\n", optopt);
11169             errflg++;
11170         }
11171     }
11172     if (errflg) {
11173         fprintf(stderr, "usage: . . . ");
11174         exit(2);
11175     }
11176     for ( ; optind < argc; optind++) {
11177         if (access(argv[optind], R_OK)) {
11178             . . .
11179         }

```

11180 This code accepts any of the following as equivalent:

```

11181     cmd -ao arg path path
11182     cmd -a -o arg path path
11183     cmd -o arg -a path path
11184     cmd -a -o arg -- path path
11185     cmd -a -oarg path path
11186     cmd -aoarg path path

```

11187 APPLICATION USAGE

11188 The *getopt()* function is only required to support option characters included in Guideline 3.
 11189 Many historical implementations of *getopt()* support other characters as options. This is an
 11190 allowed extension, but applications that use extensions are not maximally portable. Note that
 11191 support for multi-byte option characters is only possible when such characters can be
 11192 represented as type **int**.

11193 The *getopt()* interface need not be reentrant.

11194 FUTURE DIRECTIONS

11195 None.

11196 SEE ALSO

11197 *exec*, *getopts*, **<unistd.h>**, the XCU specification.

11198 CHANGE HISTORY

11199 First released in Issue 1.

11200 Derived from Issue 1 of the SVID.

11201 Issue 4

11202 The following changes are incorporated for alignment with the ISO POSIX-2 standard:

- 11203 • The header **<unistd.h>** is added to the SYNOPSIS section and **<stdio.h>** is deleted.
- 11204 • The type of argument *argv* is changed from **char **** to **char * const []**.
- 11205 • The integer *optopt* is added to the list of external data items.
- 11206 • The DESCRIPTION is largely rewritten, without functional change, for alignment with the
 11207 ISO POSIX-2 standard, although the following differences should be noted:
 - 11208 — If the function detects a missing option-argument, it returns a colon (:) and sets **optopt** to
 11209 the option character.
 - 11210 — The termination conditions under which *getopt()* will return **-1** are extended. Also note
 11211 that the termination condition is explicitly **-1**, rather than the value of EOF.
- 11212 • The EXAMPLES section is changed to illustrate the new functionality.

11213 **Issue 5**

11214 A note indicating that the *getopt()* interface need not be reentrant is added to the DESCRIPTION.

11215 **NAME**11216 getpagesize — get the current page size (**LEGACY**)11217 **SYNOPSIS**

11218 EX #include <unistd.h>

11219 int getpagesize(void);

11220

11221 **DESCRIPTION**11222 The *getpagesize()* function returns the current page size.11223 The *getpagesize()* function is equivalent to *sysconf(_SC_PAGE_SIZE)* and
11224 *sysconf(_SC_PAGESIZE)*.

11225 This interface need not be reentrant.

11226 **RETURN VALUE**11227 The *getpagesize()* function returns the current page size.11228 **ERRORS**

11229 No errors are defined.

11230 **EXAMPLES**

11231 None.

11232 **APPLICATION USAGE**11233 The value returned by *getpagesize()* need not be the minimum value that *malloc()* can allocate.
11234 Moreover, the application cannot assume that an object of this size can be allocated with
11235 *malloc()*.11236 This interface, returning an **int**, may have problems representing appropriate values in the
11237 future. Applications should use the *sysconf()* function instead.11238 **FUTURE DIRECTIONS**

11239 None.

11240 **SEE ALSO**11241 *getrlimit()*, *mmap()*, *mprotect()*, *munmap()*, *msync()*, *sysconf()*, <unistd.h>.11242 **CHANGE HISTORY**

11243 First released in Issue 4, Version 2.

11244 **Issue 5**

11245 Moved from X/OPEN UNIX extension to BASE.

11246 A new paragraph is added to the APPLICATION USAGE section indicating why the interface
11247 may be withdrawn in a future issue.

11248 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

11249 Marked LEGACY.

11250 **NAME**11251 getpass — read a string of characters without echo (**LEGACY**)11252 **SYNOPSIS**11253 EX `#include <unistd.h>`11254 `char *getpass(const char *prompt);`

11255

11256 **DESCRIPTION**

11257 The *getpass()* function opens the process' controlling terminal, writes to that device the null-terminated string *prompt*, disables echoing, reads a string of characters up to the next newline character or EOF, restores the terminal state and closes the terminal.

11260 This interface need not be reentrant.

11261 **RETURN VALUE**

11262 Upon successful completion, *getpass()* returns a pointer to a null-terminated string of at most {PASS_MAX} bytes that were read from the terminal device. If an error is encountered, the terminal state is restored and a null pointer is returned.

11265 **ERRORS**11266 The *getpass()* function may fail if:11267 [EINTR] The *getpass()* function was interrupted by a signal.

11268 [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-dependent reasons.

11272 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

11273 [ENFILE] The maximum allowable number of files is currently open in the system.

11274 [ENXIO] The process does not have a controlling terminal.

11275 **EXAMPLES**

11276 None.

11277 **APPLICATION USAGE**

11278 The return value points to static data whose content may be overwritten by each call.

11279 This function was marked **LEGACY** since it provides no functionality which a user could not easily implement, and its name is misleading.

11281 **FUTURE DIRECTIONS**

11282 None.

11283 **SEE ALSO**11284 `<limits.h>`, `<unistd.h>`.11285 **CHANGE HISTORY**

11286 First released in Issue 1.

11287 Derived from System V Release 2.0.

11288 **Issue 4**

11289 The following changes are incorporated in this issue:

- 11290 • The interface is marked TO BE WITHDRAWN, because of its misleading name and because
11291 it provides dubious functionality.
- 11292 • The <**unistd.h**> header is added to the SYNOPSIS section. |
- 11293 • The type of argument *prompt* is changed from **char *** to **const char ***.
- 11294 • In the DESCRIPTION, reference to the character special file **/dev/tty** is replaced by the phrase
11295 “the process’ controlling terminal”.
- 11296 • In the RETURN VALUE section, the word “characters” is replaced by “bytes”, to indicate
11297 that this interface deals solely in single-byte values.
- 11298 • A note is added to the APPLICATION USAGE section indicating why the interface may be
11299 withdrawn in a future issue. |

11300 **Issue 5**

11301 Marked LEGACY. |

11302 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

11303 **NAME**

11304 getpgid — get the process group ID for a process

11305 **SYNOPSIS**

11306 EX #include <unistd.h>

11307 pid_t getpgid(pid_t pid);

11308

11309 **DESCRIPTION**

11310 The *getpgid()* function returns the process group ID of the process whose process ID is equal to

11311 *pid*. If *pid* is equal to 0, *getpgid()* returns the process group ID of the calling process.

11312 **RETURN VALUE**

11313 Upon successful completion, *getpgid()* returns a process group ID. Otherwise, it returns

11314 (**pid_t**)−1 and sets *errno* to indicate the error.

11315 **ERRORS**11316 The *getpgid()* function will fail if:

11317 [EPERM] The process whose process ID is equal to *pid* is not in the same session as the

11318 calling process, and the implementation does not allow access to the process

11319 group ID of that process from the calling process.

11320 [ESRCH] There is no process with a process ID equal to *pid*.

11321 The *getpgid()* function may fail if:

11322 [EINVAL] The value of the *pid* argument is invalid.

11323 **EXAMPLES**

11324 None.

11325 **APPLICATION USAGE**

11326 None.

11327 **FUTURE DIRECTIONS**

11328 None.

11329 **SEE ALSO**11330 *exec*, *fork()*, *getpgrp()*, *getpid()*, *getsid()*, *setpgid()*, *setsid()*, <unistd.h>.11331 **CHANGE HISTORY**

11332 First released in Issue 4, Version 2.

11333 **Issue 5**

11334 Moved from X/OPEN UNIX extension to BASE.

11335 **NAME**

11336 getpgrp — get the process group ID of the calling process

11337 **SYNOPSIS**

11338 OH #include <sys/types.h>

11339 #include <unistd.h>

11340 pid_t getpgrp(void);

11341 **DESCRIPTION**11342 The *getpgrp()* function returns the process group ID of the calling process.11343 **RETURN VALUE**11344 The *getpgrp()* function is always successful and no return value is reserved to indicate an error.11345 **ERRORS**

11346 No errors are defined.

11347 **EXAMPLES**

11348 None.

11349 **APPLICATION USAGE**

11350 None.

11351 **FUTURE DIRECTIONS**

11352 None.

11353 **SEE ALSO**11354 *exec*, *fork()*, *getpgid()*, *getpid()*, *getppid()*, *kill()*, *setpgid()*, *setsid()*, <sys/types.h>, <unistd.h>.11355 **CHANGE HISTORY**

11356 First released in Issue 1.

11357 Derived from Issue 1 of the SVID.

11358 **Issue 4**

11359 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 11360
- The argument list is explicitly defined as **void**.

11361 Other changes are incorporated in this issue as follows:

- 11362
- 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.

11365 **NAME**11366 `getpid` — get the process ID11367 **SYNOPSIS**11368 OH `#include <sys/types.h>`11369 `#include <unistd.h>`11370 `pid_t getpid(void);`11371 **DESCRIPTION**11372 The `getpid()` function returns the process ID of the calling process.11373 **RETURN VALUE**11374 The `getpid()` function is always successful and no return value is reserved to indicate an error.11375 **ERRORS**

11376 No errors are defined.

11377 **EXAMPLES**

11378 None.

11379 **APPLICATION USAGE**

11380 None.

11381 **FUTURE DIRECTIONS**

11382 None.

11383 **SEE ALSO**11384 `exec`, `fork()`, `getpgrp()`, `getppid()`, `kill()`, `setpgid()`, `setsid()`, `<sys/types.h>`, `<unistd.h>`.11385 **CHANGE HISTORY**

11386 First released in Issue 1.

11387 Derived from Issue 1 of the SVID.

11388 **Issue 4**

11389 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 11390 • The argument list is explicitly defined as
- void**
- .

11391 Other changes are incorporated in this issue as follows:

- 11392 • The
- `<sys/types.h>`
- header is now marked as optional (OH); this header need not be included
-
- 11393 on XSI-conformant systems.

- 11394 • The
- `<unistd.h>`
- header is added to the SYNOPSIS section.

11395 **NAME**

11396 getpmsg — get the user database entry |

11397 **SYNOPSIS**

11398 EX #include <pwd.h> |

```
11399       int getpmsg(int fildes, struct strbuf *ctlptr, struct strbuf *dataptr,  
11400                   int *bandp, int *flagsp);
```

 |

11401 |

11402 **DESCRIPTION**11403 Refer to *getmsg()*.11404 **CHANGE HISTORY**

11405 First released in Issue 4, Version 2. |

11406 **Issue 5** |

11407 Moved from X/OPEN UNIX extension to BASE.

11408 **NAME**

11409 getppid — get the parent process ID

11410 **SYNOPSIS**

11411 OH #include <sys/types.h>

11412 #include <unistd.h>

11413 pid_t getppid(void);

11414 **DESCRIPTION**11415 The *getppid()* function returns the parent process ID of the calling process.11416 **RETURN VALUE**11417 The *getppid()* function is always successful and no return value is reserved to indicate an error.11418 **ERRORS**

11419 No errors are defined.

11420 **EXAMPLES**

11421 None.

11422 **APPLICATION USAGE**

11423 None.

11424 **FUTURE DIRECTIONS**

11425 None.

11426 **SEE ALSO**11427 *exec*, *fork()*, *getpgid()*, *getpgrp()*, *getpid()*, *kill()*, *setpgid()*, *setsid()*, <sys/types.h>, <unistd.h>.11428 **CHANGE HISTORY**

11429 First released in Issue 1.

11430 Derived from Issue 1 of the SVID.

11431 **Issue 4**

11432 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 11433
- The argument list is explicitly defined as **void**.

11434 Other changes are incorporated in this issue as follows:

- 11435
- The <sys/types.h> header is now marked as optional (OH); this header need not be included
-
- 11436 on XSI-conformant systems.
-
- 11437
- The <unistd.h> header is added to the SYNOPSIS section.

11438 **NAME**

11439 getpriority, setpriority — get or set the nice value

11440 **SYNOPSIS**

11441 EX #include <sys/resource.h>

11442 int getpriority(int *which*, id_t *who*);11443 int setpriority(int *which*, id_t *who*, int *value*);

11444

11445 **DESCRIPTION**

11446 The *getpriority()* function obtains the nice value of a process, process group or user. The
 11447 *setpriority()* function sets the nice value of a process, process group or user to *value* + NZERO.

11448 Target processes are specified by the values of the *which* and *who* arguments. The *which*
 11449 argument may be one of the following values: PRIO_PROCESS, PRIO_PGRP or PRIO_USER,
 11450 indicating that the *who* argument is to be interpreted as a process ID, a process group ID or an
 11451 effective user ID, respectively. A 0 value for the *who* argument specifies the current process,
 11452 process group or user.

11453 The nice value set with *setpriority()* is applied to the process. If the process is multi-threaded,
 11454 the nice value affects all system scope threads in the process.

11455 If more than one process is specified, *getpriority()* returns value NZERO less than the lowest nice
 11456 value pertaining to any of the specified processes, and *setpriority()* sets the nice values of all of
 11457 the specified processes to *value* + NZERO.

11458 The default nice value is NZERO; lower nice values cause more favourable scheduling. While
 11459 the range of valid nice values is [0, NZERO*2 - 1], implementations may enforce more restrictive
 11460 limits. If *value* + NZERO is less than the system's lowest supported nice value, *setpriority()* sets
 11461 the nice value to the lowest supported value; if *value* + NZERO is greater than the system's
 11462 highest supported nice value, *setpriority()* sets the nice value to the highest supported value.

11463 Only a process with appropriate privileges can lower its nice value.

11464 RT Any processes or threads using SCHED_FIFO or SCHED_RR are unaffected by a call to
 11465 *setpriority()*. This is not considered an error.

11466 The effect of changing the nice value may vary depending on the process-scheduling algorithm
 11467 in effect.

11468 Because *getpriority()* can return the value -1 on successful completion, it is necessary to set *errno*
 11469 to 0 prior to a call to *getpriority()*. If *getpriority()* returns the value -1, then *errno* can be checked
 11470 to see if an error occurred or if the value is a legitimate nice value.

11471 **RETURN VALUE**

11472 Upon successful completion, *getpriority()* returns an integer in the range from -NZERO to
 11473 NZERO-1. Otherwise, -1 is returned and *errno* is set to indicate the error.

11474 Upon successful completion, *setpriority()* returns 0. Otherwise, -1 is returned and *errno* is set to
 11475 indicate the error.

11476 **ERRORS**

11477 The *getpriority()* and *setpriority()* functions will fail if:

11478 [ESRCH] No process could be located using the *which* and *who* argument values
 11479 specified.

11480 [EINVAL] The value of the *which* argument was not recognised, or the value of the *who*
 11481 argument is not a valid process ID, process group ID or user ID.

11482 In addition, *setpriority()* may fail if:

11483 [EPERM] A process was located, but neither the real nor effective user ID of the
 11484 executing process match the effective user ID of the process whose nice value
 11485 is being changed.

11486 [EACCES] A request was made to change the nice value to a lower numeric value and
 11487 the current process does not have appropriate privileges.

11488 EXAMPLES

11489 None.

11490 APPLICATION USAGE

11491 The *getpriority()* and *setpriority()* functions work with an offset nice value (nice value minus
 11492 NZERO). The nice value is in the range $[0, 2 \times \text{NZERO} - 1]$, while the return value for *getpriority()*
 11493 and the third parameter for *setpriority()* are in the range $[-\text{NZERO}, \text{NZERO} - 1]$.

11494 FUTURE DIRECTIONS

11495 None.

11496 SEE ALSO

11497 *nice()*, *sched_get_priority_max()*, *sched_setscheduler()*, <sys/resource.h>.

11498 CHANGE HISTORY

11499 First released in Issue 4, Version 2.

11500 Issue 5

11501 Moved from X/OPEN UNIX extension to BASE.

11502 The DESCRIPTION is reworded in terms of the nice value rather than *priority* to avoid confusion
 11503 with functionality in the POSIX Realtime Extension.

11504 **NAME**

11505 getpwent — get user database entry

11506 **SYNOPSIS**

11507 EX #include <pwd.h>

11508 struct passwd *getpwent(void);

11509

11510 **DESCRIPTION**11511 Refer to *endpwent()*.11512 **CHANGE HISTORY**

11513 First released in Issue 4, Version 2.

11514 **Issue 5**

11515 Moved from X/OPEN UNIX extension to BASE.

11516 NAME

11517 getpwnam, getpwnam_r — search user database for a name

11518 SYNOPSIS

11519 OH #include <sys/types.h>

11520 #include <pwd.h>

11521 struct passwd *getpwnam(const char *name);

11522 int getpwnam_r(const char *nam, struct passwd *pwd, char *buffer,

11523 size_t bufsize, struct passwd **result);

11524 DESCRIPTION

11525 The *getpwnam()* function searches the user database for an entry with a matching *name*.11526 The *getpwnam()* interface need not be reentrant.

11527 The *getpwnam_r()* function updates the **passwd** structure pointed to by *pwd* and stores a pointer to that structure at the location pointed to by *result*. The structure will contain an entry from the user database with a matching *uid* or *name*. Storage referenced by the structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. The maximum size needed for this buffer can be determined with the `{_SC_GETPW_R_SIZE_MAX}` *sysconf()* parameter. A NULL pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

11534 Applications wishing to check for error situations should set *errno* to 0 before calling *getpwnam()*. If *getpwnam()* returns a null pointer and *errno* is non-zero, an error occurred.

11536 RETURN VALUE

11537 The *getpwnam()* function returns a pointer to a **struct passwd** with the structure as defined in <**pwd.h**> with a matching entry if found. A null pointer is returned if the requested entry is not found, or an error occurs. On error, *errno* is set to indicate the error.

11540 The return value may point to a static area which is overwritten by a subsequent call to *getpwent()*, *getpwnam()* or *getpwuid()*.

11542 If successful, the *getpwnam_r()* function returns zero. Otherwise, an error number is returned to indicate the error.

11544 ERRORS

11545 The *getpwnam()* function may fail if:

11546 EX [EIO] An I/O error has occurred.

11547 [EINTR] A signal was caught during *getpwnam()*.

11548 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

11549 [ENFILE] The maximum allowable number of files is currently open in the system.

11550 The *getpwnam_r()* function may fail if:

11551 [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be referenced by the resulting **passwd** structure.

11553 EXAMPLES

11554 None.

11555 APPLICATION USAGE

11556 Three names associated with the current process can be determined: *getpwuid(getuid())* 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

11559 with the real user ID of the process.

11560 FUTURE DIRECTIONS

11561 None.

11562 SEE ALSO

11563 *getpwuid()*, *<limits.h>*, *<pwd.h>*, *<sys/types.h>*.

11564 CHANGE HISTORY

11565 First released in Issue 1.

11566 Derived from System V Release 2.0.

11567 Issue 4

11568 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 11569 • The type of argument *name* is changed from **char *** to **const char ***.

11570 Other changes are incorporated as follows:

- 11571 • The DESCRIPTION is clarified.
- 11572 • The *<sys/types.h>* header is now marked as optional (OH); this header need not be included
- 11573 on XSI-conformant systems.
- 11574 • The last sentence in the RETURN VALUE section, indicating that *errno* will be set on error, is
- 11575 marked as an extension.
- 11576 • The errors [EIO], [EINTR], [EMFILE] and [ENFILE] are marked as extensions.
- 11577 • The APPLICATION USAGE section is expanded (a) to warn about possible reuses of the area
- 11578 used to pass the return value, and (b) to indicate how applications should check for errors.

11579 Issue 5

11580 Normative text previously in the APPLICATION USAGE section is moved to the RETURN

11581 VALUE section.

11582 The *getpwnam_r()* function is included for alignment with the POSIX Threads Extension.

11583 A note indicating that the *getpwnam()* interface need not be reentrant is added to the

11584 DESCRIPTION.

11585 **NAME**

11586 getpwuid, getpwuid_r — search user database for a user ID

11587 **SYNOPSIS**

11588 OH #include <sys/types.h>

11589 #include <pwd.h>

11590 struct passwd *getpwuid(uid_t uid);

11591 int getpwuid_r(uid_t uid, struct passwd *pwd, char *buffer,

11592 size_t bufsize, struct passwd **result);

11593 **DESCRIPTION**11594 The *getpwuid()* function searches the user database for an entry with a matching *uid*.11595 The *getpwuid()* interface need not be reentrant.

11596 The *getpwuid_r()* function updates the **passwd** structure pointed to by *pwd* and stores a pointer to that structure at the location pointed to by *result*. The structure will contain an entry from the user database with a matching *uid* or *name*. Storage referenced by the structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. The maximum size needed for this buffer can be determined with the `{_SC_GETPW_R_SIZE_MAX}` *sysconf()* parameter. A NULL pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

11603 Applications wishing to check for error situations should set *errno* to 0 before calling *getpwuid()*.
 11604 If *getpwuid()* returns a null pointer and *errno* is set to non-zero, an error occurred.

11605 **RETURN VALUE**

11606 The *getpwuid()* function returns a pointer to a **struct passwd** with the structure as defined in
 11607 <**pwd.h**> with a matching entry if found. A null pointer is returned if the requested entry is not
 11608 EX found, or an error occurs. On error, *errno* is set to indicate the error.

11609 The return value may point to a static area which is overwritten by a subsequent call to
 11610 *getpwent()*, *getpwnam()* or *getpwuid()*.

11611 If successful, the *getpwuid_r()* function returns zero. Otherwise, an error number is returned to
 11612 indicate the error.

11613 **ERRORS**11614 The *getpwuid()* function may fail if:

11615 EX [EIO] An I/O error has occurred.

11616 [EINTR] A signal was caught during *getpwuid()*.

11617 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

11618 [ENFILE] The maximum allowable number of files is currently open in the system.

11619 The *getpwuid_r()* function may fail if:

11620 [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to
 11621 be referenced by the resulting **passwd** structure.

11622 **EXAMPLES**

11623 None.

11624 **APPLICATION USAGE**

11625 Three names associated with the current process can be determined: *getpwuid(getuid())* returns
 11626 the name associated with the effective user ID of the process; *getlogin()* returns the name
 11627 associated with the current login activity; and *getpwuid(getuid())* returns the name associated

11628 with the real user ID of the process.

11629 FUTURE DIRECTIONS

11630 None.

11631 SEE ALSO

11632 *getpwnam()*, *geteuid()*, *getuid()*, *getlogin()*, **<limits.h>**, **<pwd.h>**, **<sys/types.h>**.

11633 CHANGE HISTORY

11634 First released in Issue 1.

11635 Derived from System V Release 2.0.

11636 Issue 4

11637 The following changes are incorporated in this issue:

- 11638 • The DESCRIPTION is clarified.
- 11639 • The **<sys/types.h>** header is now marked as optional (OH); this header need not be included
- 11640 on XSI-conformant systems.
- 11641 • The last sentence in the RETURN VALUE section, indicating that *errno* will be set on error, is
- 11642 marked as an extension.
- 11643 • The errors [EIO], [EINTR], [EMFILE] and [ENFILE] are marked as extensions.
- 11644 • A note is added to the APPLICATION USAGE section indicating how an application should
- 11645 check for errors.

11646 Issue 5

11647 Normative text previously in the APPLICATION USAGE section is moved to the RETURN

11648 VALUE section.

11649 The *getpwuid_r()* function is included for alignment with the POSIX Threads Extension.

11650 A note indicating that the *getpwuid()* interface need not be reentrant is added to the

11651 DESCRIPTION.

11652 NAME

11653 getrlimit, setrlimit — control maximum resource consumption

11654 SYNOPSIS

11655 EX

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

11656

```
int getrlimit(int resource, struct rlimit *rlp);
```

11657

```
int setrlimit(int resource, const struct rlimit *rlp);
```

11658

11659 DESCRIPTION

11660 Limits on the consumption of a variety of resources by the calling process may be obtained with
11661 *getrlimit()* and set with *setrlimit()*.

11662 Each call to either *getrlimit()* or *setrlimit()* identifies a specific resource to be operated upon as
 11663 well as a resource limit. A resource limit is represented by an **rlimit** structure. The **rlim_cur**
 11664 member specifies the current or soft limit and the **rlim_max** member specifies the maximum or
 11665 hard limit. Soft limits may be changed by a process to any value that is less than or equal to the
 11666 hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or
 11667 equal to the soft limit. Only a process with appropriate privileges can raise a hard limit. Both
 11668 hard and soft limits can be changed in a single call to *setrlimit()* subject to the constraints
 11669 described above.

11670 The value RLIM_INFINITY, defined in **<sys/resource.h>**, is considered to be larger than any
 11671 other limit value. If a call to *getrlimit()* returns RLIM_INFINITY for a resource, it means the
 11672 implementation does not enforce limits on that resource. Specifying RLIM_INFINITY as any
 11673 resource limit value on a successful call to *setrlimit()* inhibits enforcement of that resource limit.

11674 The following resources are defined:

11675 **RLIMIT_CORE** This is the maximum size of a core file in bytes that may be created by a
 11676 process. A limit of 0 will prevent the creation of a core file. If this limit is
 11677 exceeded, the writing of a core file will terminate at this size.

11678 **RLIMIT_CPU** This is the maximum amount of CPU time in seconds used by a process. If
 11679 this limit is exceeded, SIGXCPU is generated for the process. If the process is
 11680 catching or ignoring SIGXCPU, or all threads belonging to that process are
 11681 blocking SIGXCPU, the behaviour is unspecified.

11682 **RLIMIT_DATA** This is the maximum size of a process' data segment in bytes. If this limit is
 11683 exceeded, the *brk()*, *malloc()* and *sbrk()* functions will fail with *errno* set to
 11684 [ENOMEM].

11685 **RLIMIT_FSIZE** This is the maximum size of a file in bytes that may be created by a process. If
 11686 a write or truncate operation would cause this limit to be exceeded, SIGXFSZ
 11687 is generated for the thread. If the thread is blocking, or the process is catching
 11688 or ignoring SIGXFSZ, continued attempts to increase the size of a file from
 11689 end-of-file to beyond the limit will fail with *errno* set to [EFBIG].

11690 **RLIMIT_NOFILE**

11691 This is a number one greater than the maximum value that the system may
 11692 assign to a newly-created descriptor. If this limit is exceeded, functions that
 11693 allocate new file descriptors may fail with *errno* set to [EMFILE]. This limit
 11694 constrains the number of file descriptors that a process may allocate.

11695 **RLIMIT_STACK** This is the maximum size of a process' stack in bytes. The implementation
 11696 will not automatically grow the stack beyond this limit. If this limit is
 11697 exceeded, SIGSEGV is generated for the thread. If the thread is blocking

11698 SIGSEGV, or the process is ignoring or catching SIGSEGV and has not made
 11699 arrangements to use an alternate stack, the disposition of SIGSEGV will be set
 11700 to SIG_DFL before it is generated.

11701 RLIMIT_AS This is the maximum size of a process' total available memory, in bytes. If this
 11702 limit is exceeded, the *brk()*, *malloc()*, *mmap()* and *sbrk()* functions will fail
 11703 with *errno* set to [ENOMEM]. In addition, the automatic stack growth will fail
 11704 with the effects outlined above.

11705 When using the *getrlimit()* function, if a resource limit can be represented correctly in an object
 11706 of type **rlim_t** then its representation is returned; otherwise if the value of the resource limit is
 11707 equal to that of the corresponding saved hard limit, the value returned is RLIM_SAVED_MAX;
 11708 otherwise the value returned is RLIM_SAVED_CUR.

11709 When using the *setrlimit()* function, if the requested new limit is RLIM_INFINITY the new limit
 11710 will be “no limit”; otherwise if the requested new limit is RLIM_SAVED_MAX, the new limit
 11711 will be the corresponding saved hard limit; otherwise if the requested new limit is
 11712 RLIM_SAVED_CUR, the new limit will be the corresponding saved soft limit; otherwise the new
 11713 limit will be the requested value. In addition, if the corresponding saved limit can be represented
 11714 correctly in an object of type **rlim_t** then it will be overwritten with the new limit.

11715 The result of setting a limit to RLIM_SAVED_MAX or RLIM_SAVED_CUR is unspecified unless
 11716 a previous call to *getrlimit()* returned that value as the soft or hard limit for the corresponding
 11717 resource limit.

11718 The determination of whether a limit can be correctly represented in an object of type **rlim_t** is
 11719 implementation-dependent. For example, some implementations permit a limit whose value is
 11720 greater than RLIM_INFINITY and others do not.

11721 The *exec* family of functions also cause resource limits to be saved.

11722 **RETURN VALUE**

11723 Upon successful completion, *getrlimit()* and *setrlimit()* return 0. Otherwise, these functions
 11724 return -1 and set *errno* to indicate the error.

11725 **ERRORS**

11726 The *getrlimit()* and *setrlimit()* functions will fail if:

11727 [EINVAL] An invalid *resource* was specified; or in a *setrlimit()* call, the new **rlim_cur**
 11728 exceeds the new **rlim_max**.

11729 [EPERM] The limit specified to *setrlimit()* would have raised the maximum limit value,
 11730 and the calling process does not have appropriate privileges.

11731 The *setrlimit()* function may fail if:

11732 [EINVAL] The limit specified cannot be lowered because current usage is already higher
 11733 than the limit.

11734 **EXAMPLES**

11735 None.

11736 **APPLICATION USAGE**

11737 If a process attempts to set the hard limit or soft limit for RLIMIT_NOFILE to less than the value
 11738 of _POSIX_OPEN_MAX from <limits.h>, unexpected behaviour may occur.

11739 **FUTURE DIRECTIONS**

11740 None.

11741 **SEE ALSO**

11742 *brk()*, *exec*, *fork()*, *malloc()*, *open()*, *sbrk()*, *sigaltstack()*, *sysconf()*, *ulimit()*, **<stropts.h>**,
11743 **<sys/resource.h>**.

11744 **CHANGE HISTORY**

11745 First released in Issue 4, Version 2.

11746 **Issue 5**

11747 Moved from X/OPEN UNIX extension to BASE and an APPLICATION USAGE section is added.

11748 Large File Summit extensions added.

11749 **NAME**

11750 getrusage — get information about resource utilisation

11751 **SYNOPSIS**11752 EX `#include <sys/resource.h>`11753 `int getrusage(int who, struct rusage *r_usage);`

11754

11755 **DESCRIPTION**

11756 The *getrusage()* function provides measures of the resources used by the current process or its
 11757 terminated and waited-for child processes. If the value of the *who* argument is `RUSAGE_SELF`,
 11758 information is returned about resources used by the current process. If the value of the *who*
 11759 argument is `RUSAGE_CHILDREN`, information is returned about resources used by the
 11760 terminated and waited-for children of the current process. If the child is never waited for (for
 11761 instance, if the parent has `SA_NOCLDWAIT` set or sets `SIGCHLD` to `SIG_IGN`), the resource
 11762 information for the child process is discarded and not included in the resource information
 11763 provided by *getrusage()*.

11764 The *r_usage* argument is a pointer to an object of type **struct rusage** in which the returned
 11765 information is stored.

11766 **RETURN VALUE**

11767 Upon successful completion, *getrusage()* returns 0. Otherwise, `-1` is returned, and *errno* is set to
 11768 indicate the error.

11769 **ERRORS**11770 The *getrusage()* function will fail if:11771 `[EINVAL]` The value of the *who* argument is not valid.11772 **EXAMPLES**

11773 None.

11774 **APPLICATION USAGE**

11775 None.

11776 **FUTURE DIRECTIONS**

11777 None.

11778 **SEE ALSO**11779 *exit()*, *sigaction()*, *time()*, *times()*, *wait()*, `<sys/resource.h>`.11780 **CHANGE HISTORY**

11781 First released in Issue 4, Version 2.

11782 **Issue 5**

11783 Moved from X/OPEN UNIX extension to BASE.

11784 **NAME**11785 gets — get a string from a *stdin* stream11786 **SYNOPSIS**

11787 #include <stdio.h>

11788 char *gets(char *s);

11789 **DESCRIPTION**

11790 The *gets()* function reads bytes from the standard input stream, *stdin*, into the array pointed to
 11791 by *s*, until a newline is read or an end-of-file condition is encountered. Any newline is discarded
 11792 and a null byte is placed immediately after the last byte read into the array.

11793 The *gets()* function may mark the *st_atime* field of the file associated with *stream* for update. The
 11794 *st_atime* field will be marked for update by the first successful execution of *fgetc()*, *fgets()*,
 11795 *fread()*, *getc()*, *getchar()*, *gets()*, *fscanf()* or *scanf()* using *stream* that returns data not supplied by a
 11796 prior call to *ungetc()*.

11797 **RETURN VALUE**

11798 Upon successful completion, *gets()* returns *s*. If the stream is at end-of-file, the end-of-file
 11799 indicator for the stream is set and *gets()* returns a null pointer. If a read error occurs, the error
 11800 indicator for the stream is set, *gets()* returns a null pointer and sets *errno* to indicate the error.

11801 **ERRORS**11802 Refer to *fgetc()*.11803 **EXAMPLES**

11804 None.

11805 **APPLICATION USAGE**

11806 Reading a line that overflows the array pointed to by *s* causes undefined results. The use of
 11807 *fgets()* is recommended.

11808 **FUTURE DIRECTIONS**

11809 None.

11810 **SEE ALSO**11811 *feof()*, *ferror()*, *fgets()*, <stdio.h>.11812 **CHANGE HISTORY**

11813 First released in Issue 1.

11814 Derived from Issue 1 of the SVID.

11815 **Issue 4**

11816 The following change is incorporated in this issue:

- 11817 • In the DESCRIPTION (a) the text is changed to make it clear that the function reads bytes
- 11818 rather than (possibly multi-byte) characters, and (b) the list of functions that may cause the
- 11819 *st_atime* field to be updated is revised.

11820 **NAME**

11821 getsid — get the process group ID of session leader

11822 **SYNOPSIS**11823 EX `#include <unistd.h>`11824 `pid_t getsid(pid_t pid);`

11825

11826 **DESCRIPTION**

11827 The *getsid()* function obtains the process group ID of the process that is the session leader of the
 11828 process specified by *pid*. If *pid* is (**pid_t**)0, it specifies the calling process.

11829 **RETURN VALUE**

11830 Upon successful completion, *getsid()* returns the process group ID of the session leader of the
 11831 specified process. Otherwise, it returns (**pid_t**)−1 and sets *errno* to indicate the error.

11832 **ERRORS**11833 The *getsid()* function will fail if:

11834 [EPERM] The process specified by *pid* is not in the same session as the calling process,
 11835 and the implementation does not allow access to the process group ID of the
 11836 session leader of that process from the calling process.

11837 [ESRCH] There is no process with a process ID equal to *pid*.

11838 **EXAMPLES**

11839 None.

11840 **APPLICATION USAGE**

11841 None.

11842 **FUTURE DIRECTIONS**

11843 None.

11844 **SEE ALSO**11845 *exec*, *fork()*, *getpid()*, *getpgid()*, *setpgid()*, *setsid()*, **<unistd.h>**.11846 **CHANGE HISTORY**

11847 First released in Issue 4, Version 2.

11848 **Issue 5**

11849 Moved from X/OPEN UNIX extension to BASE.

11850 **NAME**

11851 getsubopt — parse suboption arguments from a string

11852 **SYNOPSIS**

11853 EX #include <stdlib.h>

11854 int getsubopt(char ***optionp*, char * const **tokens*, char ***valuep*);

11855

11856 **DESCRIPTION**

11857 The *getsubopt()* function parses suboption arguments in a flag argument that was initially parsed
 11858 by *getopt()*. These suboption arguments must be separated by commas and may consist of
 11859 either a single token, or a token-value pair separated by an equal sign. Because commas delimit
 11860 suboption arguments in the option string, they are not allowed to be part of the suboption
 11861 arguments or the value of a suboption argument. Similarly, because the equal sign separates a
 11862 token from its value, a token must not contain an equal sign.

11863 The *getsubopt()* function takes the address of a pointer to the option argument string, a vector of
 11864 possible tokens, and the address of a value string pointer. If the option argument string at
 11865 **optionp* contains only one suboption argument, *getsubopt()* updates **optionp* to point to the null
 11866 at the end of the string. Otherwise, it isolates the suboption argument by replacing the comma
 11867 separator with a null, and updates **optionp* to point to the start of the next suboption argument.
 11868 If the suboption argument has an associated value, *getsubopt()* updates **valuep* to point to the
 11869 value's first character. Otherwise it sets **valuep* to a null pointer.

11870 The token vector is organised as a series of pointers to strings. The end of the token vector is
 11871 identified by a null pointer.

11872 When *getsubopt()* returns, if **valuep* is not a null pointer then the suboption argument processed
 11873 included a value. The calling program may use this information to determine if the presence or
 11874 lack of a value for this suboption is an error.

11875 Additionally, when *getsubopt()* fails to match the suboption argument with the tokens in the
 11876 *tokens* array, the calling program should decide if this is an error, or if the unrecognised option
 11877 should be passed on to another program.

11878 **RETURN VALUE**

11879 The *getsubopt()* function returns the index of the matched token string, or -1 if no token strings
 11880 were matched.

11881 **ERRORS**

11882 No errors are defined.

11883 **EXAMPLES**

11884 None.

11885 **APPLICATION USAGE**

11886 None.

11887 **FUTURE DIRECTIONS**

11888 None.

11889 **SEE ALSO**11890 *getopt()*, <stdlib.h>.11891 **CHANGE HISTORY**

11892 First released in Issue 4, Version 2.

11893 **Issue 5**

11894 Moved from X/OPEN UNIX extension to BASE.

|

11895 **NAME**

11896 gettimeofday — get the date and time

11897 **SYNOPSIS**

11898 EX #include <sys/time.h>

11899 int gettimeofday(struct timeval *tp, void *tzp);

11900

11901 **DESCRIPTION**

11902 The *gettimeofday()* function obtains the current time, expressed as seconds and microseconds
11903 since 00:00 Coordinated Universal Time (UTC), January 1, 1970, and stores it in the **timeval**
11904 structure pointed to by *tp*. The resolution of the system clock is unspecified.

11905 If *tzp* is not a null pointer, the behaviour is unspecified.

11906 **RETURN VALUE**

11907 The *gettimeofday()* function returns 0 and no value is reserved to indicate an error.

11908 **ERRORS**

11909 No errors are defined.

11910 **EXAMPLES**

11911 None.

11912 **APPLICATION USAGE**

11913 None.

11914 **FUTURE DIRECTIONS**

11915 None.

11916 **SEE ALSO**

11917 *ctime()*, *ftime()*, <sys/time.h>.

11918 **CHANGE HISTORY**

11919 First released in Issue 4, Version 2.

11920 **Issue 5**

11921 Moved from X/OPEN UNIX extension to BASE.

11922 **NAME**

11923 getuid — get a real user ID

11924 **SYNOPSIS**

11925 OH #include <sys/types.h>

11926 #include <unistd.h>

11927 uid_t getuid (void);

11928 **DESCRIPTION**11929 The *getuid()* function returns the real user ID of the calling process.11930 **RETURN VALUE**11931 The *getuid()* function is always successful and no return value is reserved to indicate the error.11932 **ERRORS**

11933 No errors are defined.

11934 **EXAMPLES**

11935 None.

11936 **APPLICATION USAGE**

11937 None.

11938 **FUTURE DIRECTIONS**

11939 None.

11940 **SEE ALSO**11941 *geteuid()*, *getgid()*, *setuid()*, <sys/types.h>, <unistd.h>.11942 **CHANGE HISTORY**

11943 First released in Issue 1.

11944 Derived from Issue 1 of the SVID.

11945 **Issue 4**

11946 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 11947
- The argument list is explicitly defined as **void**.

11948 Other changes are incorporated as follows:

- 11949
- 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.

11952 **NAME**

11953 getutxent, getutxid, getutxline — get user accounting database entries

11954 **SYNOPSIS**

11955 EX #include <utmpx.h>

11956 struct utmpx *getutxent(void);

11957 struct utmpx *getutxid(const struct utmpx *id);

11958 struct utmpx *getutxline(const struct utmpx *line);

11959

11960 **DESCRIPTION**11961 Refer to *endutxent()*.11962 **CHANGE HISTORY**

11963 First released in Issue 4, Version 2.

11964 **Issue 5**

11965 Moved from X/OPEN UNIX extension to BASE.

11966 **NAME**11967 getw — get a word from a stream (**LEGACY**)11968 **SYNOPSIS**

11969 EX #include <stdio.h>

11970 int getw(FILE *stream);

11971

11972 **DESCRIPTION**

11973 The *getw()* function reads the next word from the *stream*. The size of a word is the size of an **int**
 11974 and may vary from machine to machine. The *getw()* function presumes no special alignment in
 11975 the file.

11976 The *getw()* function may mark the *st_atime* field of the file associated with *stream* for update.
 11977 The *st_atime* field will be marked for update by the first successful execution of *fgetc()*, *fgets()*,
 11978 *fread()*, *getc()*, *getchar()*, *gets()*, *fscanf()* or *scanf()* using *stream* that returns data not supplied by a
 11979 prior call to *ungetc()*.

11980 This interface need not be reentrant.

11981 **RETURN VALUE**

11982 Upon successful completion, *getw()* returns the next word from the input stream pointed to by
 11983 *stream*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and *getw()*
 11984 returns EOF. If a read error occurs, the error indicator for the stream is set, *getw()* returns EOF
 11985 and sets *errno* to indicate the error.

11986 **ERRORS**11987 Refer to *fgetc()*.11988 **EXAMPLES**

11989 None.

11990 **APPLICATION USAGE**

11991 Because of possible differences in word length and byte ordering, files written using *putw()* are
 11992 implementation-dependent, and possibly cannot be read using *getw()* by a different application
 11993 or by the same application on a different processor.

11994 Because the representation of EOF is a valid integer, applications wishing to check for errors
 11995 should use *ferror()* and *feof()*.

11996 The *getw()* function is inherently byte stream-oriented and is not tenable in the context of either
 11997 multibyte character streams or wide-character streams. Application programmers are
 11998 recommended to use one of the character-based input functions instead.

11999 **FUTURE DIRECTIONS**

12000 None.

12001 **SEE ALSO**12002 *feof()*, *ferror()*, *getc()*, *putw()*, <stdio.h>, <utmpx.h>.12003 **CHANGE HISTORY**

12004 First released in Issue 1.

12005 Derived from Issue 1 of the SVID.

12006 **Issue 4**

12007 The following changes are incorporated in this issue:

- 12008 • In the DESCRIPTION, the list of functions that may cause the *st_atime* field to be updated is
- 12009 revised.

- | | | |
|-------|--|--|
| 12010 | • The APPLICATION USAGE section is amended because EOF is always a valid integer. | |
| 12011 | Issue 5 | |
| 12012 | A note indicating that this interface need not be reentrant is added to the DESCRIPTION. | |
| 12013 | Marked LEGACY. | |

12014 **NAME**

12015 getwc — get a wide character from a stream

12016 **SYNOPSIS**

12017 #include <stdio.h>

12018 #include <wchar.h>

12019 wint_t getwc(FILE **stream*);

12020 **DESCRIPTION**

12021 The *getwc()* function is equivalent to *fgetwc()*, except that if it is implemented as a macro it may
12022 evaluate *stream* more than once, so the argument should never be an expression with side effects.

12023 **RETURN VALUE**

12024 Refer to *fgetwc()*.

12025 **ERRORS**

12026 Refer to *fgetwc()*.

12027 **EXAMPLES**

12028 None.

12029 **APPLICATION USAGE**

12030 Because it may be implemented as a macro, *getwc()* may treat incorrectly a *stream* argument with
12031 side effects. In particular, *getwc(*f++)* will not necessarily work as expected. Therefore, use of
12032 this interface is not recommended; *fgetwc()* should be used instead.

12033 **FUTURE DIRECTIONS**

12034 None.

12035 **SEE ALSO**

12036 *fgetwc()*, <stdio.h>, <wchar.h>.

12037 **CHANGE HISTORY**

12038 First released as a World-wide Portability Interface in Issue 4.

12039 Derived from the MSE working draft.

12040 **Issue 5**

12041 The Optional Header (OH) marking is removed from <stdio.h>.

12042 **NAME**12043 getwchar — get a wide character from a *stdin* stream12044 **SYNOPSIS**

12045 #include <wchar.h>

12046 wint_t getwchar(void);

12047 **DESCRIPTION**12048 The *getwchar()* function is equivalent to *getwc(stdin)*.12049 **RETURN VALUE**12050 Refer to *fgetwc()*.12051 **ERRORS**12052 Refer to *fgetwc()*.12053 **EXAMPLES**

12054 None.

12055 **APPLICATION USAGE**12056 If the value returned by *getwchar()* is stored into a variable of type **wchar_t** and then compared
12057 against the **wint_t** macro WEOF, the comparison need never succeed.12058 **FUTURE DIRECTIONS**

12059 None.

12060 **SEE ALSO**12061 *fgetwc()*, *getwc()*, <wchar.h>.12062 **CHANGE HISTORY**

12063 First released as a World-wide Portability Interface in Issue 4.

12064 Derived from the MSE working draft.

12065 **NAME**

12066 getwd — get the current working directory pathname |

12067 **SYNOPSIS**

12068 EX #include <unistd.h> |

12069 char *getwd(char *path_name); |

12070 |

12071 **DESCRIPTION**12072 The *getwd()* function determines an absolute pathname of the current working directory of the
12073 calling process, and copies that pathname into the array pointed to by the *path_name* argument.12074 If the length of the pathname of the current working directory is greater than ({PATH_MAX} + 1)
12075 including the null byte, *getwd()* fails and returns a null pointer.12076 **RETURN VALUE**12077 Upon successful completion, a pointer to the string containing the absolute pathname of the
12078 current working directory is returned. Otherwise, *getwd()* returns a null pointer and the
12079 contents of the array pointed to by *path_name* are undefined.12080 **ERRORS**

12081 No errors are defined. |

12082 **EXAMPLES**

12083 None. |

12084 **APPLICATION USAGE**12085 For portability to implementations conforming to earlier versions of this specification, *getcwd()*
12086 is preferred over this function. |12087 **FUTURE DIRECTIONS**

12088 None. |

12089 **SEE ALSO**12090 *getcwd()*, <unistd.h>.12091 **CHANGE HISTORY**

12092 First released in Issue 4, Version 2. |

12093 **Issue 5**

12094 Moved from X/OPEN UNIX extension to BASE. |

12095 **NAME**

12096 glob, globfree — generate pathnames matching a pattern

12097 **SYNOPSIS**

12098 #include <glob.h>

```
12099 int glob(const char *pattern, int flags,
12100         int(*errfunc)(const char *epath, int errno), glob_t *pglob);
12101 void globfree(glob_t *pglob);
```

12102 **DESCRIPTION**

12103 The *glob()* function is a pathname generator that implements the rules defined in the XCU
 12104 specification, **Section 2.13, Pattern Matching Notation**, with optional support for rule 3 in the
 12105 XCU specification, **Section 2.13.3, Patterns Used for Filename Expansion**.

12106 The structure type **glob_t** is defined in the header <glob.h> and includes at least the following
 12107 members:

12108

12109

Member Type	Member Name	Description
size_t	gl_pathc	Count of paths matched by <i>pattern</i> .
char **	gl_pathv	Pointer to a list of matched pathnames.
size_t	gl_offs	Slots to reserve at the beginning of gl_pathv .

12110

12111

12112

12113 The argument *pattern* is a pointer to a pathname pattern to be expanded. The *glob()* function
 12114 matches all accessible pathnames against this pattern and develops a list of all pathnames that
 12115 match. In order to have access to a pathname, *glob()* requires search permission on every
 12116 component of a path except the last, and read permission on each directory of any filename
 12117 component of *pattern* that contains any of the following special characters:

12118 * ? [

12119 The *glob()* function stores the number of matched pathnames into *pglob*→**gl_pathc** and a pointer
 12120 to a list of pointers to pathnames into *pglob*→**gl_pathv**. The pathnames are in sort order as
 12121 defined by the current setting of the LC_COLLATE category, see the XBD specification, **Section**
 12122 **5.3.2, LC_COLLATE**. The first pointer after the last pathname is a null pointer. If the pattern
 12123 does not match any pathnames, the returned number of matched paths is set to 0, and the
 12124 contents of *pglob*→**gl_pathv** are implementation-dependent.

12125 It is the caller's responsibility to create the structure pointed to by *pglob*. The *glob()* function
 12126 allocates other space as needed, including the memory pointed to by **gl_pathv**. The *globfree()*
 12127 function frees any space associated with *pglob* from a previous call to *glob()*.

12128 The *flags* argument is used to control the behaviour of *glob()*. The value of *flags* is a bitwise
 12129 inclusive OR of zero or more of the following constants, which are defined in the header
 12130 <glob.h>:

12131	GLOBAL_APPEND	Append pathnames generated to the ones from a previous call to <i>glob()</i> .
12132	GLOBAL_DOOFFS	Make use of <i>pglob</i> → gl_offs . If this flag is set, <i>pglob</i> → gl_offs is used to 12133 specify how many null pointers to add to the beginning of 12134 <i>pglob</i> → gl_pathv . In other words, <i>pglob</i> → gl_pathv will point to 12135 <i>pglob</i> → gl_offs null pointers, followed by <i>pglob</i> → gl_pathc pathname 12136 pointers, followed by a null pointer. 12137 ne 2
12138	GLOBAL_ERR	Causes <i>glob()</i> to return when it encounters a directory that it cannot open 12139 or read. Ordinarily, <i>glob()</i> continues to find matches.

12142	GLOB_NOCHECK	Support rule 3 in the XCU specification, Section 2.13.3, Patterns Used for
12143		Filename Expansion . If <i>pattern</i> does not match any pathname, then
12144		<i>glob()</i> returns a list consisting of only <i>pattern</i> , and the number of matched
12145		pathnames is 1.

12147	GLOB_NOSORT	Ordinarily, <i>glob()</i> sorts the matching pathnames according to the current
12148		setting of the LC_COLLATE category, see the XBD specification, Section
12149		5.3.2, LC_COLLATE . When this flag is used the order of pathnames
12150		returned is unspecified.

12154 1. The first such call must not set GLOB_APPEND. All subsequent calls must set it.

12156 3. After the second call, `pglob->gl_pathv` points to a list containing the following:

12158 b. Pointers to the pathnames that were in the *pglob->gl_pathv* list before the call, in the
12159 same order as before.

12161 4. The count returned in *pglob->gl_pathc* will be the total number of pathnames from the
12162 two calls.

12166 If, during the search, a directory is encountered that cannot be opened or read and *errfunc* is not
12167 a null pointer, *glob()* calls (**errfunc()*) with two arguments:

12169 2. The *errno* argument is the value of *errno* from the failure, as set by *opendir()*, *readdir()* or
12170 *stat()*. (Other values may be used to report other errors not explicitly documented for
12171 those functions.)

12173	GLOB_ABORTED	The scan was stopped because GLOB_ERR was set or (<i>*errfunc()</i>)
12174		returned non-zero.

```
12177      GLOB_NOSPACE      An attempt to allocate memory failed.
```

12178 If *(*errfunc)()* is called and returns non-zero, or if the GLOB_ERR flag is set in *flags*, *glob()* stops
12179 the scan and returns GLOB_ABORTED after setting *gl_pathc* and *gl_pathv* in *pglob* to reflect the
12180 paths already scanned. If GLOB_ERR is not set and either *errfunc* is a null pointer or *(*errfunc())*
12181 returns 0, the error is ignored.

12182 **RETURN VALUE**

12183 On successful completion, *glob()* returns 0. The argument *pglob*→**gl_pathc** returns the number
 12184 of matched pathnames and the argument *pglob*→**gl_pathv** contains a pointer to a null-
 12185 terminated list of matched and sorted pathnames. However, if *pglob*→**gl_pathc** is 0, the content
 12186 of *pglob*→**gl_pathv** is undefined.

12187 The *globfree()* function returns no value.

12188 If *glob()* terminates due to an error, it returns one of the non-zero constants defined in <glob.h>.
 12189 The arguments *pglob*→**gl_pathc** and *pglob*→**gl_pathv** are still set as defined above.

12190 **ERRORS**

12191 No errors are defined.

12192 **EXAMPLES**

12193 One use of the GLOB_DOOFFS flag is by applications that build an argument list for use with
 12194 *execv()*, *execve()* or *execvp()*. Suppose, for example, that an application wants to do the
 12195 equivalent of:

```
12196     ls -l *.c
```

12197 but for some reason:

```
12198     system("ls -l *.c")
```

12199 is not acceptable. The application could obtain approximately the same result using the
 12200 sequence:

```
12201     globbuf.gl_offs = 2;  
12202     glob ("*.c", GLOB_DOOFFS, NULL, &globbuf);  
12203     globbuf.gl_pathv[0] = "ls";  
12204     globbuf.gl_pathv[1] = "-l";  
12205     execvp ("ls", &globbuf.gl_pathv[0]);
```

12206 Using the same example:

```
12207     ls -l *.c *.h
```

12208 could be approximately simulated using GLOB_APPEND as follows:

```
12209     globbuf.gl_offs = 2;  
12210     glob ("*.c", GLOB_DOOFFS, NULL, &globbuf);  
12211     glob ("*.h", GLOB_DOOFFS|GLOB_APPEND, NULL, &globbuf);  
12212     ...
```

12213 **APPLICATION USAGE**

12214 This function is not provided for the purpose of enabling utilities to perform pathname
 12215 expansion on their arguments, as this operation is performed by the shell, and utilities are
 12216 explicitly not expected to redo this. Instead, it is provided for applications that need to do
 12217 pathname expansion on strings obtained from other sources, such as a pattern typed by a user or
 12218 read from a file.

12219 If a utility needs to see if a pathname matches a given pattern, it can use *fnmatch()*.

12220 Note that **gl_pathc** and **gl_pathv** have meaning even if *glob()* fails. This allows *glob()* to report
 12221 partial results in the event of an error. However, if **gl_pathc** is 0, **gl_pathv** is unspecified even if
 12222 *glob()* did not return an error.

12223 The GLOB_NOCHECK option could be used when an application wants to expand a pathname
 12224 if wildcards are specified, but wants to treat the pattern as just a string otherwise. The *sh* utility
 12225 might use this for option-arguments, for example.

12226 The new pathnames generated by a subsequent call with GLOB_APPEND are not sorted
12227 together with the previous pathnames. This mirrors the way that the shell handles pathname
12228 expansion when multiple expansions are done on a command line.

12229 Applications that need tilde and parameter expansion should use *wordexp()*. |

12230 **FUTURE DIRECTIONS** |

12231 None.

12232 **SEE ALSO**

12233 *execv()*, *fnmatch()*, *opendir()*, *readdir()*, *stat()*, *wordexp()*, **<glob.h>**, the XCU specification.

12234 **CHANGE HISTORY**

12235 First released in Issue 4.

12236 Derived from the ISO POSIX-2 standard. |

12237 **Issue 5** |

12238 Moved from POSIX2 C-language Binding to BASE.

12239 **NAME**

12240 gmtime, gmtime_r — convert a time value to a broken-down UTC time

12241 **SYNOPSIS**

12242 #include <time.h>

12243 struct tm *gmtime(const time_t *timer);

12244 struct tm *gmtime_r(const time_t *clock, struct tm *result);

12245 **DESCRIPTION**

12246 The *gmtime()* function converts the time in seconds since the Epoch pointed to by *timer* into a
 12247 broken-down time, expressed as Coordinated Universal Time (UTC).

12248 The *gmtime()* interface need not be reentrant.

12249 The *gmtime_r()* function converts the calendar time pointed to by *clock* into a broken-down time
 12250 expressed as Coordinated Universal Time (UTC). The broken-down time is stored in the
 12251 structure referred to by *result*. The *gmtime_r()* function also returns the address of the same
 12252 structure.

12253 **RETURN VALUE**

12254 The *gmtime()* function returns a pointer to a **struct tm**.

12255 Upon successful completion, *gmtime_r()* returns the address of the structure pointed to by the
 12256 argument *result*. If an error is detected, or UTC is not available, *gmtime_r()* returns a NULL
 12257 pointer.

12258 **ERRORS**

12259 No errors are defined.

12260 **EXAMPLES**

12261 None.

12262 **APPLICATION USAGE**

12263 The *asctime()*, *ctime()*, *gmtime()* and *localtime()* functions return values in one of two static
 12264 objects: a broken-down time structure and an array of **char**. Execution of any of the functions
 12265 may overwrite the information returned in either of these objects by any of the other functions.

12266 **FUTURE DIRECTIONS**

12267 None.

12268 **SEE ALSO**

12269 *asctime()*, *clock()*, *ctime()*, *difftime()*, *localtime()*, *mktime()*, *strftime()*, *strptime()*, *time()*, *utime()*,
 12270 <time.h>.

12271 **CHANGE HISTORY**

12272 First released in Issue 1.

12273 Derived from Issue 1 of the SVID.

12274 **Issue 4**

12275 The following change is incorporated for alignment with the ISO C standard:

- 12276 • The type of argument *timer* is changed from **time_t*** to **const time_t***.

12277 Another change is incorporated as follows:

- 12278 • In the APPLICATION USAGE section, the list of functions with which this function may
 12279 interact is revised and the wording clarified.

12280 Issue 5

12281 A note indicating that the *gmtime()* interface need not be reentrant is added to the
12282 DESCRIPTION.

12283 The *gmtime_r()* function is included for alignment with the POSIX Threads Extension.

12284 **NAME**

12285 grantpt — grant access to the slave pseudo-terminal device

12286 **SYNOPSIS**

12287 EX #include <stdlib.h>

12288 int grantpt(int *fildev*);

12289

12290 **DESCRIPTION**

12291 The *grantpt()* function changes the mode and ownership of the slave pseudo-terminal device
 12292 associated with its master pseudo-terminal counter part. The *fildev* argument is a file descriptor
 12293 that refers to a master pseudo-terminal device. The user ID of the slave is set to the real UID of
 12294 the calling process and the group ID is set to an unspecified group ID. The permission mode of
 12295 the slave pseudo-terminal is set to readable and writable by the owner, and writable by the
 12296 group.

12297 The behaviour of the *grantpt()* function is unspecified if the application has installed a signal
 12298 handler to catch SIGCHLD signals

12299 **RETURN VALUE**

12300 Upon successful completion, *grantpt()* returns 0. Otherwise, it returns -1 and sets *errno* to
 12301 indicate the error.

12302 **ERRORS**12303 The *grantpt()* function may fail if:12304 [EBADF] The *fildev* argument is not a valid open file descriptor.12305 [EINVAL] The *fildev* argument is not associated with a master pseudo-terminal device.

12306 [EACCES] The corresponding slave pseudo-terminal device could not be accessed.

12307 **EXAMPLES**

12308 None.

12309 **APPLICATION USAGE**

12310 None.

12311 **FUTURE DIRECTIONS**

12312 None.

12313 **SEE ALSO**12314 *open()*, *ptsname()*, *unlockpt()*, <stdlib.h>.12315 **CHANGE HISTORY**

12316 First released in Issue 4, Version 2.

12317 **Issue 5**

12318 Moved from X/OPEN UNIX extension to BASE.

12319 The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section in
 12320 previous issues.

12321 NAME

12322 hcreate, hdestroy, hsearch — manage hash search table

12323 SYNOPSIS

```
12324 EX      #include <search.h>

12325          int hcreate(size_t nel);
12326          void hdestroy(void);
12327          ENTRY *hsearch (ENTRY item, ACTION action);
12328
```

12329 DESCRIPTION

12330 The *hcreate()*, *hdestroy()* and *hsearch()* functions manage hash search tables.

12331 The *hcreate()* function allocates sufficient space for the table, and must be called before *hsearch()*
 12332 is used. The *nel* argument is an estimate of the maximum number of entries that the table will
 12333 contain. This number may be adjusted upward by the algorithm in order to obtain certain
 12334 mathematically favourable circumstances.

12335 The *hdestroy()* function disposes of the search table, and may be followed by another call to
 12336 *hcreate()*. After the call to *hdestroy()*, the data can no longer be considered accessible.

12337 The *hsearch()* function is a hash-table search routine. It returns a pointer into a hash table
 12338 indicating the location at which an entry can be found. The *item* argument is a structure of type
 12339 **ENTRY** (defined in the *<search.h>* header) containing two pointers: *item.key* points to the
 12340 comparison key (a **char ***), and *item.data* (a **void ***) points to any other data to be associated with
 12341 that key. The comparison function used by *hsearch()* is *strcmp()*. The *action* argument is a
 12342 member of an enumeration type **ACTION** indicating the disposition of the entry if it cannot be
 12343 found in the table. **ENTER** indicates that the item should be inserted in the table at an
 12344 appropriate point. **FIND** indicates that no entry should be made. Unsuccessful resolution is
 12345 indicated by the return of a null pointer.

12346 RETURN VALUE

12347 The *hcreate()* function returns 0 if it cannot allocate sufficient space for the table, and returns
 12348 non-zero otherwise.

12349 The *hdestroy()* function returns no value.

12350 The *hsearch()* function returns a null pointer if either the action is **FIND** and the item could not
 12351 be found or the action is **ENTER** and the table is full.

12352 ERRORS

12353 The *hcreate()* and *hsearch()* functions may fail if:

12354 [ENOMEM] Insufficient storage space is available.

12355 EXAMPLES

12356 The following example will read in strings followed by two numbers and store them in a hash
 12357 table, discarding duplicates. It will then read in strings and find the matching entry in the hash
 12358 table and print it out.

```
12359      #include <stdio.h>
12360      #include <search.h>
12361      #include <string.h>
12362      struct info {          /* this is the info stored in the table */
12363          int age, room;     /* other than the key. */
12364      };
12365      #define NUM_EMPL      5000    /* # of elements in search table */
12366      int main(void)
```

```

12367 {
12368     char string_space[NUM_EMPL*20];    /* space to store strings */
12369     struct info info_space[NUM_EMPL];  /* space to store employee info*/
12370     char *str_ptr = string_space;      /* next space in string_space */
12371     struct info *info_ptr = info_space; /* next space in info_space */
12372     ENTRY item;
12373     ENTRY *found_item;    /* name to look for in table */
12374     char name_to_find[30];
12375     int i = 0;
12376     /* create table; no error checking is performed */
12377     (void) hcreate(NUM_EMPL);
12378     while (scanf("%s%d%d", str_ptr, &info_ptr->age,
12379         &info_ptr->room) != EOF && i++ < NUM_EMPL) {
12380         /* put information in structure, and structure in item */
12381         item.key = str_ptr;
12382         item.data = info_ptr;
12383         str_ptr += strlen(str_ptr) + 1;
12384         info_ptr++;
12385         /* put item into table */
12386         (void) hsearch(item, ENTER);
12387     }
12388     /* access table */
12389     item.key = name_to_find;
12390     while (scanf("%s", item.key) != EOF) {
12391         if ((found_item = hsearch(item, FIND)) != NULL) {
12392             /* if item is in the table */
12393             (void)printf("found %s, age = %d, room = %d\n",
12394                 found_item->key,
12395                 ((struct info *)found_item->data)->age,
12396                 ((struct info *)found_item->data)->room);
12397         } else
12398             (void)printf("no such employee %s\n", name_to_find);
12399     }
12400     return 0;
12401 }

```

12402 APPLICATION USAGE

12403 The *hcreate()* and *hsearch()* functions may use *malloc()* to allocate space.

12404 FUTURE DIRECTIONS

12405 None.

12406 SEE ALSO

12407 *bsearch()*, *lsearch()*, *malloc()*, *strcmp()*, *tsearch()*, <search.h>.

12408 **CHANGE HISTORY**

12409 First released in Issue 1.

12410 Derived from Issue 1 of the SVID.

12411 **Issue 4**

12412 The following changes are incorporated in this issue:

- 12413 • In the SYNOPSIS section, the type of argument *nel* in the declaration of *hcreate()* is changed |
- 12414 from **unsigned** to **size_t**, and the argument list is explicitly defined as **void** in the declaration |
- 12415 of *hdestroy()*. |
- 12416 • In the DESCRIPTION, the type of the comparison key is explicitly defined as **char ***, the type |
- 12417 of *item.data* is explicitly defined as **void***, and a statement is added indicating that *hsearch()* |
- 12418 uses *strcmp()* as the comparison function. |
- 12419 • In the EXAMPLES section, the sample code is updated to use ISO C syntax. |
- 12420 • An ERRORS section is added and [ENOMEM] is defined as an error that may be returned by |
- 12421 *hsearch()* and *hcreate()*. |

12422 **NAME**

12423 hypot — Euclidean distance function

12424 **SYNOPSIS**

12425 EX #include <math.h>

12426 double hypot(double x, double y);

12427

12428 **DESCRIPTION**12429 The *hypot()* function computes the length of the hypotenuse of a right-angled triangle:12430 $\sqrt{x^2+y^2}$ 12431 An application wishing to check for error situations should set *errno* to 0 before calling *hypot()*.12432 If *errno* is non-zero on return, or the return value is HUGE_VAL or NaN, an error has occurred.12433 **RETURN VALUE**12434 Upon successful completion, *hypot()* returns the length of the hypotenuse of a right angled triangle with sides of length *x* and *y*.12436 If the result would cause overflow, HUGE_VAL is returned and *errno* may be set to [ERANGE].12437 If *x* or *y* is NaN, NaN is returned. and *errno* may be set to [EDOM].12438 If the correct result would cause underflow, 0 is returned and *errno* may be set to [ERANGE].12439 **ERRORS**12440 The *hypot()* function may fail if:12441 [EDOM] The value of *x* or *y* is NaN.

12442 [ERANGE] The result overflows or underflows.

12443 No other errors will occur.

12444 **EXAMPLES**

12445 None.

12446 **APPLICATION USAGE**12447 The *hypot()* function takes precautions against overflow during intermediate steps of the computation. If the calculated result would still overflow a double, then *hypot()* returns HUGE_VAL.12450 **FUTURE DIRECTIONS**

12451 None.

12452 **SEE ALSO**12453 *isnan()*, *sqrt()*, <math.h>.12454 **CHANGE HISTORY**

12455 First released in Issue 1.

12456 Derived from Issue 1 of the SVID.

12457 **Issue 4**

12458 The following changes are incorporated in this issue:

- 12459 • References to *matherr()* are removed.
- 12460 • The RETURN VALUE and ERRORS sections are substantially rewritten to rationalise error handling in the mathematics functions.

12462 **Issue 5**

12463 The DESCRIPTION is updated to indicate how an application should check for an error. This
12464 text was previously published in the APPLICATION USAGE section.

12465 **NAME**

12466 iconv — codeset conversion function

12467 **SYNOPSIS**

12468 EX #include <iconv.h>

```
12469       size_t iconv(iconv_t cd, const char **inbuf, size_t *inbytesleft,
12470                    char **outbuf, size_t *outbytesleft);
```

12471

12472 **DESCRIPTION**

12473 The *iconv()* function converts the sequence of characters from one codeset, in the array specified
 12474 by *inbuf*, into a sequence of corresponding characters in another codeset, in the array specified
 12475 by *outbuf*. The codesets are those specified in the *iconv_open()* call that returned the conversion
 12476 descriptor, *cd*. The *inbuf* argument points to a variable that points to the first character in the
 12477 input buffer and *inbytesleft* indicates the number of bytes to the end of the buffer to be converted.
 12478 The *outbuf* argument points to a variable that points to the first available byte in the output
 12479 buffer and *outbytesleft* indicates the number of the available bytes to the end of the buffer.

12480 For state-dependent encodings, the conversion descriptor *cd* is placed into its initial shift state by
 12481 a call for which *inbuf* is a null pointer, or for which *inbuf* points to a null pointer. When *iconv()* is
 12482 called in this way, and if *outbuf* is not a null pointer or a pointer to a null pointer, and *outbytesleft*
 12483 points to a positive value, *iconv()* will place, into the output buffer, the byte sequence to change
 12484 the output buffer to its initial shift state. If the output buffer is not large enough to hold the
 12485 entire reset sequence, *iconv()* will fail and set *errno* to [E2BIG]. Subsequent calls with *inbuf* as
 12486 other than a null pointer or a pointer to a null pointer cause the conversion to take place from
 12487 the current state of the conversion descriptor.

12488 If a sequence of input bytes does not form a valid character in the specified codeset, conversion
 12489 stops after the previous successfully converted character. If the input buffer ends with an
 12490 incomplete character or shift sequence, conversion stops after the previous successfully
 12491 converted bytes. If the output buffer is not large enough to hold the entire converted input,
 12492 conversion stops just prior to the input bytes that would cause the output buffer to overflow.
 12493 The variable pointed to by *inbuf* is updated to point to the byte following the last byte
 12494 successfully used in the conversion. The value pointed to by *inbytesleft* is decremented to reflect
 12495 the number of bytes still not converted in the input buffer. The variable pointed to by *outbuf* is
 12496 updated to point to the byte following the last byte of converted output data. The value pointed
 12497 to by *outbytesleft* is decremented to reflect the number of bytes still available in the output buffer.
 12498 For state-dependent encodings, the conversion descriptor is updated to reflect the shift state in
 12499 effect at the end of the last successfully converted byte sequence.

12500 If *iconv()* encounters a character in the input buffer that is valid, but for which an identical
 12501 character does not exist in the target codeset, *iconv()* performs an implementation-dependent
 12502 conversion on this character.

12503 **RETURN VALUE**

12504 The *iconv()* function updates the variables pointed to by the arguments to reflect the extent of
 12505 the conversion and returns the number of non-identical conversions performed. If the entire
 12506 string in the input buffer is converted, the value pointed to by *inbytesleft* will be 0. If the input
 12507 conversion is stopped due to any conditions mentioned above, the value pointed to by *inbytesleft*
 12508 will be non-zero and *errno* is set to indicate the condition. If an error occurs *iconv()* returns
 12509 (**size_t**)−1 and sets *errno* to indicate the error.

12510 **ERRORS**12511 The *iconv()* function will fail if:12512 [EILSEQ] Input conversion stopped due to an input byte that does not belong to the
12513 input codeset.

12514 [E2BIG] Input conversion stopped due to lack of space in the output buffer.

12515 [EINVAL] Input conversion stopped due to an incomplete character or shift sequence at
12516 the end of the input buffer.12517 The *iconv()* function may fail if:12518 [EBADF] iX EBADF The *cd* argument is not a valid open conversion descriptor.12519 **EXAMPLES**

12520 None.

12521 **APPLICATION USAGE**

12522 The *inbuf* argument indirectly points to the memory area which contains the conversion input
 12523 data. The *outbuf* argument indirectly points to the memory area which is to contain the result of
 12524 the conversion. The objects indirectly pointed to by *inbuf* and *outbuf* are not restricted to
 12525 containing data that is directly representable in the ISO C language **char** data type. The type of
 12526 *inbuf* and *outbuf*, **char ****, does not imply that the objects pointed to are interpreted as null-
 12527 terminated C strings or arrays of characters. Any interpretation of a byte sequence that
 12528 represents a character in a given character set encoding scheme is done internally within the
 12529 codeset converters. For example, the area pointed to indirectly by *inbuf* and/or *outbuf* can
 12530 contain all zero octets that are not interpreted as string terminators but as coded character data
 12531 according to the respective codeset encoding scheme. The type of the data (**char**, **short int**, **long**
 12532 **int**, and so on) read or stored in the objects is not specified, but may be inferred for both the
 12533 input and output data by the converters determined by the *fromcode* and *toencode* arguments of
 12534 *iconv_open()*.

12535 Regardless of the data type inferred by the converter, the size of the remaining space in both
 12536 input and output objects (the *inbytesleft* and *outbytesleft* arguments) is always measured in bytes.

12537 For implementations that support the conversion of state-dependent encodings, the conversion
 12538 descriptor must be able to accurately reflect the shift-state in effect at the end of the last
 12539 successful conversion. It is not required that the conversion descriptor itself be updated, which
 12540 would require it to be a pointer type. Thus, implementations are free to implement the
 12541 descriptor as a handle (other than a pointer type) by which the conversion information can be
 12542 accessed and updated.

12543 **FUTURE DIRECTIONS**

12544 None.

12545 **SEE ALSO**12546 *iconv_open()*, *iconv_close()*, <**iconv.h**>.12547 **CHANGE HISTORY**

12548 First released in Issue 4.

12549 Derived from the HP-UX manual.

12550 **NAME**

12551 iconv_close — codeset conversion deallocation function

12552 **SYNOPSIS**12553 EX `#include <iconv.h>`12554 `int iconv_close(iconv_t cd);`

12555

12556 **DESCRIPTION**

12557 The *iconv_close()* function deallocates the conversion descriptor *cd* and all other associated
12558 resources allocated by *iconv_open()*.

12559 If a file descriptor is used to implement the type **iconv_t**, that file descriptor will be closed.

12560 **RETURN VALUE**

12561 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate
12562 the error.

12563 **ERRORS**

12564 The *iconv_close()* function may fail if:

12565 [EBADF] The conversion descriptor is invalid.

12566 **EXAMPLES**

12567 None.

12568 **APPLICATION USAGE**

12569 None.

12570 **FUTURE DIRECTIONS**

12571 None.

12572 **SEE ALSO**

12573 *iconv()*, *iconv_open()*, <iconv.h>.

12574 **CHANGE HISTORY**

12575 First released in Issue 4.

12576 Derived from the HP-UX manual.

12577 **NAME**

12578 iconv_open — codeset conversion allocation function

12579 **SYNOPSIS**12580 EX `#include <iconv.h>`12581 `iconv_t iconv_open(const char *tocode, const char *fromcode);`

12582

12583 **DESCRIPTION**

12584 The *iconv_open()* function returns a conversion descriptor that describes a conversion from the
 12585 codeset specified by the string pointed to by the *fromcode* argument to the codeset specified by
 12586 the string pointed to by the *tocode* argument. For state-dependent encodings, the conversion
 12587 descriptor will be in a codeset-dependent initial shift state, ready for immediate use with *iconv()*.

12588 Settings of *fromcode* and *tocode* and their permitted combinations are implementation-dependent.

12589 A conversion descriptor remains valid in a process until that process closes it.

12590 If a file descriptor is used to implement conversion descriptors, the FD_CLOEXEC flag will be
 12591 set; see <fcntl.h>.

12592 **RETURN VALUE**

12593 Upon successful completion, *iconv_open()* returns a conversion descriptor for use on subsequent
 12594 calls to *iconv()*. Otherwise *iconv_open()* returns (**iconv_t**)-1 and sets *errno* to indicate the error.

12595 **ERRORS**

12596 The *iconv_open()* function may fail if:

12597 [EMFILE] {OPEN_MAX} files descriptors are currently open in the calling process.

12598 [ENFILE] Too many files are currently open in the system.

12599 [ENOMEM] Insufficient storage space is available.

12600 [EINVAL] The conversion specified by *fromcode* and *tocode* is not supported by the
 12601 implementation.

12602 **EXAMPLES**

12603 None.

12604 **APPLICATION USAGE**

12605 Some implementations of *iconv_open()* use *malloc()* to allocate space for internal buffer areas.
 12606 The *iconv_open()* function may fail if there is insufficient storage space to accommodate these
 12607 buffers.

12608 Portable applications must assume that conversion descriptors are not valid after a call to one of
 12609 the *exec* functions.

12610 **FUTURE DIRECTIONS**

12611 None.

12612 **SEE ALSO**

12613 *iconv()*, *iconv_close()*, <iconv.h>.

12614 **CHANGE HISTORY**

12615 First released in Issue 4.

12616 Derived from the HP-UX manual.

12617 **NAME**

12618 ilogb — return an unbiased exponent

12619 **SYNOPSIS**

12620 EX #include <math.h>

12621 int ilogb (double x)

12622

12623 **DESCRIPTION**

12624 The *ilogb()* function returns the exponent part of *x*. Formally, the return value is the integral part of $\log_r |x|$ as a signed integral value, for non-zero *x*, where *r* is the radix of the machine's floating point arithmetic.

12627 The call *ilogb(x)* is equivalent to *(int)logb(x)*.

12628 **RETURN VALUE**

12629 Upon successful completion, *ilogb()* returns the exponent part of *x*.

12630 If *x* is 0, then *ilogb()* returns `-INT_MIN`. If *x* is NaN or $\pm\text{Inf}$, then *ilogb()* returns `INT_MAX`.

12631 **ERRORS**

12632 No errors are defined.

12633 **EXAMPLES**

12634 None.

12635 **APPLICATION USAGE**

12636 None.

12637 **FUTURE DIRECTIONS**

12638 None.

12639 **SEE ALSO**

12640 *logb()*, <math.h>.

12641 **CHANGE HISTORY**

12642 First released in Issue 4, Version 2.

12643 **Issue 5**

12644 Moved from X/OPEN UNIX extension to BASE.

12645 **NAME**

12646 index — character string operations

12647 **SYNOPSIS**12648 EX `#include <strings.h>`12649 `char *index(const char *s, int c);`

12650

12651 **DESCRIPTION**12652 The *index()* function is identical to *strchr()*.12653 **RETURN VALUE**12654 See *strchr()*.12655 **ERRORS**12656 See *strchr()*.12657 **EXAMPLES**

12658 None.

12659 **APPLICATION USAGE**12660 For portability to implementations conforming to earlier versions of this specification, *strchr()* is
12661 preferred over this function.12662 **FUTURE DIRECTIONS**

12663 None.

12664 **SEE ALSO**12665 *strchr()*, *<strings.h>*.12666 **CHANGE HISTORY**

12667 First released in Issue 4, Version 2.

12668 **Issue 5**

12669 Moved from X/OPEN UNIX extension to BASE.

12670 **NAME**

12671 initstate, random, setstate, srandom — pseudorandom number functions

12672 **SYNOPSIS**

12673 EX #include <stdlib.h>

12674 char *initstate(unsigned int seed, char *state, size_t size);

12675 long random(void);

12676 char *setstate(const char *state);

12677 void srandom(unsigned int seed);

12678

12679 **DESCRIPTION**

12680 The *random()* function uses a non-linear additive feedback random-number generator
 12681 employing a default state array size of 31 long integers to return successive pseudo-random
 12682 numbers in the range from 0 to $2^{31}-1$. The period of this random-number generator is
 12683 approximately $16 \times (2^{31}-1)$. The size of the state array determines the period of the random-
 12684 number generator. Increasing the state array size increases the period.

12685 With 256 bytes of state information, the period of the random-number generator is greater than
 12686 2^{69} .

12687 Like *rand()*, *random()* produces by default a sequence of numbers that can be duplicated by
 12688 calling *srandom()* with 1 as the seed.

12689 The *srandom()* function initialises the current state array using the value of *seed*.

12690 The *initstate()* and *setstate()* functions handle restarting and changing random-number
 12691 generators. The *initstate()* function allows a state array, pointed to by the *state* argument, to be
 12692 initialised for future use. The *size* argument, which specifies the size in bytes of the state array, is
 12693 used by *initstate()* to decide what type of random-number generator to use; the larger the state
 12694 array, the more random the numbers. Values for the amount of state information are 8, 32, 64,
 12695 128, and 256 bytes. Other values greater than 8 bytes are rounded down to the nearest one of
 12696 these values. For values greater than or equal to 8, or less than 32 *random()* uses a simple linear
 12697 congruential random number generator. The *seed* argument specifies a starting point for the
 12698 random-number sequence and provides for restarting at the same point. The *initstate()* function
 12699 returns a pointer to the previous state information array.

12700 If *initstate()* has not been called, then *random()* behaves as though *initstate()* had been called
 12701 with *seed* = 1 and *size* = 128.

12702 If *initstate()* is called with $8 \leq \textit{size} < 32$, then *random()* uses a simple linear congruential
 12703 random number generator.

12704 Once a state has been initialised, *setstate()* allows switching between state arrays. The array
 12705 defined by the *state* argument is used for further random-number generation until *initstate()* is
 12706 called or *setstate()* is called again. The *setstate()* function returns a pointer to the previous state
 12707 array.

12708 **RETURN VALUE**

12709 If *initstate()* is called with *size* less than 8, it returns NULL.

12710 The *random()* function returns the generated pseudo-random number.

12711 The *srandom()* function returns no value.

12712 Upon successful completion, *initstate()* and *setstate()* return a pointer to the previous state array.
 12713 Otherwise, a null pointer is returned.

12714 **ERRORS**

12715 No errors are defined.

12716 **EXAMPLES**

12717 None.

12718 **APPLICATION USAGE**

12719 After initialisation, a state array can be restarted at a different point in one of two ways:

- 12720 • The *initstate()* function can be used, with the desired seed, state array, and size of the array.
- 12721 • The *setstate()* function, with the desired state, can be used, followed by *srandom()* with the
- 12722 desired seed. The advantage of using both of these functions is that the size of the state array
- 12723 does not have to be saved once it is initialised.

12724 Although some implementations of *random()* have written messages to standard error, such

12725 implementations do not conform to this specification.

12726 Issue 5 restores the historical behaviour of this function.

12727 Threaded applications should use *rand_r()*, *erand48()*, *nrand48()* or *jrand48()* instead of

12728 *random()* when an independent random number sequence in multiple threads is required.

12729 **FUTURE DIRECTIONS**

12730 None.

12731 **SEE ALSO**12732 *drand48()*, *rand()*, <stdlib.h>.12733 **CHANGE HISTORY**

12734 First released in Issue 4, Version 2.

12735 **Issue 5**

12736 Moved from X/OPEN UNIX extension to BASE.

12737 In the DESCRIPTION, the phrase "values smaller than 8" is replaced with "values greater than or

12738 equal to 8, or less than 32", "size<8" is replaced with "size>=8 and <32", and a new first paragraph

12739 is added to the RETURN VALUE section. A note is added to the APPLICATION USAGE

12740 indicating that these changes restore the historical behaviour of the function.

12741 NAME

12742 insque, remque — insert or remove an element in a queue

12743 SYNOPSIS

12744 EX

```
#include <search.h>
```

12745

```
void insque(void *element, void *pred);
```

12746

```
void remque(void *element);
```

12747

12748 DESCRIPTION

12749 The *insque()* and *remque()* functions manipulate queues built from doubly-linked lists. The
 12750 queue can be either circular or linear. An application using *insque()* or *remque()* must define a
 12751 structure in which the first two members of the structure are pointers to the same type of
 12752 structure, and any further members are application-specific. The first member of the structure is
 12753 a forward pointer to the next entry in the queue. The second member is a backward pointer to
 12754 the previous entry in the queue. If the queue is linear, the queue is terminated with null
 12755 pointers. The names of the structure and of the pointer members are not subject to any special
 12756 restriction.

12757 The *insque()* function inserts the element pointed to by *element* into a queue immediately after
 12758 the element pointed to by *pred*.

12759 The *remque()* function removes the element pointed to by *element* from a queue.

12760 If the queue is to be used as a linear list, invoking *insque(&element, NULL)*, where *element* is the
 12761 initial element of the queue, will initialise the forward and backward pointers of *element* to null
 12762 pointers.

12763 If the queue is to be used as a circular list, the application must initialise the forward pointer and
 12764 the backward pointer of the initial element of the queue to the element's own address.

12765 RETURN VALUE

12766 The *insque()* and *remque()* functions do not return a value.

12767 ERRORS

12768 No errors are defined.

12769 EXAMPLES

12770 None.

12771 APPLICATION USAGE

12772 The historical implementations of these functions described the arguments as being of type
 12773 **struct qelem *** rather than as being of type **void *** as defined here. In those implementations,
 12774 **struct qelem** was commonly defined in **<search.h>** as:

```
12775 struct qelem {
12776     struct qelem *q_forw;
12777     struct qelem *q_back;
12778 };
```

12779 Applications using these functions, however, were never able to use this structure directly since
 12780 it provided no room for the actual data contained in the elements. Most applications defined
 12781 structures that contained the two pointers as the initial elements and also provided space for, or
 12782 pointers to, the object's data. Applications that used these functions to update more than one
 12783 type of table also had the problem of specifying two or more different structures with the same
 12784 name, if they literally used **struct qelem** as specified.

12785 As described here, the implementations were actually expecting a structure type where the first
12786 two members were forward and backward pointers to structures. With C compilers that didn't
12787 provide function prototypes, applications used structures as specified in the DESCRIPTION
12788 above and the compiler did what the application expected.

12789 If this method had been carried forward with an ISO C compiler and the historical function
12790 prototype, most applications would have to be modified to cast pointers to the structures
12791 actually used to be pointers to **struct qelem** to avoid compilation warnings. By specifying **void ***
12792 as the argument type, applications won't need to change (unless they specifically referenced
12793 **struct qelem** and depended on it being defined in **<search.h>**).

12794 FUTURE DIRECTIONS

12795 None.

12796 SEE ALSO

12797 **<search.h>**.

12798 CHANGE HISTORY

12799 First released in Issue 4, Version 2.

12800 Issue 5

12801 Moved from X/OPEN UNIX extension to BASE.

12802 **NAME**12803 **ioctl** — control a STREAMS device12804 **SYNOPSIS**12805 **EX** #include <stropts.h>12806 int ioctl(int *fildev*, int *request*, ... /* *arg* */);

12807

12808 **DESCRIPTION**

12809 The *ioctl()* function performs a variety of control functions on STREAMS devices. For non-
 12810 STREAMS devices, the functions performed by this call are unspecified. The *request* argument
 12811 and an optional third argument (with varying type) are passed to and interpreted by the
 12812 appropriate part of the STREAM associated with *fildev*.

12813 The *fildev* argument is an open file descriptor that refers to a device.

12814 The *request* argument selects the control function to be performed and will depend on the
 12815 STREAMS device being addressed.

12816 The *arg* argument represents additional information that is needed by this specific STREAMS
 12817 device to perform the requested function. The type of *arg* depends upon the particular control
 12818 request, but it is either an integer or a pointer to a device-specific data structure.

12819 The *ioctl()* commands applicable to STREAMS, their arguments, and error statuses that apply to
 12820 each individual command are described below.

12821 The following *ioctl()* commands, with error values indicated, are applicable to all STREAMS
 12822 files:

12823 **I_PUSH** Pushes the module whose name is pointed to by *arg* onto the top of the
 12824 current STREAM, just below the STREAM head. It then calls the *open()*
 12825 function of the newly-pushed module.

12826 The *ioctl()* function with the I_PUSH command will fail if:

12827 [EINVAL] Invalid module name.

12828 [ENXIO] Open function of new module failed.

12829 [ENXIO] Hangup received on *fildev*.

12830 **I_POP** Removes the module just below the STREAM head of the STREAM pointed to
 12831 by *fildev*. The *arg* argument should be 0 in an I_POP request.

12832 The *ioctl()* function with the I_POP command will fail if:

12833 [EINVAL] No module present in the STREAM.

12834 [ENXIO] Hangup received on *fildev*.

12835 **I_LOOK** Retrieves the name of the module just below the STREAM head of the
 12836 STREAM pointed to by *fildev*, and places it in a character string pointed to by
 12837 *arg*. The buffer pointed to by *arg* should be at least FMNAMESZ+1 bytes long,
 12838 where FMNAMESZ is defined in <stropts.h>.

12839 The *ioctl()* function with the I_LOOK command will fail if:

12840 [EINVAL] No module present in the STREAM.

12841 **I_FLUSH** This request flushes read and/or write queues, depending on the value of *arg*.
 12842 Valid *arg* values are:

12843		FLUSHR	Flush all read queues.
12844		FLUSHW	Flush all write queues.
12845		FLUSHRW	Flush all read and all write queues.
12846		The <i>ioctl()</i> function with the <i>I_FLUSH</i> command will fail if:	
12847		[EINVAL]	Invalid <i>arg</i> value.
12848		[EAGAIN] or [ENOSR]	Unable to allocate buffers for flush message.
12849			
12850		[ENXIO]	Hangup received on <i>fildev</i> .
12851	I_FLUSHBAND	Flushes a particular band of messages. The <i>arg</i> argument points to a bandinfo structure. The bi_flag member may be one of FLUSHR, FLUSHW, or FLUSHRW as described above. The bi_pri member determines the priority band to be flushed.	
12852			
12853			
12854			
12855	I_SETSIG	Requests that the STREAMS implementation send the SIGPOLL signal to the calling process when a particular event has occurred on the STREAM associated with <i>fildev</i> . I_SETSIG supports an asynchronous processing capability in STREAMS. The value of <i>arg</i> is a bitmask that specifies the events for which the process should be signaled. It is the bitwise-OR of any combination of the following constants:	
12856			
12857			
12858			
12859			
12860			
12861		S_RDNORM	A normal (priority band set to 0) message has arrived at the head of a STREAM head read queue. A signal will be generated even if the message is of zero length.
12862			
12863			
12864		S_RDBAND	A message with a non-zero priority band has arrived at the head of a STREAM head read queue. A signal will be generated even if the message is of zero length.
12865			
12866			
12867		S_INPUT	A message, other than a high-priority message, has arrived at the head of a STREAM head read queue. A signal will be generated even if the message is of zero length.
12868			
12869			
12870		S_HIPRI	A high-priority message is present on a STREAM head read queue. A signal will be generated even if the message is of zero length.
12871			
12872			
12873		S_OUTPUT	The write queue for normal data (priority band 0) just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) normal data downstream.
12874			
12875			
12876			
12877		S_WRNORM	Same as S_OUTPUT.
12878		S_WRBAND	The write queue for a non-zero priority band just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) priority data downstream.
12879			
12880			
12881			
12882		S_MSG	A STREAMS signal message that contains the SIGPOLL signal has reached the front of the STREAM head read queue.
12883			
12884			
12885		S_ERROR	Notification of an error condition has reached the STREAM head.
12886			

12887	S_HANGUP	Notification of a hangup has reached the STREAM head.
12888	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.
12889		
12890		
12891		If <i>arg</i> is 0, the calling process will be unregistered and will not receive further SIGPOLL signals for the stream associated with <i>fildev</i> .
12892		
12893		Processes that wish to receive SIGPOLL signals must 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 will be signaled when the event occurs.
12894		
12895		
12896		
12897		The <i>ioctl()</i> function with the I_SETSIG command will fail if:
12898	[EINVAL]	The value of <i>arg</i> is invalid.
12899	[EINVAL]	The value of <i>arg</i> is 0 and the calling process is not registered to receive the SIGPOLL signal.
12900		
12901	[EAGAIN]	There were insufficient resources to store the signal request.
12902	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 <i>int</i> pointed to by <i>arg</i> , where the events are those specified in the description of I_SETSIG above.
12903		
12904		
12905		
12906		The <i>ioctl()</i> function with the I_GETSIG command will fail if:
12907	[EINVAL]	Process is not registered to receive the SIGPOLL signal.
12908	I_FIND	This request compares the names of all modules currently present in the STREAM to the name pointed to by <i>arg</i> , and returns 1 if the named module is present in the STREAM, or returns 0 if the named module is not present.
12909		
12910		
12911		The <i>ioctl()</i> function with the I_FIND command will fail if:
12912	[EINVAL]	<i>arg</i> does not contain a valid module name.
12913	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 <i>getmsg()</i> except that this command does not remove the message from the queue. The <i>arg</i> argument points to a strpeek structure.
12914		
12915		
12916		
12917		The maxlen member in the ctlbuf and databuf structures must be 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 <i>getmsg()</i> . If the process sets flags to RS_HIPRI, for example, I_PEEK will only look for a high-priority message on the STREAM head read queue.
12918		
12919		
12920		
12921		
12922		
12923		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.
12924		
12925		
12926		
12927		
12928		
12929	I_SRDOPT	Sets the read mode using the value of the argument <i>arg</i> . Read modes are described in <i>read()</i> . Valid <i>arg</i> flags are:
12930		

12931	RNORM	Byte-stream mode, the default.
12932	RMSGD	Message-discard mode.
12933	RMSGN	Message-nondiscard mode.
12934	The bitwise inclusive OR of RMSGD and RMSGN will return [EINVAL]. The	
12935	bitwise inclusive OR of RNORM and either RMSGD or RMSGN will result in	
12936	the other flag overriding RNORM which is the default.	
12937	In addition, treatment of control messages by the STREAM head may be	
12938	changed by setting any of the following flags in <i>arg</i> :	
12939	RPROTNORM	Fail <i>read()</i> with [EBADMSG] if a message containing a
12940		control part is at the front of the STREAM head read queue.
12941	RPROTDAT	Deliver the control part of a message as data when a
12942		process issues a <i>read()</i> .
12943	RPROTDIS	Discard the control part of a message, delivering any data
12944		portion, when a process issues a <i>read()</i> .
12945	The <i>ioctl()</i> function with the I_SRDOPT command will fail if:	
12946	[EINVAL]	The <i>arg</i> argument is not valid.
12947	I_GRDOPT	Returns the current read mode setting as, described above, in an int pointed to
12948		by the argument <i>arg</i> . Read modes are described in <i>read()</i> .
12949	I_NREAD	Counts the number of data bytes in the data part of the first message on the
12950		STREAM head read queue and places this value in the int pointed to by <i>arg</i> .
12951		The return value for the command is the number of messages on the STREAM
12952		head read queue. For example, if 0 is returned in <i>arg</i> , but the <i>ioctl()</i> return
12953		value is greater than 0, this indicates that a zero-length message is next on the
12954		queue.
12955	I_FDINSERT	Creates a message from specified buffer(s), adds information about another
12956		STREAM, and sends the message downstream. The message contains a
12957		control part and an optional data part. The data and control parts to be sent
12958		are distinguished by placement in separate buffers, as described below. The
12959		<i>arg</i> argument points to a strfdinsert structure.
12960	The len member in the ctlbuf strbuf structure must be set to the size of a	
12961	t_uscalar_t plus the number of bytes of control information to be sent with the	
12962	message. The fildev member specifies the file descriptor of the other	
12963	STREAM, and the offset member, which must be suitably aligned for use as a	
12964	t_uscalar_t , specifies the offset from the start of the control buffer where	
12965	I_FDINSERT will store a t_uscalar_t whose interpretation is specific to the	
12966	STREAM end. The len member in the databuf strbuf structure must be set to	
12967	the number of bytes of data information to be sent with the message, or to 0 if	
12968	no data part is to be sent.	
12969	The flags member specifies the type of message to be created. A normal	
12970	message is created if flags is set to 0, and a high-priority message is created if	
12971	flags is set to RS_HIPRI. For non-priority messages, I_FDINSERT will block if	
12972	the STREAM write queue is full due to internal flow control conditions. For	
12973	priority messages, I_FDINSERT does not block on this condition. For non-	
12974	priority messages, I_FDINSERT does not block when the write queue is full	
12975	and O_NONBLOCK is set. Instead, it fails and sets <i>errno</i> to [EAGAIN].	

12976		I_FDINSERT also blocks, unless prevented by lack of internal resources,
12977		waiting for the availability of message blocks in the STREAM, regardless of
12978		priority or whether O_NONBLOCK has been specified. No partial message is
12979		sent.
12980		The <i>ioctl()</i> function with the I_FDINSERT command will fail if:
12981	[EAGAIN]	A non-priority message is specified, the O_NONBLOCK
12982		flag is set, and the STREAM write queue is full due to
12983		internal flow control conditions.
12984	[EAGAIN] or [ENOSR]	
12985		Buffers can not be allocated for the message that is to be
12986		created.
12987	[EINVAL]	One of the following:
12988		• The <i>fildev</i> member of the strfdinsert structure is not a
12989		valid, open STREAM file descriptor.
12990		• The size of a t_uscalar_t plus <i>offset</i> is greater than the <i>len</i>
12991		member for the buffer specified through <i>ctlptr</i> .
12992		• The <i>offset</i> member does not specify a properly-aligned
12993		location in the data buffer.
12994		• An undefined value is stored in flags .
12995	[ENXIO]	Hangup received on the STREAM identified by either the
12996		<i>fildev</i> argument or the <i>fildev</i> member of the strfdinsert
12997		structure.
12998	[ERANGE]	The <i>len</i> member for the buffer specified through <i>databuf</i>
12999		does not fall within the range specified by the maximum
13000		and minimum packet sizes of the topmost STREAM module
13001		or the <i>len</i> member for the buffer specified through <i>databuf</i> is
13002		larger than the maximum configured size of the data part of
13003		a message; or the <i>len</i> member for the buffer specified
13004		through <i>ctlbuf</i> is larger than the maximum configured size
13005		of the control part of a message.
13006	I_STR	Constructs an internal STREAMS <i>ioctl()</i> message from the data pointed to by
13007		<i>arg</i> , and sends that message downstream.
13008		This mechanism is provided to send <i>ioctl()</i> requests to downstream modules
13009		and drivers. It allows information to be sent with <i>ioctl()</i> , and returns to the
13010		process any information sent upstream by the downstream recipient. I_STR
13011		blocks until the system responds with either a positive or negative
13012		acknowledgement message, or until the request "times out" after some period
13013		of time. If the request times out, it fails with <i>errno</i> set to [ETIME].
13014		At most, one I_STR can be active on a STREAM. Further I_STR calls will block
13015		until the active I_STR completes at the STREAM head. The default timeout
13016		interval for these requests is 15 seconds. The O_NONBLOCK flag has no
13017		effect on this call.
13018		To send requests downstream, <i>arg</i> must point to a strioc structure.
13019		The ic_cmd member is the internal <i>ioctl()</i> command intended for a
13020		downstream module or driver and ic_timeout is the number of seconds (–1 =

13021		infinite, 0 = use implementation-dependent timeout interval, >0 = as
13022		specified) an I_STR request will wait for acknowledgement before timing out.
13023		ic_len is the number of bytes in the data argument, and ic_dp is a pointer to
13024		the data argument. The ic_len member has two uses: on input, it contains the
13025		length of the data argument passed in, and on return from the command, it
13026		contains the number of bytes being returned to the process (the buffer pointed
13027		to by ic_dp should be large enough to contain the maximum amount of data
13028		that any module or the driver in the STREAM can return).
13029		The STREAM head will convert the information pointed to by the strioc
13030		structure to an internal <i>ioctl()</i> command message and send it downstream.
13031		The <i>ioctl()</i> function with the I_STR command will fail if:
13032		[EAGAIN] or [ENOSR]
13033		Unable to allocate buffers for the <i>ioctl()</i> message.
13034		[EINVAL] The <i>ic_len</i> member is less than 0 or larger than the
13035		maximum configured size of the data part of a message, or
13036		<i>ic_timeout</i> is less than -1.
13037		[ENXIO] Hangup received on <i>fil</i> des.
13038		[ETIME] A downstream <i>ioctl()</i> timed out before acknowledgement
13039		was received.
13040		An I_STR can also fail while waiting for an acknowledgement if a message
13041		indicating an error or a hangup is received at the STREAM head. In addition,
13042		an error code can be returned in the positive or negative acknowledgement
13043		message, in the event the <i>ioctl()</i> command sent downstream fails. For these
13044		cases, I_STR fails with <i>errno</i> set to the value in the message.
13045	I_SWROPT	Sets the write mode using the value of the argument <i>arg</i> . Valid bit settings for
13046		<i>arg</i> are:
13047		SNDZERO Send a zero-length message downstream when a <i>write()</i> of
13048		0 bytes occurs. To not send a zero-length message when a
13049		<i>write()</i> of 0 bytes occurs, this bit must not be set in <i>arg</i> (for
13050		example, <i>arg</i> would be set to 0).
13051		The <i>ioctl()</i> function with the I_SWROPT command will fail if:
13052		[EINVAL] <i>arg</i> is not the above value.
13053	I_GWROPT	Returns the current write mode setting, as described above, in the int that is
13054		pointed to by the argument <i>arg</i> .
13055	I_SENDFD	I_SENDFD creates a new reference to the open file description associated with
13056		the file descriptor <i>arg</i> , and writes a message on the STREAMS-based pipe <i>fil</i> des
13057		containing this reference, together with the user ID and group ID of the calling
13058		process.
13059		The <i>ioctl()</i> function with the I_SENDFD command will fail if:
13060		[EAGAIN] The sending STREAM is unable to allocate a message block
13061		to contain the file pointer; or the read queue of the receiving
13062		STREAM head is full and cannot accept the message sent by
13063		I_SENDFD.

13064		[EBADF]	The <i>arg</i> argument is not a valid, open file descriptor.
13065		[EINVAL]	The <i>fildev</i> argument is not connected to a STREAM pipe.
13066		[ENXIO]	Hangup received on <i>fildev</i> .
13067	I_RECVFD		Retrieves the reference to an open file description from a message written to a STREAMS-based pipe using the I_SENDFD command, and allocates a new file descriptor in the calling process that refers to this open file description. The <i>arg</i> argument is a pointer to an strrecvfd data structure as defined in <stropts.h> .
13068			
13069			
13070			
13071			
13072			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.
13073			
13074			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 <i>errno</i> set to [EAGAIN] if no message is present at the STREAM head.
13075			
13076			
13077			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 <i>arg</i> .
13078			
13079			
13080			
13081			The <i>ioctl()</i> function with the I_RECVFD command will fail if:
13082		[EAGAIN]	A message is not present at the STREAM head read queue and the O_NONBLOCK flag is set.
13083			
13084		[EBADMSG]	The message at the STREAM head read queue is not a message containing a passed file descriptor.
13085			
13086		[EMFILE]	The process has the maximum number of file descriptors currently open that it is allowed.
13087			
13088		[ENXIO]	Hangup received on <i>fildev</i> .
13089	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 <i>arg</i> is a null pointer, the return value is the number of modules, including the driver, that are on the STREAM pointed to by <i>fildev</i> . This lets the process allocate enough space for the module names. Otherwise, it should point to an str_list structure.
13090			
13091			
13092			
13093			
13094			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 <i>ioctl()</i> 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.
13095			
13096			
13097			
13098			
13099			
13100			
13101			
13102			The <i>ioctl()</i> function with the I_LIST command will fail if:
13103		[EINVAL]	The sl_nmods member is less than 1.
13104		[EAGAIN] or [ENOSR]	
13105			Unable to allocate buffers.
13106	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 <i>arg</i>
13107			

13108		argument determines how the checking is done when there may be multiple
13109		marked messages on the STREAM head read queue. It may take on the
13110		following values:
13111		ANYMARK Check if the message is marked.
13112		LASTMARK Check if the message is the last one marked on the queue.
13113		The bitwise inclusive OR of the flags ANYMARK and LASTMARK is
13114		permitted.
13115		The return value is 1 if the mark condition is satisfied and 0 otherwise.
13116		The <i>ioctl()</i> function with the <i>I_ATMARK</i> command will fail if:
13117		[EINVAL] Invalid <i>arg</i> value.
13118	<i>I_CKBAND</i>	Check if the message of a given priority band exists on the STREAM head
13119		read queue. This returns 1 if a message of the given priority exists, 0 if no
13120		such message exists, or -1 on error. <i>arg</i> should be of type <i>int</i> .
13121		The <i>ioctl()</i> function with the <i>I_CKBAND</i> command will fail if:
13122		[EINVAL] Invalid <i>arg</i> value.
13123	<i>I_GETBAND</i>	Return the priority band of the first message on the STREAM head read queue
13124		in the integer referenced by <i>arg</i> .
13125		The <i>ioctl()</i> function with the <i>I_GETBAND</i> command will fail if:
13126		[ENODATA] No message on the STREAM head read queue.
13127	<i>I_CANPUT</i>	Check if a certain band is writable. <i>arg</i> is set to the priority band in question.
13128		The return value is 0 if the band is flow-controlled, 1 if the band is writable, or
13129		-1 on error.
13130		The <i>ioctl()</i> function with the <i>I_CANPUT</i> command will fail if:
13131		[EINVAL] Invalid <i>arg</i> value.
13132	<i>I_SETCLTIME</i>	This request allows the process to set the time the STREAM head will delay
13133		when a STREAM is closing and there is data on the write queues. Before
13134		closing each module or driver, if there is data on its write queue, the STREAM
13135		head will delay for the specified amount of time to allow the data to drain. If,
13136		after the delay, data is still present, they will be flushed. The <i>arg</i> argument is a
13137		pointer to an integer specifying the number of milliseconds to delay, rounded
13138		up to the nearest valid value. If <i>I_SETCLTIME</i> is not performed on a
13139		STREAM, an implementation-dependent default timeout interval is used.
13140		The <i>ioctl()</i> function with the <i>I_SETCLTIME</i> command will fail if:
13141		[EINVAL] Invalid <i>arg</i> value.
13142	<i>I_GETCLTIME</i>	This request returns the close time delay in the integer pointed to by <i>arg</i> .

13143 **Multiplexed STREAMS Configurations**

13144 The following four commands are used for connecting and disconnecting multiplexed
 13145 STREAMS configurations. These commands use an implementation-dependent default timeout
 13146 interval.

13147 **I_LINK** Connects two STREAMs, where *fildev* is the file descriptor of the STREAM
 13148 connected to the multiplexing driver, and *arg* is the file descriptor of the
 13149 STREAM connected to another driver. The STREAM designated by *arg* gets
 13150 connected below the multiplexing driver. I_LINK requires the multiplexing
 13151 driver to send an acknowledgement message to the STREAM head regarding
 13152 the connection. This call returns a multiplexer ID number (an identifier used
 13153 to disconnect the multiplexer; see I_UNLINK) on success, and -1 on failure.

13154 The *ioctl()* function with the I_LINK command will fail if:

13155 [ENXIO] Hangup received on *fildev*.
 13156 [ETIME] Time out before acknowledgement message was received at
 13157 STREAM head.
 13158 [EAGAIN] or [ENOSR]
 13159 Unable to allocate STREAMS storage to perform the
 13160 I_LINK.
 13161 [EBADF] The *arg* argument is not a valid, open file descriptor.
 13162 [EINVAL] The *fildev* argument does not support multiplexing; or *arg* is
 13163 not a STREAM or is already connected downstream from a
 13164 multiplexer; or the specified I_LINK operation would
 13165 connect the STREAM head in more than one place in the
 13166 multiplexed STREAM.

13167 An I_LINK can also fail while waiting for the multiplexing driver to
 13168 acknowledge the request, if a message indicating an error or a hangup is
 13169 received at the STREAM head of *fildev*. In addition, an error code can be
 13170 returned in the positive or negative acknowledgement message. For these
 13171 cases, I_LINK fails with *errno* set to the value in the message.

13172 **I_UNLINK** Disconnects the two STREAMs specified by *fildev* and *arg*. *fildev* is the file
 13173 descriptor of the STREAM connected to the multiplexing driver. The *arg*
 13174 argument is the multiplexer ID number that was returned by the I_LINK
 13175 *ioctl()* command when a STREAM was connected downstream from the
 13176 multiplexing driver. If *arg* is MUXID_ALL, then all STREAMs that were
 13177 connected to *fildev* are disconnected. As in I_LINK, this command requires
 13178 acknowledgement.

13179 The *ioctl()* function with the I_UNLINK command will fail if:

13180 [ENXIO] Hangup received on *fildev*.
 13181 [ETIME] Time out before acknowledgement message was received at
 13182 STREAM head.
 13183 [EAGAIN] or [ENOSR]
 13184 Unable to allocate buffers for the acknowledgement
 13185 message.
 13186 [EINVAL] Invalid multiplexer ID number.

13187		An I_UNLINK can also fail while waiting for the multiplexing driver to
13188		acknowledge the request if a message indicating an error or a hangup is
13189		received at the STREAM head of <i>filde</i> s. In addition, an error code can be
13190		returned in the positive or negative acknowledgement message. For these
13191		cases, I_UNLINK fails with <i>errno</i> set to the value in the message.
13192	I_PLINK	Creates a <i>persistent connection</i> between two STREAMs, where <i>filde</i> s is the file
13193		descriptor of the STREAM connected to the multiplexing driver, and <i>arg</i> is the
13194		file descriptor of the STREAM connected to another driver. This call creates a
13195		persistent connection which can exist even if the file descriptor <i>filde</i> s
13196		associated with the upper STREAM to the multiplexing driver is closed. The
13197		STREAM designated by <i>arg</i> gets connected via a persistent connection below
13198		the multiplexing driver. I_PLINK requires the multiplexing driver to send an
13199		acknowledgement message to the STREAM head. This call returns a
13200		multiplexer ID number (an identifier that may be used to disconnect the
13201		multiplexer, see I_PUNLINK) on success, and -1 on failure.
13202		The <i>ioctl</i> () function with the I_PLINK command will fail if:
13203	[ENXIO]	Hangup received on <i>filde</i> s.
13204	[ETIME]	Time out before acknowledgement message was received at
13205		STREAM head.
13206	[EAGAIN] or [ENOSR]	
13207		Unable to allocate STREAMS storage to perform the
13208		I_PLINK.
13209	[EBADF]	The <i>arg</i> argument is not a valid, open file descriptor.
13210	[EINVAL]	The <i>filde</i> s argument does not support multiplexing; or <i>arg</i> is
13211		not a STREAM or is already connected downstream from a
13212		multiplexer; or the specified I_PLINK operation would
13213		connect the STREAM head in more than one place in the
13214		multiplexed STREAM.
13215		An I_PLINK can also fail while waiting for the multiplexing driver to
13216		acknowledge the request, if a message indicating an error or a hangup is
13217		received at the STREAM head of <i>filde</i> s. In addition, an error code can be
13218		returned in the positive or negative acknowledgement message. For these
13219		cases, I_PLINK fails with <i>errno</i> set to the value in the message.
13220	I_PUNLINK	Disconnects the two STREAMs specified by <i>filde</i> s and <i>arg</i> from a persistent
13221		connection. The <i>filde</i> s argument is the file descriptor of the STREAM
13222		connected to the multiplexing driver. The <i>arg</i> argument is the multiplexer ID
13223		number that was returned by the I_PLINK <i>ioctl</i> () command when a STREAM
13224		was connected downstream from the multiplexing driver. If <i>arg</i> is
13225		MUXID_ALL then all STREAMs which are persistent connections to <i>filde</i> s are
13226		disconnected. As in I_PLINK, this command requires the multiplexing driver
13227		to acknowledge the request.
13228		The <i>ioctl</i> () function with the I_PUNLINK command will fail if:
13229	[ENXIO]	Hangup received on <i>filde</i> s.
13230	[ETIME]	Time out before acknowledgement message was received at
13231		STREAM head.

13232 [EAGAIN] or [ENOSR]
 13233 Unable to allocate buffers for the acknowledgement
 13234 message.
 13235 [EINVAL] Invalid multiplexer ID number.
 13236 An I_PUNLINK can also fail while waiting for the multiplexing driver to
 13237 acknowledge the request if a message indicating an error or a hangup is
 13238 received at the STREAM head of *fildev*. In addition, an error code can be
 13239 returned in the positive or negative acknowledgement message. For these
 13240 cases, I_PUNLINK fails with *errno* set to the value in the message.

13241 RETURN VALUE

13242 Upon successful completion, *ioctl()* returns a value other than -1 that depends upon the
 13243 STREAMS device control function. Otherwise, it returns -1 and sets *errno* to indicate the error.

13244 ERRORS

13245 Under the following general conditions, *ioctl()* will fail if:

13246 [EBADF] The *fildev* argument is not a valid open file descriptor.
 13247 [EINTR] A signal was caught during the *ioctl()* operation.
 13248 [EINVAL] The STREAM or multiplexer referenced by *fildev* is linked (directly or
 13249 indirectly) downstream from a multiplexer.

13250 If an underlying device driver detects an error, then *ioctl()* will fail if:

13251 [EINVAL] The *request* or *arg* argument is not valid for this device.
 13252 [EIO] Some physical I/O error has occurred.
 13253 [ENOTTY] The *fildev* argument is not associated with a STREAMS device that accepts
 13254 control functions.
 13255 [ENXIO] The *request* and *arg* arguments are valid for this device driver, but the service
 13256 requested cannot be performed on this particular sub-device.
 13257 [ENODEV] The *fildev* argument refers to a valid STREAMS device, but the corresponding
 13258 device driver does not support the *ioctl()* function.

13259 If a STREAM is connected downstream from a multiplexer, any *ioctl()* command except
 13260 I_UNLINK and I_PUNLINK will set *errno* to [EINVAL].

13261 EXAMPLES

13262 None.

13263 APPLICATION USAGE

13264 The implementation-dependent timeout interval for STREAMS has historically been 15 seconds.

13265 FUTURE DIRECTIONS

13266 None.

13267 SEE ALSO

13268 *close()*, *fcntl()*, *getmsg()*, *open()*, *pipe()*, *poll()*, *putmsg()*, *read()*, *sigaction()*, *write()*, <stropts.h>,
 13269 Section 2.5 on page 34.

13270 CHANGE HISTORY

13271 First released in Issue 4, Version 2.

13272 **Issue 5**

13273 Moved from X/OPEN UNIX extension to BASE.

|

13274 **NAME**

13275 isalnum — test for an alphanumeric character

13276 **SYNOPSIS**

13277 #include <ctype.h>

13278 int isalnum(int *c*);

13279 **DESCRIPTION**

13280 The *isalnum()* function tests whether *c* is a character of class **alpha** or **digit** in the program's
13281 current locale, see the **XBD** specification, **Chapter 5, Locale**.

13282 In all cases *c* is an **int**, the value of which must be representable as an **unsigned char** or must
13283 equal the value of the macro EOF. If the argument has any other value, the behaviour is
13284 undefined.

13285 **RETURN VALUE**

13286 The *isalnum()* function returns non-zero if *c* is an alphanumeric character; otherwise it returns 0.

13287 **ERRORS**

13288 No errors are defined.

13289 **EXAMPLES**

13290 None.

13291 **APPLICATION USAGE**

13292 To ensure application portability, especially across natural languages, only this function and
13293 those listed in the SEE ALSO section should be used for character classification.

13294 **FUTURE DIRECTIONS**

13295 None.

13296 **SEE ALSO**

13297 *isalpha()*, *isctrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*,
13298 *setlocale()*, <ctype.h>, <stdio.h>, the **XBD** specification, **Chapter 5, Locale**.

13299 **CHANGE HISTORY**

13300 First released in Issue 1.

13301 Derived from Issue 1 of the SVID.

13302 **Issue 4**

13303 The following change is incorporated in this issue:

- 13304 • The text of the DESCRIPTION and RETURN VALUE sections is revised, although there are
13305 no functional differences between this issue and Issue 3. Operation in the C locale is no
13306 longer described explicitly on this page.

13307 **NAME**

13308 isalpha — test for an alphabetic character

13309 **SYNOPSIS**

13310 #include <ctype.h>

13311 int isalpha(int c);

13312 **DESCRIPTION**

13313 The *isalpha()* function tests whether *c* is a character of class **alpha** in the program's current locale,
13314 see the **XBD** specification, **Chapter 5, Locale**.

13315 In all cases *c* is an **int**, the value of which must be representable as an **unsigned char** or must
13316 equal the value of the macro EOF. If the argument has any other value, the behaviour is
13317 undefined.

13318 **RETURN VALUE**

13319 The *isalpha()* function returns non-zero if *c* is an alphabetic character; otherwise it returns 0.

13320 **ERRORS**

13321 No errors are defined.

13322 **EXAMPLES**

13323 None.

13324 **APPLICATION USAGE**

13325 To ensure application portability, especially across natural languages, only this function and
13326 those listed in the SEE ALSO section should be used for character classification.

13327 **FUTURE DIRECTIONS**

13328 None.

13329 **SEE ALSO**

13330 *isalnum()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*,
13331 *isxdigit()*, *setlocale()*, <ctype.h>, <stdio.h>, the **XBD** specification, **Chapter 5, Locale**.

13332 **CHANGE HISTORY**

13333 First released in Issue 1.

13334 Derived from Issue 1 of the SVID.

13335 **Issue 4**

13336 The following change is incorporated in this issue:

- 13337 • The text of the DESCRIPTION and RETURN VALUE sections is revised, although there are
13338 no functional differences between this issue and Issue 3. Operation in the C locale is no
13339 longer described explicitly on this page.

13340 **NAME**

13341 isascii — test for a 7-bit US-ASCII character

13342 **SYNOPSIS**

13343 EX #include <ctype.h>

13344 int isascii(int c);

13345

13346 **DESCRIPTION**13347 The *isascii()* function tests whether *c* is a 7-bit US-ASCII character code.13348 The *isascii()* function is defined on all integer values.13349 **RETURN VALUE**13350 The *isascii()* function returns non-zero if *c* is a 7-bit US-ASCII character code between 0 and octal

13351 0177 inclusive; otherwise it returns 0.

13352 **ERRORS**

13353 No errors are defined.

13354 **EXAMPLES**

13355 None.

13356 **APPLICATION USAGE**

13357 None.

13358 **FUTURE DIRECTIONS**

13359 None.

13360 **SEE ALSO**

13361 <ctype.h>.

13362 **CHANGE HISTORY**

13363 First released in Issue 1.

13364 Derived from Issue 1 of the SVID.

13365 **NAME**

13366 isastream — test a file descriptor

13367 **SYNOPSIS**13368 EX `#include <stropts.h>`13369 `int isastream(int fildes);`

13370

13371 **DESCRIPTION**

13372 The *isastream()* function tests whether *fildes*, an open file descriptor, is associated with a
13373 STREAMS-based file.

13374 **RETURN VALUE**

13375 Upon successful completion, *isastream()* returns 1 if *fildes* refers to a STREAMS-based file and 0 if
13376 not. Otherwise, *isastream()* returns -1 and sets *errno* to indicate the error.

13377 **ERRORS**13378 The *isastream()* function will fail if:

13379 [EBADF] The *fildes* argument is not a valid open file descriptor.

13380 **EXAMPLES**

13381 None.

13382 **APPLICATION USAGE**

13383 None.

13384 **FUTURE DIRECTIONS**

13385 None.

13386 **SEE ALSO**13387 `<stropts.h>`.13388 **CHANGE HISTORY**

13389 First released in Issue 4, Version 2.

13390 **Issue 5**

13391 Moved from X/OPEN UNIX extension to BASE.

13392 **NAME**

13393 isatty — test for a terminal device

13394 **SYNOPSIS**

13395 #include <unistd.h>

13396 int isatty(int *fildes*);13397 **DESCRIPTION**13398 The *isatty()* function tests whether *fildes*, an open file descriptor, is associated with a terminal device.
1339913400 **RETURN VALUE**13401 The *isatty()* function returns 1 if *fildes* is associated with a terminal; otherwise it returns 0 and may set *errno* to indicate the error.
1340213403 **ERRORS**13404 The *isatty()* function may fail if:13405 EX [EBADF] The *fildes* argument is not a valid open file descriptor.13406 [ENOTTY] The *fildes* argument is not associated with a terminal.13407 **EXAMPLES**

13408 None.

13409 **APPLICATION USAGE**13410 The *isatty()* function does not necessarily indicate that a human being is available for interaction via *fildes*. It is quite possible that non-terminal devices are connected to the communications line.
1341213413 **FUTURE DIRECTIONS**

13414 None.

13415 **SEE ALSO**

13416 <unistd.h>.

13417 **CHANGE HISTORY**

13418 First released in Issue 1.

13419 Derived from Issue 1 of the SVID.

13420 **Issue 4**

13421 The following changes are incorporated in this issue:

- 13422 • The header <unistd.h> is added to the SYNOPSIS section.
- 13423 • In the RETURN VALUE section, the sentence indicating that this function may set *errno* is marked as an extension.
- 13424 • The errors [EBADF] and [ENOTTY] are marked as extensions.
- 13425

13426 **NAME**

13427 isctrl — test for a control character

13428 **SYNOPSIS**

13429 #include <ctype.h>

13430 int isctrl(int c);

13431 **DESCRIPTION**

13432 The *isctrl()* function tests whether *c* is a character of class **cntrl** in the program's current locale,
13433 see the **XBD** specification, **Chapter 5, Locale**.

13434 In all cases *c* is a type **int**, the value of which must be a character representable as an **unsigned**
13435 **char** or must equal the value of the macro **EOF**. If the argument has any other value, the
13436 behaviour is undefined.

13437 **RETURN VALUE**13438 The *isctrl()* function returns non-zero if *c* is a control character; otherwise it returns 0.13439 **ERRORS**

13440 No errors are defined.

13441 **EXAMPLES**

13442 None.

13443 **APPLICATION USAGE**

13444 To ensure applications portability, especially across natural languages, only this function and
13445 those listed in the SEE ALSO section should be used for character classification.

13446 **FUTURE DIRECTIONS**

13447 None.

13448 **SEE ALSO**

13449 *isalnum()*, *isalpha()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*,
13450 *isxdigit()*, *setlocale()*, <ctype.h>, the **XBD** specification, **Chapter 5, Locale**.

13451 **CHANGE HISTORY**

13452 First released in Issue 1.

13453 Derived from Issue 1 of the SVID.

13454 **Issue 4**

13455 The following change is incorporated in this issue:

- 13456 • The text of the **DESCRIPTION** and **RETURN VALUE** sections is revised, although there are
13457 no functional differences between this issue and Issue 3. Operation in the C locale is no
13458 longer described explicitly on this page.

13459 **NAME**

13460 isdigit — test for a decimal digit

13461 **SYNOPSIS**

13462 #include <ctype.h>

13463 int isdigit(int c);

13464 **DESCRIPTION**13465 The *isdigit()* function tests whether *c* is a character of class **digit** in the program's current locale, see the **XBD** specification, **Chapter 5, Locale**.13467 In all cases *c* is an **int**, the value of which must be a character representable as an **unsigned char** or must equal the value of the macro EOF. If the argument has any other value, the behaviour is undefined.13470 **RETURN VALUE**13471 The *isdigit()* function returns non-zero if *c* is a decimal digit; otherwise it returns 0.13472 **ERRORS**

13473 No errors are defined.

13474 **EXAMPLES**

13475 None.

13476 **APPLICATION USAGE**

13477 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.

13479 **FUTURE DIRECTIONS**

13480 None.

13481 **SEE ALSO**13482 *isalnum()*, *isalpha()*, *iscntrl()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*, <ctype.h>.13484 **CHANGE HISTORY**

13485 First released in Issue 1.

13486 Derived from Issue 1 of the SVID.

13487 **Issue 4**

13488 The following change is incorporated in this issue:

- 13489 • The text of the DESCRIPTION is revised, although there are no functional differences
- 13490 between this issue and Issue 3.

13491 **NAME**

13492 isgraph — test for a visible character

13493 **SYNOPSIS**

13494 #include <ctype.h>

13495 int isgraph(int c);

13496 **DESCRIPTION**

13497 The *isgraph()* function tests whether *c* is a character of class **graph** in the program's current
13498 locale, see the **XBD** specification, **Chapter 5, Locale**.

13499 In all cases *c* is an **int**, the value of which must be a character representable as an **unsigned char**
13500 or must equal the value of the macro **EOF**. If the argument has any other value, the behaviour is
13501 undefined.

13502 **RETURN VALUE**

13503 The *isgraph()* function returns non-zero if *c* is a character with a visible representation; otherwise
13504 it returns 0.

13505 **ERRORS**

13506 No errors are defined.

13507 **EXAMPLES**

13508 None.

13509 **APPLICATION USAGE**

13510 To ensure applications portability, especially across natural languages, only this function and
13511 those listed in the SEE ALSO section should be used for character classification.

13512 **FUTURE DIRECTIONS**

13513 None.

13514 **SEE ALSO**

13515 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*,
13516 *setlocale()*, <ctype.h>, the **XBD** specification, **Chapter 5, Locale**.

13517 **CHANGE HISTORY**

13518 First released in Issue 1.

13519 Derived from Issue 1 of the SVID.

13520 **Issue 4**

13521 The following change is incorporated in this issue:

- 13522 • The text of the **DESCRIPTION** and **RETURN VALUE** sections is revised, although there are
13523 no functional differences between this issue and Issue 3. Operation in the C locale is no
13524 longer described explicitly on this page.

13525 **NAME**

13526 islower — test for a lower-case letter

13527 **SYNOPSIS**

13528 #include <ctype.h>

13529 int islower(int c);

13530 **DESCRIPTION**

13531 The *islower()* function tests whether *c* is a character of class **lower** in the program's current
13532 locale, see the **XBD** specification, **Chapter 5, Locale**.

13533 In all cases *c* is an **int**, the value of which must be a character representable as an **unsigned char**
13534 or must equal the value of the macro EOF. If the argument has any other value, the behaviour is
13535 undefined.

13536 **RETURN VALUE**

13537 The *islower()* function returns non-zero if *c* is a lower-case letter; otherwise it returns 0.

13538 **ERRORS**

13539 No errors are defined.

13540 **EXAMPLES**

13541 None.

13542 **APPLICATION USAGE**

13543 To ensure applications portability, especially across natural languages, only this function and
13544 those listed in the SEE ALSO section should be used for character classification.

13545 **FUTURE DIRECTIONS**

13546 None.

13547 **SEE ALSO**

13548 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*,
13549 *isxdigit()*, *setlocale()*, <ctype.h>, the **XBD** specification, **Chapter 5, Locale**.

13550 **CHANGE HISTORY**

13551 First released in Issue 1.

13552 Derived from Issue 1 of the SVID.

13553 **Issue 4**

13554 The following change is incorporated in this issue:

- 13555 • The text of the DESCRIPTION and RETURN VALUE sections is revised, although there are
13556 no functional differences between this issue and Issue 3. Operation in the C locale is no
13557 longer described explicitly on this page.

13558 **NAME**

13559 isnan — test for a NaN

13560 **SYNOPSIS**

13561 EX #include <math.h>

13562 int isnan(double x);

13563

13564 **DESCRIPTION**13565 The *isnan()* function tests whether *x* is NaN.13566 On systems not supporting NaN values, *isnan()* always returns 0.13567 **RETURN VALUE**13568 The *isnan()* function returns non-zero if *x* is NaN. Otherwise, 0 is returned.13569 **ERRORS**

13570 No errors are defined.

13571 **EXAMPLES**

13572 None.

13573 **APPLICATION USAGE**

13574 None.

13575 **FUTURE DIRECTIONS**

13576 None.

13577 **SEE ALSO**

13578 <math.h>.

13579 **CHANGE HISTORY**

13580 First released in Issue 3.

13581 **Issue 4**

13582 The following change is incorporated in this issue:

- 13583
- The words “not supporting NaN” are added to the APPLICATION USAGE section.

13584 **Issue 5**13585 The DESCRIPTION is updated to indicate the return value when NaN is not supported. This
13586 text was previously published in the APPLICATION USAGE section.

13587 **NAME**

13588 isprint — test for a printing character

13589 **SYNOPSIS**

13590 #include <ctype.h>

13591 int isprint(int c);

13592 **DESCRIPTION**13593 The *isprint()* function tests whether *c* is a character of class **print** in the program's current locale,
13594 see the **XBD** specification, **Chapter 5, Locale**.13595 In all cases *c* is an **int**, the value of which must be a character representable as an **unsigned char**
13596 or must equal the value of the macro EOF. If the argument has any other value, the behaviour is
13597 undefined.13598 **RETURN VALUE**13599 The *isprint()* function returns non-zero if *c* is a printing character; otherwise it returns 0.13600 **ERRORS**

13601 No errors are defined.

13602 **EXAMPLES**

13603 None.

13604 **APPLICATION USAGE**13605 To ensure applications portability, especially across natural languages, only this function and
13606 those listed in the SEE ALSO section should be used for character classification.13607 **FUTURE DIRECTIONS**

13608 None.

13609 **SEE ALSO**13610 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *ispunct()*, *isspace()*, *isupper()*,
13611 *isxdigit()*, *setlocale()*, <ctype.h>, the **XBD** specification, **Chapter 5, Locale**.13612 **CHANGE HISTORY**

13613 First released in Issue 1.

13614 Derived from Issue 1 of the SVID.

13615 **Issue 4**

13616 The following change is incorporated in this issue:

- 13617
- The text of the DESCRIPTION and RETURN VALUE sections is revised, although there are
13618 no functional differences between this issue and Issue 3. Operation in the C locale is no
13619 longer described explicitly on this page.

13620 **NAME**

13621 ispunct — test for a punctuation character

13622 **SYNOPSIS**

13623 #include <ctype.h>

13624 int ispunct(int *c*);13625 **DESCRIPTION**13626 The *ispunct()* function tests whether *c* is a character of class **punct** in the program's current
13627 locale, see the **XBD** specification, **Chapter 5, Locale**.13628 In all cases *c* is an **int**, the value of which must be a character representable as an **unsigned char**
13629 or must equal the value of the macro EOF. If the argument has any other value, the behaviour is
13630 undefined.13631 **RETURN VALUE**13632 The *ispunct()* function returns non-zero if *c* is a punctuation character; otherwise it returns 0.13633 **ERRORS**

13634 No errors are defined.

13635 **EXAMPLES**

13636 None.

13637 **APPLICATION USAGE**13638 To ensure applications portability, especially across natural languages, only this function and
13639 those listed in the SEE ALSO section should be used for character classification.13640 **FUTURE DIRECTIONS**

13641 None.

13642 **SEE ALSO**13643 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *isspace()*, *isupper()*, *isxdigit()*,
13644 *setlocale()*, <ctype.h>, the **XBD** specification, **Chapter 5, Locale**.13645 **CHANGE HISTORY**

13646 First released in Issue 1.

13647 Derived from Issue 1 of the SVID.

13648 **Issue 4**

13649 The following change is incorporated in this issue:

- 13650
- The text of the DESCRIPTION and RETURN VALUE sections is revised, although there are
13651 no functional differences between this issue and Issue 3. Operation in the C locale is no
13652 longer described explicitly on this page.

13653 **NAME**

13654 isspace — test for a white-space character

13655 **SYNOPSIS**

13656 #include <ctype.h>

13657 int isspace(int c);

13658 **DESCRIPTION**13659 The *isspace()* function tests whether *c* is a character of class **space** in the program's current locale, see the **XBD** specification, **Chapter 5, Locale**.13661 In all cases *c* is an **int**, the value of which must be a character representable as an **unsigned char** or must equal the value of the macro EOF. If the argument has any other value, the behaviour is undefined.13664 **RETURN VALUE**13665 The *isspace()* function returns non-zero if *c* is a white-space character; otherwise it returns 0.13666 **ERRORS**

13667 No errors are defined.

13668 **EXAMPLES**

13669 None.

13670 **APPLICATION USAGE**

13671 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.

13673 **FUTURE DIRECTIONS**

13674 None.

13675 **SEE ALSO**13676 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isupper()*, *isxdigit()*, *setlocale()*, <ctype.h>, the **XBD** specification, **Chapter 5, Locale**.13678 **CHANGE HISTORY**

13679 First released in Issue 1.

13680 Derived from Issue 1 of the SVID.

13681 **Issue 4**

13682 The following change is incorporated in this issue:

- 13683 • The text of the DESCRIPTION and RETURN VALUE sections is revised, although there are
- 13684 no functional differences between this issue and Issue 3. Operation in the C locale is no
- 13685 longer described explicitly on this page.

13686 **NAME**

13687 isupper — test for an upper-case letter

13688 **SYNOPSIS**

13689 #include <ctype.h>

13690 int isupper(int c);

13691 **DESCRIPTION**

13692 The *isupper()* function tests whether *c* is a character of class **upper** in the program's current
13693 locale, see the **XBD** specification, **Chapter 5, Locale**.

13694 In all cases *c* is an **int**, the value of which must be a character representable as an **unsigned char**
13695 or must equal the value of the macro **EOF**. If the argument has any other value, the behaviour is
13696 undefined.

13697 **RETURN VALUE**

13698 The *isupper()* function returns non-zero if *c* is an upper-case letter; otherwise it returns 0.

13699 **ERRORS**

13700 No errors are defined.

13701 **EXAMPLES**

13702 None.

13703 **APPLICATION USAGE**

13704 To ensure applications portability, especially across natural languages, only this function and
13705 those listed in the SEE ALSO section should be used for character classification.

13706 **FUTURE DIRECTIONS**

13707 None.

13708 **SEE ALSO**

13709 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isxdigit()*,
13710 *setlocale()*, <ctype.h>, the **XBD** specification, **Chapter 5, Locale**.

13711 **CHANGE HISTORY**

13712 First released in Issue 1.

13713 Derived from Issue 1 of the SVID.

13714 **Issue 4**

13715 The following change is incorporated in this issue:

- 13716 • The text of the **DESCRIPTION** and **RETURN VALUE** sections is revised, although there are
13717 no functional differences between this issue and Issue 3. Operation in the C locale is no
13718 longer described explicitly on this page.

13719 **NAME**

13720 iswalnum — test for an alphanumeric wide-character code

13721 **SYNOPSIS**

13722 #include <wctype.h>

13723 int iswalnum(wint_t wc);

13724 **DESCRIPTION**13725 The *iswalnum()* function tests whether *wc* is a wide-character code representing a character of
13726 class **alpha** or **digit** in the program's current locale, see the **XBD** specification, **Chapter 5, Locale**.13727 In all cases *wc* is a **wint_t**, the value of which must be a wide-character code corresponding to a
13728 valid character in the current locale or must equal the value of the macro **WEOF**. If the
13729 argument has any other value, the behaviour is undefined.13730 **RETURN VALUE**13731 The *iswalnum()* function returns non-zero if *wc* is an alphanumeric wide-character code;
13732 otherwise it returns 0.13733 **ERRORS**

13734 No errors are defined.

13735 **EXAMPLES**

13736 None.

13737 **APPLICATION USAGE**13738 To ensure applications portability, especially across natural languages, only this function and
13739 those listed in the **SEE ALSO** section should be used for classification of wide-character codes.13740 **FUTURE DIRECTIONS**

13741 None.

13742 **SEE ALSO**13743 *iswalpunct()*, *iswcntrl()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*, *iswspace()*,
13744 *iswupper()*, *iswxdigit()*, *setlocale()*, <wctype.h>, <wchar.h>, <stdio.h>, the **XBD** specification,
13745 **Chapter 5, Locale**.13746 **CHANGE HISTORY**

13747 First released as a World-wide Portability Interface in Issue 4.

13748 **Issue 5**13749 The following change has been made in this issue for alignment with ISO/IEC
13750 9899:1990/Amendment 1:1994 (E).

- 13751
- The **SYNOPSIS** has been changed to indicate that this function and associated data types are
13752 now made visible by inclusion of the header <wctype.h> rather than <wchar.h>.

13753 **NAME**

13754 iswalpha — test for an alphabetic wide-character code

13755 **SYNOPSIS**

13756 #include <wctype.h>

13757 int iswalpha(wint_t wc);

13758 **DESCRIPTION**

13759 The *iswalpha()* function tests whether *wc* is a wide-character code representing a character of
13760 class **alpha** in the program's current locale, see the **XBD** specification, **Chapter 5, Locale**.

13761 In all cases *wc* is a **wint_t**, the value of which must be a wide-character code corresponding to a
13762 valid character in the current locale or must equal the value of the macro **WEOF**. If the
13763 argument has any other value, the behaviour is undefined.

13764 **RETURN VALUE**

13765 The *iswalpha()* function returns non-zero if *wc* is an alphabetic wide-character code; otherwise it
13766 returns 0.

13767 **ERRORS**

13768 No errors are defined.

13769 **EXAMPLES**

13770 None.

13771 **APPLICATION USAGE**

13772 To ensure applications portability, especially across natural languages, only this function and
13773 those listed in the SEE ALSO section should be used for classification of wide-character codes.

13774 **FUTURE DIRECTIONS**

13775 None.

13776 **SEE ALSO**

13777 *iswalnum()*, *iswcntrl()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*, *iswspace()*,
13778 *iswupper()*, *iswxdigit()*, *setlocale()*, <wctype.h>, <wchar.h>, <stdio.h>, the **XBD** specification,
13779 **Chapter 5, Locale**.

13780 **CHANGE HISTORY**

13781 First released in Issue 4.

13782 **Issue 5**

13783 The following change has been made in this issue for alignment with ISO/IEC
13784 9899:1990/Amendment 1:1994 (E).

- 13785 • The SYNOPSIS has been changed to indicate that this function and associated data types are
13786 now made visible by inclusion of the header <wctype.h> rather than <wchar.h>.

13787 **NAME**

13788 iswcntrl — test for a control wide-character code

13789 **SYNOPSIS**

13790 #include <wctype.h>

13791 int iswcntrl(wint_t wc);

13792 **DESCRIPTION**13793 The *iswcntrl()* function tests whether *wc* is a wide-character code representing a character of
13794 class **control** in the program's current locale, see the **XBD** specification, **Chapter 5, Locale**.13795 In all cases *wc* is a **wint_t**, the value of which must be a wide-character code corresponding to a
13796 valid character in the current locale or must equal the value of the macro WEOF. If the
13797 argument has any other value, the behaviour is undefined.13798 **RETURN VALUE**13799 The *iswcntrl()* function returns non-zero if *wc* is a control wide-character code; otherwise it
13800 returns 0.13801 **ERRORS**

13802 No errors are defined.

13803 **EXAMPLES**

13804 None.

13805 **APPLICATION USAGE**13806 To ensure applications portability, especially across natural languages, only this function and
13807 those listed in the SEE ALSO section should be used for classification of wide-character codes.13808 **FUTURE DIRECTIONS**

13809 None.

13810 **SEE ALSO**13811 *iswalnum()*, *iswalpha()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*, *iswspace()*,
13812 *iswupper()*, *iswxdigit()*, *setlocale()*, <wctype.h>, <wchar.h>, the **XBD** specification, **Chapter 5,**
13813 **Locale**.13814 **CHANGE HISTORY**

13815 First released in Issue 4.

13816 **Issue 5**13817 The following change has been made in this issue for alignment with ISO/IEC
13818 9899:1990/Amendment 1:1994 (E).

- 13819
- The SYNOPSIS has been changed to indicate that this function and associated data types are
13820 now made visible by inclusion of the header <wctype.h> rather than <wchar.h>.

13821 **NAME**

13822 iswctype — test character for a specified class

13823 **SYNOPSIS**

13824 #include <wctype.h>

13825 int iswctype(wint_t *wc*, wctype_t *charclass*);13826 **DESCRIPTION**

13827 The *iswctype()* function determines whether the wide-character code *wc* has the character class
 13828 *charclass*, returning true or false. The *iswctype()* function is defined on WEOF and wide-
 13829 character codes corresponding to the valid character encodings in the current locale. If the *wc*
 13830 argument is not in the domain of the function, the result is undefined. If the value of *charclass* is
 13831 invalid (that is, not obtained by a call to *wctype()* or *charclass* is invalidated by a subsequent call
 13832 to *setlocale()* that has affected category LC_CTYPE) the result is implementation-dependent.

13833 **RETURN VALUE**13834 The *iswctype()* function returns 0 for false and non-zero for true.13835 **ERRORS**

13836 No errors are defined.

13837 **EXAMPLES**

13838 None.

13839 **APPLICATION USAGE**

13840 The twelve strings — "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct",
 13841 "space", "upper" and "xdigit" — are reserved for the standard character classes. In the table
 13842 below, the functions in the left column are equivalent to the functions in the right column.

13843	iswalnum(<i>wc</i>)	iswctype(<i>wc</i> , wctype("alnum"))
13844	iswalpha(<i>wc</i>)	iswctype(<i>wc</i> , wctype("alpha"))
13845	iswcntrl(<i>wc</i>)	iswctype(<i>wc</i> , wctype("cntrl"))
13846	iswdigit(<i>wc</i>)	iswctype(<i>wc</i> , wctype("digit"))
13847	iswgraph(<i>wc</i>)	iswctype(<i>wc</i> , wctype("graph"))
13848	iswlower(<i>wc</i>)	iswctype(<i>wc</i> , wctype("lower"))
13849	iswprint(<i>wc</i>)	iswctype(<i>wc</i> , wctype("print"))
13850	iswpunct(<i>wc</i>)	iswctype(<i>wc</i> , wctype("punct"))
13851	iswspace(<i>wc</i>)	iswctype(<i>wc</i> , wctype("space"))
13852	iswupper(<i>wc</i>)	iswctype(<i>wc</i> , wctype("upper"))
13853	iswxdigit(<i>wc</i>)	iswctype(<i>wc</i> , wctype("xdigit"))

13854 **Note:** The call:13855 iswctype(*wc*, wctype("blank"))13856 does not have an equivalent *isw*()* function.13857 **FUTURE DIRECTIONS**

13858 None.

13859 **SEE ALSO**

13860 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*,
 13861 *iswspace()*, *iswupper()*, *iswxdigit()*, *wctype()*, <wctype.h>, <wchar.h>.

13862 **CHANGE HISTORY**

13863 First released as World-wide Portability Interfaces in Issue 4.

13864 **Issue 5**

13865 The following change has been made in this issue for alignment with ISO/IEC
13866 9899:1990/Amendment 1:1994 (E).

- 13867 • The SYNOPSIS has been changed to indicate that this function and associated data types are
13868 now made visible by inclusion of the header `<wctype.h>` rather than `<wchar.h>`.

13869 **NAME**

13870 iswdigit — test for a decimal digit wide-character code

13871 **SYNOPSIS**

13872 #include <wctype.h>

13873 int iswdigit(wint_t wc);

13874 **DESCRIPTION**13875 The *iswdigit()* function tests whether *wc* is a wide-character code representing a character of
13876 class **digit** in the program's current locale, see the **XBD** specification, **Chapter 5, Locale**.13877 In all cases *wc* is a **wint_t**, the value of which must be a wide-character code corresponding to a
13878 valid character in the current locale or must equal the value of the macro WEOF. If the
13879 argument has any other value, the behaviour is undefined.13880 **RETURN VALUE**13881 The *iswdigit()* function returns non-zero if *wc* is a decimal digit wide-character code; otherwise it
13882 returns 0.13883 **ERRORS**

13884 No errors are defined.

13885 **EXAMPLES**

13886 None.

13887 **APPLICATION USAGE**13888 To ensure applications portability, especially across natural languages, only this function and
13889 those listed in the SEE ALSO section should be used for classification of wide-character codes.13890 **FUTURE DIRECTIONS**

13891 None.

13892 **SEE ALSO**13893 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*, *iswspace()*,
13894 *iswupper()*, *iswxdigit()*, <wctype.h>, <wchar.h>.13895 **CHANGE HISTORY**

13896 First released in Issue 4.

13897 **Issue 5**13898 The following change has been made in this issue for alignment with ISO/IEC
13899 9899:1990/Amendment 1:1994 (E).

- 13900
- The SYNOPSIS has been changed to indicate that this function and associated data types are
 - 13901 now made visible by inclusion of the header <wctype.h> rather than <wchar.h>.

13902 **NAME**

13903 iswgraph — test for a visible wide-character code

13904 **SYNOPSIS**

13905 #include <wctype.h>

13906 int iswgraph(wint_t wc);

13907 **DESCRIPTION**13908 The *iswgraph()* function tests whether *wc* is a wide-character code representing a character of
13909 class **graph** in the program's current locale, see the **XBD** specification, **Chapter 5, Locale**.13910 In all cases *wc* is a **wint_t**, the value of which must be a wide-character code corresponding to a
13911 valid character in the current locale or must equal the value of the macro WEOF. If the
13912 argument has any other value, the behaviour is undefined.13913 **RETURN VALUE**13914 The *iswgraph()* function returns non-zero if *wc* is a wide-character code with a visible
13915 representation; otherwise it returns 0.13916 **ERRORS**

13917 No errors are defined.

13918 **EXAMPLES**

13919 None.

13920 **APPLICATION USAGE**13921 To ensure applications portability, especially across natural languages, only this function and
13922 those listed in the SEE ALSO section should be used for classification of wide-character codes.13923 **FUTURE DIRECTIONS**

13924 None.

13925 **SEE ALSO**13926 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswdigit()*, *iswlower()*, *iswprint()*, *iswpunct()*, *iswspace()*,
13927 *iswupper()*, *iswxdigit()*, *setlocale()*, <**wctype.h**>, <**wchar.h**>, the **XBD** specification, **Chapter 5,**
13928 **Locale**.13929 **CHANGE HISTORY**

13930 First released in Issue 4.

13931 **Issue 5**13932 The following change has been made in this issue for alignment with ISO/IEC
13933 9899:1990/Amendment 1:1994 (E).

- 13934 • The SYNOPSIS has been changed to indicate that this function and associated data types are
13935 now made visible by inclusion of the header <**wctype.h**> rather than <**wchar.h**>.

13936 **NAME**

13937 iswlower — test for a lower-case letter wide-character code

13938 **SYNOPSIS**

13939 #include <wctype.h>

13940 int iswlower(wint_t wc);

13941 **DESCRIPTION**

13942 The *iswlower()* function tests whether *wc* is a wide-character code representing a character of
13943 class **lower** in the program's current locale, see the **XBD** specification, **Chapter 5, Locale**.

13944 In all cases *wc* is a **wint_t**, the value of which must be a wide-character code corresponding to a
13945 valid character in the current locale or must equal the value of the macro WEOF. If the
13946 argument has any other value, the behaviour is undefined.

13947 **RETURN VALUE**

13948 The *iswlower()* function returns non-zero if *wc* is a lower-case letter wide-character code;
13949 otherwise it returns 0.

13950 **ERRORS**

13951 No errors are defined.

13952 **EXAMPLES**

13953 None.

13954 **APPLICATION USAGE**

13955 To ensure applications portability, especially across natural languages, only this function and
13956 those listed in the SEE ALSO section should be used for classification of wide-character codes.

13957 **FUTURE DIRECTIONS**

13958 None.

13959 **SEE ALSO**

13960 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswdigit()*, *iswgraph()*, *iswprint()*, *iswpunct()*, *iswspace()*,
13961 *iswupper()*, *iswxdigit()*, *setlocale()*, <wctype.h>, <wchar.h>, the **XBD** specification, **Chapter 5,**
13962 **Locale**.

13963 **CHANGE HISTORY**

13964 First released in Issue 4.

13965 **Issue 5**

13966 The following change has been made in this issue for alignment with ISO/IEC
13967 9899:1990/Amendment 1:1994 (E).

- 13968 • The SYNOPSIS has been changed to indicate that this function and associated data types are
13969 now made visible by inclusion of the header <wctype.h> rather than <wchar.h>.

13970 **NAME**

13971 iswprint — test for a printing wide-character code

13972 **SYNOPSIS**

13973 #include <wctype.h>

13974 int iswprint(wint_t wc);

13975 **DESCRIPTION**13976 The *iswprint()* function tests whether *wc* is a wide-character code representing a character of
13977 class **print** in the program's current locale, see the **XBD** specification, **Chapter 5, Locale**.13978 In all cases *wc* is a **wint_t**, the value of which must be a wide-character code corresponding to a
13979 valid character in the current locale or must equal the value of the macro WEOF. If the
13980 argument has any other value, the behaviour is undefined.13981 **RETURN VALUE**13982 The *iswprint()* function returns non-zero if *wc* is a printing wide-character code; otherwise it
13983 returns 0.13984 **ERRORS**

13985 No errors are defined.

13986 **EXAMPLES**

13987 None.

13988 **APPLICATION USAGE**13989 To ensure applications portability, especially across natural languages, only this function and
13990 those listed in the SEE ALSO section should be used for classification of wide-character codes.13991 **FUTURE DIRECTIONS**

13992 None.

13993 **SEE ALSO**13994 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswpunct()*, *iswspace()*,
13995 *iswupper()*, *iswxdigit()*, *setlocale()*, <**wctype.h**>, <**wchar.h**>, the **XBD** specification, **Chapter 5,**
13996 **Locale**.13997 **CHANGE HISTORY**

13998 First released in Issue 4.

13999 **Issue 5**14000 The following change has been made in this issue for alignment with ISO/IEC
14001 9899:1990/Amendment 1:1994 (E).

- 14002 • The SYNOPSIS has been changed to indicate that this function and associated data types are
- 14003 now made visible by inclusion of the header <**wctype.h**> rather than <**wchar.h**>.

14004 **NAME**

14005 iswpunct — test for a punctuation wide-character code

14006 **SYNOPSIS**

14007 #include <wctype.h>

14008 int iswpunct(wint_t wc);

14009 **DESCRIPTION**14010 The *iswpunct()* function tests whether *wc* is a wide-character code representing a character of
14011 class **punct** in the program's current locale, see the **XBD** specification, **Chapter 5, Locale**.14012 In all cases *wc* is a **wint_t**, the value of which must be a wide-character code corresponding to a
14013 valid character in the current locale or must equal the value of the macro WEOF. If the
14014 argument has any other value, the behaviour is undefined.14015 **RETURN VALUE**14016 The *iswpunct()* function returns non-zero if *wc* is a punctuation wide-character code; otherwise it
14017 returns 0.14018 **ERRORS**

14019 No errors are defined.

14020 **EXAMPLES**

14021 None.

14022 **APPLICATION USAGE**14023 To ensure applications portability, especially across natural languages, only this function and
14024 those listed in the SEE ALSO section should be used for classification of wide-character codes.14025 **FUTURE DIRECTIONS**

14026 None.

14027 **SEE ALSO**14028 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswspace()*,
14029 *iswupper()*, *iswxdigit()*, *setlocale()*, <wctype.h>, <wchar.h>, the **XBD** specification, **Chapter 5,**
14030 **Locale**.14031 **CHANGE HISTORY**

14032 First released in Issue 4.

14033 **Issue 5**14034 The following change has been made in this issue for alignment with ISO/IEC
14035 9899:1990/Amendment 1:1994 (E).

- 14036
- The SYNOPSIS has been changed to indicate that this function and associated data types are
 - 14037 now made visible by inclusion of the header <wctype.h> rather than <wchar.h>.

14038 **NAME**

14039 iswspace — test for a white-space wide-character code

14040 **SYNOPSIS**

14041 #include <wctype.h>

14042 int iswspace(wint_t *wc*);14043 **DESCRIPTION**14044 The *iswspace()* function tests whether *wc* is a wide-character code representing a character of
14045 class **space** in the program's current locale, see the **XBD** specification, **Chapter 5, Locale**.14046 In all cases *wc* is a **wint_t**, the value of which must be a wide-character code corresponding to a
14047 valid character in the current locale or must equal the value of the macro WEOF. If the
14048 argument has any other value, the behaviour is undefined.14049 **RETURN VALUE**14050 The *iswspace()* function returns non-zero if *wc* is a white-space wide-character code; otherwise it
14051 returns 0.14052 **ERRORS**

14053 No errors are defined.

14054 **EXAMPLES**

14055 None.

14056 **APPLICATION USAGE**14057 To ensure applications portability, especially across natural languages, only this function and
14058 those listed in the SEE ALSO section should be used for classification of wide-character codes.14059 **FUTURE DIRECTIONS**

14060 None.

14061 **SEE ALSO**14062 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*,
14063 *iswupper()*, *iswxdigit()*, *setlocale()*, <wctype.h>, <wchar.h>, the **XBD** specification, **Chapter 5,**
14064 **Locale**.14065 **CHANGE HISTORY**

14066 First released in Issue 4.

14067 **Issue 5**14068 The following change has been made in this issue for alignment with ISO/IEC
14069 9899:1990/Amendment 1:1994 (E).

- 14070
- The SYNOPSIS has been changed to indicate that this function and associated data types are
14071 now made visible by inclusion of the header <wctype.h> rather than <wchar.h>.

14072 **NAME**

14073 iswupper — test for an upper-case letter wide-character code

14074 **SYNOPSIS**

14075 #include <wctype.h>

14076 int iswupper(wint_t wc);

14077 **DESCRIPTION**

14078 The *iswupper()* function tests whether *wc* is a wide-character code representing a character of
14079 class **upper** in the program's current locale, see the **XBD** specification, **Chapter 5, Locale**.

14080 In all cases *wc* is a **wint_t**, the value of which must be a wide-character code corresponding to a
14081 valid character in the current locale or must equal the value of the macro WEOF. If the
14082 argument has any other value, the behaviour is undefined.

14083 **RETURN VALUE**

14084 The *iswupper()* function returns non-zero if *wc* is an upper-case letter wide-character code;
14085 otherwise it returns 0.

14086 **ERRORS**

14087 No errors are defined.

14088 **EXAMPLES**

14089 None.

14090 **APPLICATION USAGE**

14091 To ensure applications portability, especially across natural languages, only this function and
14092 those listed in the SEE ALSO section should be used for classification of wide-character codes.

14093 **FUTURE DIRECTIONS**

14094 None.

14095 **SEE ALSO**

14096 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*,
14097 *iswspace()*, *iswxdigit()*, *setlocale()*, <wctype.h>, <wchar.h>, the **XBD** specification, **Chapter 5,**
14098 **Locale**.

14099 **CHANGE HISTORY**

14100 First released in Issue 4.

14101 **Issue 5**

14102 The following change has been made in this issue for alignment with ISO/IEC
14103 9899:1990/Amendment 1:1994 (E).

- 14104 • The SYNOPSIS has been changed to indicate that this function and associated data types are
14105 now made visible by inclusion of the header <wctype.h> rather than <wchar.h>.

14106 **NAME**

14107 iswxdigit — test for a hexadecimal digit wide-character code

14108 **SYNOPSIS**

14109 #include <wctype.h>

14110 int iswxdigit(wint_t wc);

14111 **DESCRIPTION**14112 The *iswxdigit()* function tests whether *wc* is a wide-character code representing a character of
14113 class **xdigit** in the program's current locale, see the **XBD** specification, **Chapter 5, Locale**.14114 In all cases *wc* is a **wint_t**, the value of which must be a wide-character code corresponding to a
14115 valid character in the current locale or must equal the value of the macro **WEOF**. If the
14116 argument has any other value, the behaviour is undefined.14117 **RETURN VALUE**14118 The *iswxdigit()* function returns non-zero if *wc* is a hexadecimal digit wide-character code;
14119 otherwise it returns 0.14120 **ERRORS**

14121 No errors are defined.

14122 **EXAMPLES**

14123 None.

14124 **APPLICATION USAGE**14125 To ensure applications portability, especially across natural languages, only this function and
14126 those listed in the **SEE ALSO** section should be used for classification of wide-character codes.14127 **FUTURE DIRECTIONS**

14128 None.

14129 **SEE ALSO**14130 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*,
14131 *iswspace()*, *iswupper()*, *setlocale()*, <**wctype.h**>, <**wchar.h**>.14132 **CHANGE HISTORY**

14133 First released in Issue 4.

14134 **Issue 5**14135 The following change has been made in this issue for alignment with ISO/IEC
14136 9899:1990/Amendment 1:1994 (E).

- 14137
- The **SYNOPSIS** has been changed to indicate that this function and associated data types are
14138 now made visible by inclusion of the header <**wctype.h**> rather than <**wchar.h**>.

14139 **NAME**

14140 isxdigit — test for a hexadecimal digit

14141 **SYNOPSIS**

14142 #include <ctype.h>

14143 int isxdigit(int c);

14144 **DESCRIPTION**

14145 The *isxdigit()* function tests whether *c* is a character of class **xdigit** in the program's current
14146 locale, see the **XBD** specification, **Chapter 5, Locale**.

14147 In all cases *c* is an **int**, the value of which must be a character representable as an **unsigned char**
14148 or must equal the value of the macro **EOF**. If the argument has any other value, the behaviour is
14149 undefined.

14150 **RETURN VALUE**

14151 The *isxdigit()* function returns non-zero if *c* is a hexadecimal digit; otherwise it returns 0.

14152 **ERRORS**

14153 No errors are defined.

14154 **EXAMPLES**

14155 None.

14156 **APPLICATION USAGE**

14157 To ensure applications portability, especially across natural languages, only this function and
14158 those listed in the **SEE ALSO** section should be used for character classification.

14159 **FUTURE DIRECTIONS**

14160 None.

14161 **SEE ALSO**

14162 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*,
14163 <ctype.h>.

14164 **CHANGE HISTORY**

14165 First released in Issue 1.

14166 Derived from Issue 1 of the SVID.

14167 **Issue 4**

14168 The following change is incorporated in this issue:

- 14169 • The text of the **DESCRIPTION** is revised, although there are no functional differences
14170 between this issue and Issue 3.

14171 **NAME**

14172 j0, j1, jn — Bessel functions of the first kind

14173 **SYNOPSIS**

14174 EX #include <math.h>

14175 double j0(double x);

14176 double j1(double x);

14177 double jn(int n, double x);

14178

14179 **DESCRIPTION**14180 The *j0()*, *j1()* and *jn()* functions compute Bessel functions of *x* of the first kind of orders 0, 1 and
14181 *n* respectively.14182 An application wishing to check for error situations should set *errno* to 0 before calling *j0()*, *j1()*
14183 or *jn()*. If *errno* is non-zero on return, or the return value is NaN, an error has occurred.14184 **RETURN VALUE**14185 Upon successful completion, *j0()*, *j1()* and *jn()* return the relevant Bessel value of *x* of the first
14186 kind.14187 If the *x* argument is too large in magnitude, 0 is returned and *errno* may be set to [ERANGE].14188 If *x* is NaN, NaN is returned and *errno* may be set to [EDOM].14189 If the correct result would cause underflow, 0 is returned and *errno* may be set to [ERANGE].14190 **ERRORS**14191 The *j0()*, *j1()* and *jn()* functions may fail if:14192 [EDOM] The value of *x* is NaN.14193 [ERANGE] The value of *x* was too large in magnitude, or underflow occurred.

14194 No other errors will occur.

14195 **EXAMPLES**

14196 None.

14197 **APPLICATION USAGE**

14198 None.

14199 **FUTURE DIRECTIONS**

14200 None.

14201 **SEE ALSO**

14202 isnan(), y0(), <math.h>.

14203 **CHANGE HISTORY**

14204 First released in Issue 1.

14205 Derived from Issue 1 of the SVID.

14206 **Issue 4**

14207 The following changes are incorporated in this issue:

14208 • References to *matherr()* are removed.14209 • The RETURN VALUE and ERRORS sections are substantially rewritten to rationalise error
14210 handling in the mathematics functions.

14211 **Issue 5**

14212 The DESCRIPTION is updated to indicate how an application should check for an error. This
14213 text was previously published in the APPLICATION USAGE section.

14214 **NAME**

14215 jrand48 — generate a uniformly distributed pseudo-random long signed integer

14216 **SYNOPSIS**

14217 EX #include <stdlib.h>

14218 long int jrand48(unsigned short int xsubi[3]);

14219

14220 **DESCRIPTION**

14221 Refer to *drand48()*.

14222 **CHANGE HISTORY**

14223 First released in Issue 1.

14224 Derived from Issue 1 of the SVID.

14225 **Issue 4**

14226 The following changes are incorporated this issue:

- 14227 • The <stdlib.h> header is added to the SYNOPSIS section.
- 14228 • The word **long** is replaced by the words **long int** in the SYNOPSIS section.

14229 **NAME**14230 **kill** — send a signal to a process or a group of processes14231 **SYNOPSIS**14232 OH `#include <sys/types.h>`14233 `#include <signal.h>`14234 `int kill(pid_t pid, int sig);`14235 **DESCRIPTION**

14236 The *kill()* function will send a signal to a process or a group of processes specified by *pid*. The
 14237 signal to be sent is specified by *sig* and is either one from the list given in **<signal.h>** or 0. If *sig* is
 14238 0 (the null signal), error checking is performed but no signal is actually sent. The null signal can
 14239 be used to check the validity of *pid*.

14240 { _POSIX_SAVED_IDS } will be defined on all XSI-conformant systems, and for a process to have
 14241 permission to send a signal to a process designated by *pid*, the real or effective user ID of the
 14242 FIPS sending process must match the real or saved set-user-ID of the receiving process, unless the
 14243 sending process has appropriate privileges.

14244 If *pid* is greater than 0, *sig* will be sent to the process whose process ID is equal to *pid*.

14245 If *pid* is 0, *sig* will be sent to all processes (excluding an unspecified set of system processes)
 14246 whose process group ID is equal to the process group ID of the sender, and for which the
 14247 process has permission to send a signal.

14248 EX If *pid* is -1, *sig* will be sent to all processes (excluding an unspecified set of system processes) for
 14249 which the process has permission to send that signal.

14250 If *pid* is negative, but not -1, *sig* will be sent to all processes (excluding an unspecified set of
 14251 system processes) whose process group ID is equal to the absolute value of *pid*, and for which
 14252 the process has permission to send a signal.

14253 If the value of *pid* causes *sig* to be generated for the sending process, and if *sig* is not blocked for
 14254 the calling thread and if no other thread has *sig* unblocked or is waiting in a *sigwait()* function
 14255 for *sig*, either *sig* or at least one pending unblocked signal will be delivered to the sending thread
 14256 before *kill()* returns.

14257 The user ID tests described above will not be applied when sending SIGCONT to a process that
 14258 is a member of the same session as the sending process.

14259 An implementation that provides extended security controls may impose further
 14260 implementation-dependent restrictions on the sending of signals, including the null signal. In
 14261 particular, the system may deny the existence of some or all of the processes specified by *pid*.

14262 The *kill()* function is successful if the process has permission to send *sig* to any of the processes
 14263 specified by *pid*. If *kill()* fails, no signal will be sent.

14264 **RETURN VALUE**

14265 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate
 14266 the error.

14267 **ERRORS**

14268 The *kill()* function will fail if:

14269 [EINVAL] The value of the *sig* argument is an invalid or unsupported signal number.

14270 [EPERM] The process does not have permission to send the signal to any receiving
 14271 process.

14272 [ESRCH] No process or process group can be found corresponding to that specified by
 14273 *pid*.

14274 EXAMPLES

14275 None.

14276 APPLICATION USAGE

14277 None.

14278 FUTURE DIRECTIONS

14279 None.

14280 SEE ALSO

14281 *getpid()*, *raise()*, *setsid()*, *sigaction()*, **<signal.h>**, *sigqueue()*, **<sys/types.h>**.

14282 CHANGE HISTORY

14283 First released in Issue 1.

14284 Derived from Issue 1 of the SVID.

14285 Issue 4

14286 The following change is incorporated for alignment with the FIPS requirements:

- 14287 • In the DESCRIPTION, the second paragraph is reworded to indicate that the saved set-user-
 14288 ID of the calling process will be checked in place of its effective user ID. This functionality is
 14289 marked as an extension.

14290 Other changes are incorporated as follows:

- 14291 • The **<sys/types.h>** header is now marked as optional (OH); this header need not be included
 14292 on XSI-conformant systems.
- 14293 • The DESCRIPTION is clarified in various places.

14294 Issue 5

14295 The DESCRIPTION is updated for alignment with POSIX Threads Extension.

14296 **NAME**

14297 killpg — send a signal to a process group

14298 **SYNOPSIS**

14299 EX #include <signal.h>

14300 int killpg(pid_t pgrp, int sig);

14301

14302 **DESCRIPTION**14303 The *killpg()* function sends the signal specified by *sig* to the process group specified by *pgrp*.14304 If *pgrp* is greater than 1, *killpg(pgrp, sig)* is equivalent to *kill(-pgrp, sig)*. If *pgrp* is less than or
14305 equal to 1, the behaviour of *killpg()* is undefined.14306 **RETURN VALUE**14307 Refer to *kill()*.14308 **ERRORS**14309 Refer to *kill()*.14310 **EXAMPLES**

14311 None.

14312 **APPLICATION USAGE**

14313 None.

14314 **FUTURE DIRECTIONS**

14315 None.

14316 **SEE ALSO**14317 *getpgid()*, *getpid()*, *kill()*, *raise()*, <signal.h>.14318 **CHANGE HISTORY**

14319 First released in Issue 4, Version 2.

14320 **Issue 5**

14321 Moved from X/OPEN UNIX extension to BASE.

14322 **NAME**

14323 l64a — convert a 32-bit integer to a radix-64 ASCII string

14324 **SYNOPSIS**

14325 EX #include <stdlib.h>

14326 char *l64a(long value);

14327

14328 **DESCRIPTION**14329 Refer to *a64l()*.14330 **CHANGE HISTORY**

14331 First released in Issue 4, Version 2.

14332 **Issue 5**

14333 Moved from X/OPEN UNIX extension to BASE.

14334 **NAME**

14335 labs — return a long integer absolute value

14336 **SYNOPSIS**

14337 #include <stdlib.h>

14338 long int labs(long int *i*);14339 **DESCRIPTION**14340 The *labs()* function computes the absolute value of its long integer operand, *i*. If the result
14341 cannot be represented, the behaviour is undefined.14342 **RETURN VALUE**14343 The *labs()* function returns the absolute value of its long integer operand.14344 **ERRORS**

14345 No errors are defined.

14346 **EXAMPLES**

14347 None.

14348 **APPLICATION USAGE**

14349 None.

14350 **FUTURE DIRECTIONS**

14351 None.

14352 **SEE ALSO**14353 *abs()*, <stdlib.h>.14354 **CHANGE HISTORY**

14355 First released in Issue 4.

14356 Derived from the ISO C standard.

14357 **NAME**

14358 lchown — change the owner and group of a symbolic link

14359 **SYNOPSIS**14360 EX `#include <unistd.h>`14361 `int lchown(const char *path, uid_t owner, gid_t group);`

14362

14363 **DESCRIPTION**

14364 The *lchown()* function has the same effect as *chown()* except in the case where the named file is a
 14365 symbolic link. In this case *lchown()* changes the ownership of the symbolic link file itself, while
 14366 *chown()* changes the ownership of the file or directory to which the symbolic link refers.

14367 **RETURN VALUE**

14368 Upon successful completion, *lchown()* returns 0. Otherwise, it returns *-1* and sets *errno* to
 14369 indicate an error.

14370 **ERRORS**14371 The *lchown()* function will fail if:14372 [EACCES] Search permission is denied on a component of the path prefix of *path*.

14373 [EINVAL] The owner or group id is not a value supported by the implementation.

14374 [ENAMETOOLONG]

14375 The length of a pathname exceeds {PATH_MAX}, or pathname component is
 14376 longer than {NAME_MAX}.

14377 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.14378 [ENOTDIR] A component of the path prefix of *path* is not a directory.

14379 [EOPNOTSUPP] The *path* argument names a symbolic link and the implementation does not
 14380 support setting the owner or group of a symbolic link.

14381 [ELOOP] Too many symbolic links were encountered in resolving *path*.

14382 [EPERM] The effective user ID does not match the owner of the file and the process
 14383 does not have appropriate privileges.

14384 [EROFS] The file resides on a read-only file system.

14385 The *lchown()* function may fail if:

14386 [EIO] An I/O error occurred while reading or writing to the file system.

14387 [INTR] A signal was caught during execution of the function.

14388 [ENAMETOOLONG]

14389 Pathname resolution of a symbolic link produced an intermediate result
 14390 whose length exceeds {PATH_MAX}.

14391 **EXAMPLES**

14392 None.

14393 **APPLICATION USAGE**

14394 None.

14395 **FUTURE DIRECTIONS**

14396 None.

14397 SEE ALSO

14398 *chown()*, *symlink()*, **<unistd.h>**.

14399 CHANGE HISTORY

14400 First released in Issue 4, Version 2.

14401 Issue 5

14402 Moved from X/OPEN UNIX extension to BASE.

14403 **NAME**

14404 lcong48 — seed a uniformly distributed pseudo-random signed long integer generator

14405 **SYNOPSIS**

14406 EX #include <stdlib.h>

14407 void lcong48(unsigned short int param[7]);

14408

14409 **DESCRIPTION**14410 Refer to *drand48()*.14411 **CHANGE HISTORY**

14412 First released in Issue 1.

14413 Derived from Issue 1 of the SVID.

14414 **Issue 4**

14415 The following change is incorporated in this issue:

- 14416
- The <stdlib.h> header is now included in the SYNOPSIS section.

14417 **NAME**

14418 ldexp — load exponent of a floating point number

14419 **SYNOPSIS**

14420 #include <math.h>

14421 double ldexp(double x, int exp);

14422 **DESCRIPTION**14423 The *ldexp()* function computes the quantity $x * 2^{exp}$.

14424 An application wishing to check for error situations should set *errno* to 0 before calling *ldexp()*.
 14425 If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

14426 **RETURN VALUE**

14427 Upon successful completion, *ldexp()* returns a **double** representing the value *x* multiplied by 2
 14428 raised to the power *exp*.

14429 EX If the value of *x* is NaN, NaN is returned and *errno* may be set to [EDOM].

14430 If *ldexp()* would cause overflow, $\pm\text{HUGE_VAL}$ is returned (according to the sign of *x*), and *errno*
 14431 is set to [ERANGE].

14432 If *ldexp()* would cause underflow, 0 is returned and *errno* may be set to [ERANGE].14433 **ERRORS**14434 The *ldexp()* function will fail if:

14435 [ERANGE] The value to be returned would have caused overflow.

14436 The *ldexp()* function may fail if:14437 EX [EDOM] The argument *x* is NaN.

14438 [ERANGE] The value to be returned would have caused underflow.

14439 No other errors will occur.

14440 **EXAMPLES**

14441 None.

14442 **APPLICATION USAGE**

14443 None.

14444 **FUTURE DIRECTIONS**

14445 None.

14446 **SEE ALSO**14447 *frexp()*, *isnan()*, <math.h>.14448 **CHANGE HISTORY**

14449 First released in Issue 1.

14450 Derived from Issue 1 of the SVID.

14451 **Issue 4**

14452 The following changes are incorporated in this issue:

14453 • Removed references to *matherr()*.

14454 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
 14455 the ISO C standard and to rationalise error handling in the mathematics functions.

14456 • The return value specified for [EDOM] is marked as an extension.

14457 **Issue 5**

14458 The DESCRIPTION is updated to indicate how an application should check for an error. This
14459 text was previously published in the APPLICATION USAGE section.

14460 **NAME**

14461 ldiv — compute quotient and remainder of a long division

14462 **SYNOPSIS**

14463 #include <stdlib.h>

14464 ldiv_t ldiv(long int *numer*, long int *denom*);

14465 **DESCRIPTION**

14466 The *ldiv()* function computes the quotient and remainder of the division of the numerator *numer*
14467 by the denominator *denom*. If the division is inexact, the resulting quotient is the long integer of
14468 lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented,
14469 the behaviour is undefined; otherwise, *quot * denom + rem* will equal *numer*.

14470 **RETURN VALUE**

14471 The *ldiv()* function returns a structure of type **ldiv_t**, comprising both the quotient and the
14472 remainder. The structure includes the following members, in any order:

14473 long int quot; /* quotient */
14474 long int rem; /* remainder */

14475 **ERRORS**

14476 No errors are defined.

14477 **EXAMPLES**

14478 None.

14479 **APPLICATION USAGE**

14480 None.

14481 **FUTURE DIRECTIONS**

14482 None.

14483 **SEE ALSO**

14484 (), <stdlib.h>.

14485 **CHANGE HISTORY**

14486 First released in Issue 4.

14487 Derived from the ISO C standard.

14488 **NAME**

14489 lfind — find entry in a linear search table

14490 **SYNOPSIS**

14491 EX #include <search.h>

```
14492       void *lfind(const void *key, const void *base, size_t *nelp,  
14493                   size_t width, int (*compar)(const void *, const void *));
```

14494

14495 **DESCRIPTION**14496 Refer to *lsearch()*.14497 **CHANGE HISTORY**

14498 First released in Issue 1.

14499 Derived from Issue 1 of the SVID.

14500 **Issue 4**

14501 The following change is incorporated in this issue:

- 14502 • In the SYNOPSIS section, the type of the function return value is changed from **char*** to
14503 **void***, the type of the *key* and *base* arguments is changed from **void*** to **const void***, and
14504 argument declarations for *compar()* are added.

14505 **NAME**

14506 lgamma — log gamma function

14507 **SYNOPSIS**14508 EX `#include <math.h>`14509 `double lgamma(double x);`14510 `extern int signgam;`

14511

14512 **DESCRIPTION**

14513 The *lgamma()* function computes $\log_e |\Gamma(x)|$ where $\Gamma(x)$ is defined as $\int_0^\infty e^{-t} t^{x-1} dt$. The sign of $\Gamma(x)$
 14514 is returned in the external integer *signgam*. The argument *x* need not be a non-positive integer
 14515 ($\Gamma(x)$ is defined over the reals, except the non-positive integers).
 14516

14517 An application wishing to check for error situations should set *errno* to 0 before calling *lgamma()*.
 14518 If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

14519 This interface need not be reentrant.

14520 **RETURN VALUE**14521 Upon successful completion, *lgamma()* returns the logarithmic gamma of *x*.14522 If *x* is NaN, NaN is returned and *errno* may be set to [EDOM].

14523 If *x* is a non-positive integer, either HUGE_VAL or NaN is returned and *errno* may be set to
 14524 [EDOM].

14525 If the correct value would cause overflow, *lgamma()* returns HUGE_VAL and may set *errno* to
 14526 [ERANGE].

14527 If the correct value would cause underflow, *lgamma()* returns 0 and may set *errno* to [ERANGE].14528 **ERRORS**14529 The *lgamma()* function may fail if:14530 [EDOM] The value of *x* is a non-positive integer or NaN.

14531 [ERANGE] The value to be returned would have caused overflow or underflow.

14532 No other errors will occur.

14533 **EXAMPLES**

14534 None.

14535 **APPLICATION USAGE**

14536 None.

14537 **FUTURE DIRECTIONS**

14538 None.

14539 **SEE ALSO**14540 *exp()*, *isnan()*, <math.h>.14541 **CHANGE HISTORY**

14542 First released in Issue 3.

14543 **Issue 4**

14544 The following changes are incorporated in this issue:

- 14545 • This page no longer points to *gamma()*, but contains all information relating to *lgamma()*.
- 14546 • The RETURN VALUE and ERRORS sections are substantially rewritten to rationalise error
- 14547 handling in the mathematics functions.

14548 **Issue 5**

14549 The DESCRIPTION is updated to indicate how an application should check for an error. This

14550 text was previously published in the APPLICATION USAGE section.

14551 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

14552 NAME

14553 link — link to a file

14554 SYNOPSIS

14555 #include <unistd.h>

14556 int link(const char *path1, const char *path2);

14557 DESCRIPTION

14558 The *link()* function creates a new link (directory entry) for the existing file, *path1*.

14559 The *path1* argument points to a pathname naming an existing file. The *path2* argument points to
 14560 a pathname naming the new directory entry to be created. The *link()* function will atomically
 14561 create a new link for the existing file and the link count of the file is incremented by one.

14562 If *path1* names a directory, *link()* will fail unless the process has appropriate privileges and the
 14563 implementation supports using *link()* on directories.

14564 Upon successful completion, *link()* will mark for update the *st_ctime* field of the file. Also, the
 14565 *st_ctime* and *st_mtime* fields of the directory that contains the new entry are marked for update.

14566 If *link()* fails, no link is created and the link count of the file will remain unchanged.

14567 The implementation may require that the calling process has permission to access the existing
 14568 file.

14569 RETURN VALUE

14570 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate
 14571 the error.

14572 ERRORS

14573 The *link()* function will fail if:

14574 [EACCES] A component of either path prefix denies search permission, or the requested
 14575 link requires writing in a directory with a mode that denies write permission,
 14576 or the calling process does not have permission to access the existing file and
 14577 this is required by the implementation.

14578 [EEXIST] The link named by *path2* exists.14579 EX [ELOOP] Too many symbolic links were encountered in resolving *path1* or *path2*.14580 [EMLINK] The number of links to the file named by *path1* would exceed {LINK_MAX}.

14581 [ENAMETOOLONG]

14582 FIPS The length of *path1* or *path2* exceeds {PATH_MAX} or a pathname component
 14583 is longer than {NAME_MAX}.

14584 [ENOENT] A component of either path prefix does not exist; the file named by *path1* does
 14585 not exist; or *path1* or *path2* points to an empty string.

14586 [ENOSPC] The directory to contain the link cannot be extended.

14587 [ENOTDIR] A component of either path prefix is not a directory.

14588 [EPERM] The file named by *path1* is a directory and either the calling process does not
 14589 have appropriate privileges or the implementation prohibits using *link()* on
 14590 directories.

14591 [EROFS] The requested link requires writing in a directory on a read-only file system.

14592 [EXDEV] The link named by *path2* and the file named by *path1* are on different file
 14593 systems and the implementation does not support links between file systems,
 14594 EX or *path1* refers to a named STREAM.

14595 The *link()* function may fail if:

14596 EX [ENAMETOOLONG]
 14597 Pathname resolution of a symbolic link produced an intermediate result
 14598 whose length exceeds {PATH_MAX}.

14599 EXAMPLES

14600 None.

14601 APPLICATION USAGE

14602 Some implementations do allow links between file systems.

14603 FUTURE DIRECTIONS

14604 None.

14605 SEE ALSO

14606 *symlink()*, *unlink()*, <unistd.h>.

14607 CHANGE HISTORY

14608 First released in Issue 1.

14609 Derived from Issue 1 of the SVID.

14610 Issue 4

14611 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 14612 • The type of arguments *path1* and *path2* are changed from **char *** to **const char ***.

14613 The following change is incorporated for alignment with the FIPS requirements:

- 14614 • In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a
 14615 pathname component is larger than {NAME_MAX} is now defined as mandatory and marked
 14616 as an extension.

14617 Other changes are incorporated as follows:

- 14618 • The <unistd.h> header is added to the SYNOPSIS section.

14619 Issue 4, Version 2

14620 The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- 14621 • The [ELOOP] error will be returned if too many symbolic links are encountered during
 14622 pathname resolution.
- 14623 • The [EXDEV] error may also be returned if *path1* refers to a named STREAM.
- 14624 • A second [ENAMETOOLONG] condition is defined that may report excessive length of an
 14625 intermediate result of pathname resolution of a symbolic link.

14626 **NAME**14627 lio_listio — list directed I/O (**REALTIME**)14628 **SYNOPSIS**

14629 RT #include <aio.h>

```
14630       int lio_listio(int mode, struct aiocb * const list[], int nent,
14631                      struct sigevent *sig);
```

14632

14633 **DESCRIPTION**

14634 The *lio_listio()* function allows the calling process to initiate a list of I/O requests with a single
 14635 function call.

14636 The *mode* argument takes one of the values LIO_WAIT or LIO_NOWAIT declared in <aio.h> and
 14637 determines whether the function returns when the I/O operations have been completed, or as
 14638 soon as the operations have been queued. If the *mode* argument is LIO_WAIT, the function waits
 14639 until all I/O is complete and the *sig* argument is ignored.

14640 If the *mode* argument is LIO_NOWAIT, the function returns immediately, and asynchronous
 14641 notification occurs, according to the *sig* argument, when all the I/O operations complete. If *sig* is
 14642 NULL, then no asynchronous notification occurs. If *sig* is not NULL, asynchronous notification
 14643 occurs as specified in **Signal Generation and Delivery** on page 808 when all the requests in *list*
 14644 have completed.

14645 The I/O requests enumerated by *list* are submitted in an unspecified order.

14646 The *list* argument is an array of pointers to **aiocb** structures. The array contains *nent* elements.
 14647 The array may contain NULL elements, which are ignored.

14648 The *aio_lio_opcode* field of each **aiocb** structure specifies the operation to be performed. The
 14649 supported operations are LIO_READ, LIO_WRITE and LIO_NOP; these symbols are defined in
 14650 <aio.h>. The LIO_NOP operation causes the list entry to be ignored. If the *aio_lio_opcode*
 14651 element is equal to LIO_READ, then an I/O operation is submitted as if by a call to *aio_read()*
 14652 with the *aiocbp* equal to the address of the **aiocb** structure. If the *aio_lio_opcode* element is equal
 14653 to LIO_WRITE, then an I/O operation is submitted as if by a call to *aio_write()* with the *aiocbp*
 14654 equal to the address of the **aiocb** structure.

14655 The *aio_fildes* member specifies the file descriptor on which the operation is to be performed.

14656 The *aio_buf* member specifies the address of the buffer to or from which the data is to be
 14657 transferred.

14658 The *aio_nbytes* member specifies the number of bytes of data to be transferred.

14659 The members of the *aiocb* structure further describe the I/O operation to be performed, in a
 14660 manner identical to that of the corresponding **aiocb** structure when used by the *aio_read()* and
 14661 *aio_write()* functions.

14662 The *nent* argument specifies how many elements are members of the list, that is, the length of the
 14663 array.

14664 The behaviour of this function is altered according to the definitions of synchronised I/O data
 14665 integrity completion and synchronised I/O file integrity completion. If synchronised I/O is
 14666 enabled on the file associated with *aio_fildes*.

14667 EX For regular files, no data transfer will occur past the offset maximum established in the open file
 14668 description associated with *aiocbp->aio_fildes*.

14669 **RETURN VALUE**

14670 If the *mode* argument has the value `LIO_NOWAIT`, the *lio_listio()* function returns the value zero
 14671 if the I/O operations are successfully queued; otherwise, the function returns the value `-1` and
 14672 sets *errno* to indicate the error.

14673 If the *mode* argument has the value `LIO_WAIT`, the *lio_listio()* function returns the value zero
 14674 when all the indicated I/O has completed successfully. Otherwise, *lio_listio()* returns a value of
 14675 `-1` and sets *errno* to indicate the error.

14676 In either case, the return value only indicates the success or failure of the *lio_listio()* call itself,
 14677 not the status of the individual I/O requests. In some cases one or more of the I/O requests
 14678 contained in the list may fail. Failure of an individual request does not prevent completion of
 14679 any other individual request. To determine the outcome of each I/O request, the application
 14680 examines the error status associated with each **aiocb** control block. The error statuses so
 14681 returned are identical to those returned as the result of an *aio_read()* or *aio_write()* function.

14682 **ERRORS**

14683 The *lio_listio()* function will fail if:

14684 [EAGAIN] The resources necessary to queue all the I/O requests were not available. The
 14685 application may check the error status for each **aiocb** to determine the
 14686 individual request(s) that failed.

14687 [EAGAIN] The number of entries indicated by *nent* would cause the systemwide limit
 14688 `AIO_MAX` to be exceeded.

14689 [EINVAL] The *mode* argument is not a proper value, or the value of *nent* was greater than
 14690 `AIO_LISTIO_MAX`.

14691 [EINTR] A signal was delivered while waiting for all I/O requests to complete during a
 14692 `LIO_WAIT` operation. Note that, since each I/O operation invoked by
 14693 *lio_listio()* may possibly provoke a signal when it completes, this error return
 14694 may be caused by the completion of one (or more) of the very I/O operations
 14695 being awaited. Outstanding I/O requests are not canceled, and the
 14696 application examines each list element to determine whether the request was
 14697 initiated, canceled, or completed.

14698 [EIO] One or more of the individual I/O operations failed. The application may
 14699 check the error status for each **aiocb** structure to determine the individual
 14700 request(s) that failed.

14701 [ENOSYS] The *lio_listio()* function is not supported by this implementation.

14702 In addition to the errors returned by the *lio_listio()* function, if the *lio_listio()* function succeeds
 14703 or fails with errors of [EAGAIN], [EINTR], or [EIO], then some of the I/O specified by the list
 14704 may have been initiated. If the *lio_listio()* function fails with an error code other than
 14705 [EAGAIN], [EINTR], or [EIO], no operations from the list have been initiated. The I/O operation
 14706 indicated by each list element can encounter errors specific to the individual read or write
 14707 function being performed. In this event, the error status for each **aiocb** control block contains
 14708 the associated error code. The error codes that can be set are the same as would be set by a
 14709 *read()* or *write()* function, with the following additional error codes possible:

14710 [EAGAIN] The requested I/O operation was not queued due to resource limitations.

14711 [ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit
 14712 *aio_cancel()* request.

14713 EX [EFBIG] The *aiocbp->aio_lio_opcode* is `LIO_WRITE`, the file is a regular file, *aiocbp-*
 14714 *>aio_nbytes* is greater than 0, and the *aiocbp->aio_offset* is greater than or equal

14715		to the offset maximum in the open file description associated with <i>aiochp-</i>	
14716		<i>>aio_fildes</i> .	
14717		[EINPROGRESS] The requested I/O is in progress.	
14718 EX		[EOVERFLOW] The <i>aiochp->aio_lio_opcode</i> is LIO_READ, the file is a regular file, <i>aiochp-</i>	
14719		<i>>aio_nbytes</i> is greater than 0, and the <i>aiochp->aio_offset</i> is before the end-of-file	
14720		and is greater than or equal to the offset maximum in the open file description	
14721		associated with <i>aiochp->aio_fildes</i> .	
14722	EXAMPLES		
14723		None.	
14724	APPLICATION USAGE		
14725		None.	
14726	FUTURE DIRECTIONS		
14727		None.	
14728	SEE ALSO		
14729		<i>aio_read()</i> , <i>aio_write()</i> , <i>aio_error()</i> , <i>aio_return()</i> , <i>aio_cancel()</i> , <i>read()</i> , <i>lseek()</i> , <i>close()</i> , <i>_exit()</i> , <i>exec</i> ,	
14730		<i>fork()</i> .	
14731	CHANGE HISTORY		
14732		First released in Issue 5.	
14733		Included for alignment with the POSIX Realtime Extension.	
14734		Large File Summit extensions added.	

14735 **NAME**14736 loc1, loc2 — pointers to characters matched by regular expressions (**LEGACY**)14737 **SYNOPSIS**14738 EX `#include <regex.h>`14739 `extern char *loc1;`14740 `extern char *loc2;`

14741

14742 **DESCRIPTION**14743 Refer to *regex()*.14744 **APPLICATION USAGE**

14745 These variables are kept for historical reasons, but may be withdrawn in a future issue.

14746 New applications should use *fnmatch()*, *glob()*, *regcomp()* and *regexexec()*, which provide full
 14747 internationalised regular expression functionality compatible with the ISO POSIX-2 standard, as
 14748 described in the **XBD** specification, **Chapter 7, Regular Expressions**.

14749 **CHANGE HISTORY**

14750 First released in Issue 2.

14751 Derived from Issue 2 of the SVID.

14752 **Issue 4**

14753 The following changes are incorporated in this issue:

- 14754 • The **<regex.h>** header is added to the SYNOPSIS section.
- 14755 • The interfaces are marked TO BE WITHDRAWN, because improved functionality is now
- 14756 provided by interfaces introduced for alignment with the ISO POSIX-2 standard.

14757 **Issue 5**

14758 Marked LEGACY.

14759 **NAME**

14760 localeconv — determine the program locale

14761 **SYNOPSIS**

14762 #include <locale.h>

14763 struct lconv *localeconv(void);

14764 **DESCRIPTION**

14765 The *localeconv()* function sets the components of an object with the type **struct lconv** with the
 14766 values appropriate for the formatting of numeric quantities (monetary and otherwise) according
 14767 to the rules of the current locale.

14768 The members of the structure with type **char *** are pointers to strings, any of which (except
 14769 **decimal_point**) can point to "", to indicate that the value is not available in the current locale or
 14770 is of zero length. The members with type **char** are non-negative numbers, any of which can be
 14771 {CHAR_MAX} to indicate that the value is not available in the current locale.

14772 The members include the following:

14773 **char *decimal_point**

14774 The radix character used to format non-monetary quantities.

14775 **char *thousands_sep**

14776 The character used to separate groups of digits before the decimal-point character in
 14777 formatted non-monetary quantities.

14778 **char *grouping**

14779 A string whose elements taken as one-byte integer values indicate the size of each group of
 14780 digits in formatted non-monetary quantities.

14781 **char *int_curr_symbol**

14782 The international currency symbol applicable to the current locale. The first three
 14783 characters contain the alphabetic international currency symbol in accordance with those
 14784 specified in the ISO 4217:1987 standard. The fourth character (immediately preceding the
 14785 null byte) is the character used to separate the international currency symbol from the
 14786 monetary quantity.

14787 **char *currency_symbol**

14788 The local currency symbol applicable to the current locale.

14789 **char *mon_decimal_point**

14790 The radix character used to format monetary quantities.

14791 **char *mon_thousands_sep**

14792 The separator for groups of digits before the decimal-point in formatted monetary
 14793 quantities.

14794 **char *mon_grouping**

14795 A string whose elements taken as one-byte integer values indicate the size of each group of
 14796 digits in formatted monetary quantities.

14797 **char *positive_sign**

14798 The string used to indicate a non-negative valued formatted monetary quantity.

14799 **char *negative_sign**

14800 The string used to indicate a negative valued formatted monetary quantity.

14801 **char int_frac_digits**

14802 The number of fractional digits (those after the decimal-point) to be displayed in an

14803 internationally formatted monetary quantity.

14804 **char frac_digits**

14805 The number of fractional digits (those after the decimal-point) to be displayed in a

14806 formatted monetary quantity.

14807 **char p_cs_precedes**

14808 EX Set to 1 if the **currency_symbol** or **int_curr_symbol** precedes the value for a non-negative

14809 formatted monetary quantity. Set to 0 if the symbol succeeds the value.

14810 **char p_sep_by_space**

14811 EX Set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value for a

14812 non-negative formatted monetary quantity. Set to 1 if a space separates the symbol from the

14813 EX value; and set to 2 if a space separates the symbol and the sign string, if adjacent.

14814 **char n_cs_precedes**

14815 EX Set to 1 if the **currency_symbol** or **int_curr_symbol** precedes the value for a negative

14816 formatted monetary quantity. Set to 0 if the symbol succeeds the value.

14817 **char n_sep_by_space**

14818 EX Set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value for a

14819 negative formatted monetary quantity. Set to 1 if a space separates the symbol from the

14820 EX value; and set to 2 if a space separates the symbol and the sign string, if adjacent.

14821 **char p_sign_posn**

14822 Set to a value indicating the positioning of the **positive_sign** for a non-negative formatted

14823 monetary quantity.

14824 **char n_sign_posn**

14825 Set to a value indicating the positioning of the **negative_sign** for a negative formatted

14826 monetary quantity.

14827 The elements of **grouping** and **mon_grouping** are interpreted according to the following:

14828 {CHAR_MAX} No further grouping is to be performed.

14829 0 The previous element is to be repeatedly used for the remainder of the digits.

14830 *other* The integer value is the number of digits that comprise the current group. The

14831 next element is examined to determine the size of the next group of digits

14832 before the current group.

14833 The values of **p_sign_posn** and **n_sign_posn** are interpreted according to the following:

14834 EX 0 Parentheses surround the quantity and **currency_symbol** or **int_curr_symbol**.

14835 EX 1 The sign string precedes the quantity and **currency_symbol** or **int_curr_symbol**.

14836 EX 2 The sign string succeeds the quantity and **currency_symbol** or **int_curr_symbol**.

14837 EX 3 The sign string immediately precedes the **currency_symbol** or **int_curr_symbol**.

14838 EX 4 The sign string immediately succeeds the **currency_symbol** or **int_curr_symbol**.

14839 The implementation will behave as if no function in this specification calls *localeconv()*.

14840 **RETURN VALUE**

14841 The *localeconv()* function returns a pointer to the filled-in object. The structure pointed to by the

14842 return value must not be modified by the program, but may be overwritten by a subsequent call

14843 to *localeconv()*. In addition, calls to *setlocale()* with the categories LC_ALL, LC_MONETARY, or

14844 LC_NUMERIC may overwrite the contents of the structure.

14845 **ERRORS**

14846 No errors are defined.

14847 **EXAMPLES**

14848 None.

14849 **APPLICATION USAGE**

14850 The following table illustrates the rules which may be used by four countries to format monetary
 14851 quantities.

14852

14853

14854

14855

14856

14857

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

14858

14859

For these four countries, the respective values for the monetary members of the structure returned by *localeconv()* are:

14860

14861

14862

14863

14864

14865

14866

14867

14868

14869

14870

14871

14872

14873

14874

14875

14876

	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

14877 **FUTURE DIRECTIONS**

14878 None.

14879 **SEE ALSO**

14880 *isalpha()*, *isascii()*, *nl_langinfo()*, *printf()*, *scanf()*, *setlocale()*, *strcat()*, *strchr()*, *strcmp()*, *strcoll()*,
 14881 *strcpy()*, *strftime()*, *strlen()*, *strpbrk()*, *strspn()*, *strtok()*, *strxfrm()*, *strtod()*, **<langinfo.h>**,
 14882 **<locale.h>**.

14883 **CHANGE HISTORY**

14884 First released in Issue 4.

14885 Derived from the ANSI C standard.

14886 **NAME**

14887 localtime, localtime_r — convert a time value to a broken-down local time

14888 **SYNOPSIS**

14889 #include <time.h>

14890 struct tm *localtime(const time_t *timer);

14891 struct tm *localtime_r(const time_t *clock, struct tm *result);

14892 **DESCRIPTION**

14893 The *localtime()* function converts the time in seconds since the Epoch pointed to by *timer* into a
 14894 broken-down time, expressed as a local time. The function corrects for the timezone and any
 14895 seasonal time adjustments. Local timezone information is used as though *localtime()* calls
 14896 *tzset()*.

14897 The *localtime()* interface need not be reentrant.

14898 The *localtime_r()* function converts the calendar time pointed to by *clock* into a broken-down
 14899 time stored in the structure to which *result* points. The *localtime_r()* function also returns a
 14900 pointer to that same structure.

14901 Unlike *localtime()*, the reentrant version is not required to set *tzname*.

14902 **RETURN VALUE**

14903 The *localtime()* function returns a pointer to the broken-down time structure.

14904 Upon successful completion, *localtime_r()* returns a pointer to the structure pointed to by the
 14905 argument *result*.

14906 **ERRORS**

14907 No errors are defined.

14908 **EXAMPLES**

14909 None.

14910 **APPLICATION USAGE**

14911 The *asctime()*, *ctime()*, *getdate()*, *gettimeofday()*, *gmtime()* and *localtime()* functions return values
 14912 in one of two static objects: a broken-down time structure and an array of **char**. Execution of any
 14913 of the functions may overwrite the information returned in either of these objects by any of the
 14914 other functions.

14915 **FUTURE DIRECTIONS**

14916 None.

14917 **SEE ALSO**

14918 *asctime()*, *clock()*, *ctime()*, *difftime()*, *getdate()*, *gettimeofday()*, *gmtime()*, *mktime()*, *strftime()*,
 14919 *strptime()*, *time()*, *utime()*, <**time.h**>.

14920 **CHANGE HISTORY**

14921 First released in Issue 1.

14922 Derived from Issue 1 of the SVID.

14923 **Issue 4**

14924 The following change is incorporated for alignment with the ISO C standard:

- 14925 • The *timer* argument is now a type **const time_t**.

14926 Another change is incorporated as follows:

- 14927 • The APPLICATION USAGE section is expanded to provide a more complete description of
 14928 how static areas are used by the **time()* functions.

14929 **Issue 5**

14930 A note indicating that the *localtime()* interface need not be reentrant is added to the
14931 DESCRIPTION.

14932 The *localtime_r()* function is included for alignment with the POSIX Threads Extension.

14933 **NAME**

14934 lockf — record locking on files

14935 **SYNOPSIS**

14936 EX #include <unistd.h>

14937 int lockf(int *fildes*, int *function*, off_t *size*);

14938

14939 **DESCRIPTION**

14940 The *lockf()* function allows sections of a file to be locked with advisory-mode locks. Calls to
 14941 *lockf()* from other threads which attempt to lock the locked file section will either return an error
 14942 value or block until the section becomes unlocked. All the locks for a process are removed when
 14943 the process terminates. Record locking with *lockf()* is supported for regular files and may be
 14944 supported for other files.

14945 The *fildes* argument is an open file descriptor. The file descriptor must have been opened with
 14946 write-only permission (O_WRONLY) or with read/write permission (O_RDWR) to establish a
 14947 lock with this function.

14948 The *function* argument is a control value which specifies the action to be taken. The permissible
 14949 values for *function* are defined in <unistd.h> as follows:

14950

14951

14952

14953

14954

14955

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

14956 F_TEST detects if a lock by another process is present on the specified section; F_LOCK and
 14957 F_TLOCK both lock a section of a file if the section is available; F_ULOCK removes locks from a
 14958 section of the file.

14959 The *size* argument is the number of contiguous bytes to be locked or unlocked. The section to be
 14960 locked or unlocked starts at the current offset in the file and extends forward for a positive size
 14961 or backward for a negative size (the preceding bytes up to but not including the current offset).
 14962 If *size* is 0, the section from the current offset through the largest possible file offset is locked
 14963 (that is, from the current offset through the present or any future end-of-file). An area need not
 14964 be allocated to the file to be locked because locks may exist past the end-of-file.

14965 The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained
 14966 by a previously locked section for the same process. When this occurs, or if adjacent locked
 14967 sections would occur, the sections are combined into a single locked section. If the request
 14968 would cause the number of locks to exceed a system-imposed limit, the request will fail.

14969 F_LOCK and F_TLOCK requests differ only by the action taken if the section is not available.
 14970 F_LOCK blocks the calling thread until the section is available. F_TLOCK makes the function
 14971 fail if the section is already locked by another process.

14972 File locks are released on first close by the locking process of any file descriptor for the file.

14973 F_ULOCK requests may release (wholly or in part) one or more locked sections controlled by the
 14974 process. Locked sections will be unlocked starting at the current file offset through *size* bytes or
 14975 to the end of file if *size* is (off_t)0. When all of a locked section is not released (that is, when the
 14976 beginning or end of the area to be unlocked falls within a locked section), the remaining portions
 14977 of that section are still locked by the process. Releasing the center portion of a locked section
 14978 will cause the remaining locked beginning and end portions to become two separate locked

14979	sections. If the request would cause the number of locks in the system to exceed a system-imposed limit, the request will fail.	
14980		
14981	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, <i>lockf()</i> will fail with an [EDEADLK] error.	
14982		
14983		
14984	The interaction between <i>fcntl()</i> and <i>lockf()</i> locks is unspecified.	
14985	Blocking on a section is interrupted by any signal.	
14986 EX	An F_ULOCK request in which <i>size</i> is non-zero and the offset of the last byte of the requested section is the maximum value for an object of type <i>off_t</i> , when the process has an existing lock in which <i>size</i> is 0 and which includes the last byte of the requested section, will 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 will attempt to unlock only the requested section.	
14987		
14988		
14989		
14990		
14991	Attempting to lock a section of a file that is associated with a buffered stream produces unspecified results.	
14992		
14993	RETURN VALUE	
14994	Upon successful completion, <i>lockf()</i> returns 0. Otherwise, it returns -1, sets <i>errno</i> to indicate an error, and existing locks are not changed.	
14995		
14996	ERRORS	
14997	The <i>lockf()</i> function will fail if:	
14998	[EBADF] The <i>fildev</i> argument is not a valid open file descriptor; or <i>function</i> is F_LOCK or F_TLOCK and <i>fildev</i> is not a valid file descriptor open for writing.	
14999		
15000	[EACCES] or [EAGAIN]	
15001	The <i>function</i> argument is F_TLOCK or F_TEST and the section is already locked by another process.	
15002		
15003	[EDEADLK] The <i>function</i> argument is F_LOCK and a deadlock is detected.	
15004	[EINTR] A signal was caught during execution of the function.	
15005 EX	[EINVAL] The <i>function</i> argument is not one of F_LOCK, F_TLOCK, F_TEST or F_ULOCK; or <i>size</i> plus the current file offset is less than 0.	
15006		
15007 EX	[EOVERFLOW] The offset of the first, or if <i>size</i> is not 0 then the last, byte in the requested section cannot be represented correctly in an object of type <i>off_t</i> .	
15008		
15009	The <i>lockf()</i> function may fail if:	
15010	[EAGAIN] The <i>function</i> argument is F_LOCK or F_TLOCK and the file is mapped with <i>mmap()</i> .	
15011		
15012	[EDEADLK] or [ENOLCK]	
15013	The <i>function</i> argument is F_LOCK, F_TLOCK, or F_ULOCK, and the request would cause the number of locks to exceed a system-imposed limit.	
15014		
15015	[EOPNOTSUPP] or [EINVAL]	
15016	The implementation does not support the locking of files of the type indicated by the <i>fildev</i> argument.	
15017		
15018	EXAMPLES	
15019	None.	

15020 **APPLICATION USAGE**

15021 Record-locking should not be used in combination with the *fopen()*, *fread()*, *fwrite()* and other
15022 *stdio* functions. Instead, the more primitive, non-buffered functions (such as *open()*) should be
15023 used. Unexpected results may occur in processes that do buffering in the user address space.
15024 The process may later read/write data which is/was locked. The *stdio* functions are the most
15025 common source of unexpected buffering.

15026 The *alarm()* function may be used to provide a timeout facility in applications requiring it. |

15027 **FUTURE DIRECTIONS** |

15028 None. |

15029 **SEE ALSO**

15030 *alarm()*, *chmod()*, *close()*, *creat()*, *fcntl()*, *fopen()*, *mmap()*, *open()*, *read()*, *write()*, **<unistd.h>**. |

15031 **CHANGE HISTORY**

15032 First released in Issue 4, Version 2. |

15033 **Issue 5** |

15034 Moved from X/OPEN UNIX extension to BASE. |

15035 Large File Summit extensions added. In particular the description of [EINVAL] is clarified and
15036 moved from optional to mandatory status. |

15037 A note is added to the DESCRIPTION indicating the effects of attempting to lock a section of a
15038 file that is associated with a buffered stream. |

15039 **NAME**15040 locs — stop regular expression matching in a string (**LEGACY**)15041 **SYNOPSIS**

15042 EX #include <regex.h>

15043 extern char *locs;

15044

15045 **DESCRIPTION**15046 Refer to *regex()*.15047 **APPLICATION USAGE**

15048 This variable is kept for historical reasons, but may be withdrawn in a future issue.

15049 New applications should use *fnmatch()*, *glob()*, *regcomp()* and *regex()*, which provide full
 15050 internationalised regular expression functionality compatible with the ISO POSIX-2 standard, as
 15051 described in the **XBD** specification, **Chapter 7, Regular Expressions**.

15052 **CHANGE HISTORY**

15053 First released in Issue 2.

15054 Derived from Issue 2 of the SVID.

15055 **Issue 4**

15056 The following changes are incorporated in this issue:

- 15057 • The <**regex.h**> header is added to the SYNOPSIS section.
- 15058 • The interface is marked TO BE WITHDRAWN, because improved functionality is now
- 15059 provided by interfaces introduced for alignment with the ISO POSIX-2 standard.

15060 **Issue 5**

15061 Marked LEGACY.

15062 **NAME**

15063 log — natural logarithm function

15064 **SYNOPSIS**

15065 #include <math.h>

15066 double log(double x);

15067 **DESCRIPTION**15068 The *log()* function computes the natural logarithm of *x*, $\log_e(x)$. The value of *x* must be positive.15069 An application wishing to check for error situations should set *errno* to 0 before calling *log()*. If
15070 *errno* is non-zero on return, or the return value is NaN, an error has occurred.15071 **RETURN VALUE**15072 Upon successful completion, *log()* returns the natural logarithm of *x*.15073 EX If *x* is NaN, NaN is returned and *errno* may be set to [EDOM].15074 EX If *x* is less than 0, -HUGE_VAL or NaN is returned, and *errno* is set to [EDOM].15075 If *x* is 0, -HUGE_VAL is returned and *errno* may be set to [ERANGE].15076 **ERRORS**15077 The *log()* function will fail if:15078 [EDOM] The value of *x* is negative.15079 The *log()* function may fail if:15080 EX [EDOM] The value of *x* is NaN.15081 [ERANGE] The value of *x* is 0.

15082 EX No other errors will occur.

15083 **EXAMPLES**

15084 None.

15085 **APPLICATION USAGE**

15086 None.

15087 **FUTURE DIRECTIONS**

15088 None.

15089 **SEE ALSO**15090 *exp()*, *isnan()*, *log10()*, *log1p()*, <math.h>.15091 **CHANGE HISTORY**

15092 First released in Issue 1.

15093 Derived from Issue 1 of the SVID.

15094 **Issue 4**

15095 The following changes are incorporated in this issue:

- 15096 • Removed references to *matherr()*.
- 15097 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
- 15098 the ISO C standard and to rationalise error handling in the mathematics functions.
- 15099 • The return value specified for [EDOM] is marked as an extension.

15100 **Issue 5**

15101 The DESCRIPTION is updated to indicate how an application should check for an error. This
15102 text was previously published in the APPLICATION USAGE section.

15103 **NAME**

15104 log10 — base 10 logarithm function

15105 **SYNOPSIS**

15106 #include <math.h>

15107 double log10(double x);

15108 **DESCRIPTION**15109 The *log10()* function computes the base 10 logarithm of *x*, $\log_{10}(x)$. The value of *x* must be
15110 positive.15111 An application wishing to check for error situations should set *errno* to 0 before calling *log10()*.
15112 If *errno* is non-zero on return, or the return value is NaN, an error has occurred.15113 **RETURN VALUE**15114 Upon successful completion, *log10()* returns the base 10 logarithm of *x*.15115 EX If *x* is NaN, NaN is returned and *errno* may be set to [EDOM].15116 EX If *x* is less than 0, -HUGE_VAL or NaN is returned, and *errno* is set to [EDOM].15117 If *x* is 0, -HUGE_VAL is returned and *errno* may be set to [ERANGE].15118 **ERRORS**15119 The *log10()* function will fail if:15120 [EDOM] The value of *x* is negative.15121 The *log10()* function may fail if:15122 EX [EDOM] The value of *x* is NaN.15123 [ERANGE] The value of *x* is 0.

15124 EX No other errors will occur.

15125 **EXAMPLES**

15126 None.

15127 **APPLICATION USAGE**

15128 None.

15129 **FUTURE DIRECTIONS**

15130 None.

15131 **SEE ALSO**15132 *isnan()*, *log()*, *pow()*, <math.h>.15133 **CHANGE HISTORY**

15134 First released in Issue 1.

15135 Derived from Issue 1 of the SVID.

15136 **Issue 4**

15137 The following changes are incorporated in this issue:

- 15138 • Removed references to *matherr()*.
- 15139 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
- 15140 the ISO C standard and to rationalise error handling in the mathematics functions.
- 15141 • The return value specified for [EDOM] is marked as an extension.

15142 **Issue 5**

15143 The DESCRIPTION is updated to indicate how an application should check for an error. This
15144 text was previously published in the APPLICATION USAGE section.

15145 **NAME**

15146 log1p — compute a natural logarithm

15147 **SYNOPSIS**

15148 EX #include <math.h>

15149 double log1p (double x);

15150

15151 **DESCRIPTION**15152 The *log1p()* function computes $\log_e(1.0 + x)$. The value of *x* must be greater than -1.0 .15153 **RETURN VALUE**15154 Upon successful completion, *log1p()* returns the natural logarithm of $1.0 + x$.15155 If *x* is NaN, *log1p()* returns NaN and may set *errno* to [EDOM].15156 If *x* is less than -1.0 , *log1p()* returns $-\text{HUGE_VAL}$ or NaN and sets *errno* to [EDOM].15157 If *x* is -1.0 , *log1p()* returns $-\text{HUGE_VAL}$ and may set *errno* to [ERANGE].15158 **ERRORS**15159 The *log1p()* function will fail if:15160 [EDOM] The value of *x* is less than -1.0 .15161 The *log1p()* function may fail and set *errno* to:15162 [EDOM] The value of *x* is NaN.15163 [ERANGE] The value of *x* is -1.0 .

15164 No other errors will occur.

15165 **EXAMPLES**

15166 None.

15167 **APPLICATION USAGE**

15168 None.

15169 **FUTURE DIRECTIONS**

15170 None.

15171 **SEE ALSO**15172 *log()*, <math.h>.15173 **CHANGE HISTORY**

15174 First released in Issue 4, Version 2.

15175 **Issue 5**

15176 Moved from X/OPEN UNIX extension to BASE.

15177 **NAME**

15178 logb — radix-independent exponent

15179 **SYNOPSIS**

15180 EX #include <math.h>

15181 double logb(double x);

15182

15183 **DESCRIPTION**

15184 The *logb()* function computes the exponent of *x*, which is the integral part of $\log_r |x|$, as a
 15185 signed floating point value, for non-zero *x*, where *r* is the radix of the machine's floating-point
 15186 arithmetic.

15187 **RETURN VALUE**15188 Upon successful completion, *logb()* returns the exponent of *x*.15189 If *x* is 0.0, *logb()* returns `-HUGE_VAL` and sets *errno* to [EDOM].15190 If *x* is $\pm\text{Inf}$, *logb()* returns `+Inf`.15191 If *x* is NaN, *logb()* returns NaN and may set *errno* to [EDOM].15192 **ERRORS**15193 The *logb()* function will fail if:15194 [EDOM] The *x* argument is 0.0.15195 The *logb()* function may fail if:15196 [EDOM] The *x* argument is NaN.15197 **EXAMPLES**

15198 None.

15199 **APPLICATION USAGE**

15200 None.

15201 **FUTURE DIRECTIONS**

15202 None.

15203 **SEE ALSO**15204 *ilogb()*, <math.h>.15205 **CHANGE HISTORY**

15206 First released in Issue 4, Version 2.

15207 **Issue 5**

15208 Moved from X/OPEN UNIX extension to BASE.

15209 NAME

15210 _longjmp, _setjmp — non-local goto

15211 SYNOPSIS

15212 EX #include <setjmp.h>

15213 void _longjmp(jmp_buf env, int val);

15214 int _setjmp(jmp_buf env);

15215

15216 DESCRIPTION

15217 The *_longjmp()* and *_setjmp()* functions are identical to *longjmp()* and *setjmp()*, respectively, with
 15218 the additional restriction that *_longjmp()* and *_setjmp()* do not manipulate the signal mask.

15219 If *_longjmp()* is called even though *env* was never initialised by a call to *_setjmp()*, or when the
 15220 last such call was in a function that has since returned, the results are undefined.

15221 RETURN VALUE

15222 Refer to *longjmp()* and *setjmp()*.

15223 ERRORS

15224 No errors are defined.

15225 EXAMPLES

15226 None.

15227 APPLICATION USAGE

15228 If *_longjmp()* is executed and the environment in which *_setjmp()* was executed no longer exists,
 15229 errors can occur. The conditions under which the environment of the *_setjmp()* no longer exists
 15230 include exiting the function that contains the *_setjmp()* call, and exiting an inner block with
 15231 temporary storage. This condition might not be detectable, in which case the *_longjmp()* occurs
 15232 and, if the environment no longer exists, the contents of the temporary storage of an inner block
 15233 are unpredictable. This condition might also cause unexpected process termination. If the
 15234 function has returned, the results are undefined.

15235 Passing *longjmp()* a pointer to a buffer not created by *setjmp()*, passing *_longjmp()* a pointer to a
 15236 buffer not created by *_setjmp()*, passing *siglongjmp()* a pointer to a buffer not created by
 15237 *sigsetjmp()* or passing any of these three functions a buffer that has been modified by the user
 15238 can cause all the problems listed above, and more.

15239 The *_longjmp()* and *_setjmp()* functions are included to support programs written to historical
 15240 system interfaces. New applications should use *siglongjmp()* and *sigsetjmp()* respectively.

15241 FUTURE DIRECTIONS

15242 None.

15243 SEE ALSO

15244 *longjmp()*, *setjmp()*, *siglongjmp()*, *sigsetjmp()*, <setjmp.h>.

15245 CHANGE HISTORY

15246 First released in Issue 4, Version 2.

15247 Issue 5

15248 Moved from X/OPEN UNIX extension to BASE.

15249 **NAME**

15250 longjmp — non-local goto

15251 **SYNOPSIS**

15252 #include <setjmp.h>

15253 void longjmp(jmp_buf env, int val);

15254 **DESCRIPTION**

15255 The *longjmp()* function restores the environment saved by the most recent invocation of *setjmp()*
 15256 in the same thread, with the corresponding *jmp_buf* argument. If there is no such invocation, or
 15257 if the function containing the invocation of *setjmp()* has terminated execution in the interim, the
 15258 EX behaviour is undefined. It is unspecified whether *longjmp()* restores the signal mask, leaves the
 15259 signal mask unchanged or restores it to its value at the time *setjmp()* was called.

15260 All accessible objects have values as of the time *setjmp()* was called, except that the values of
 15261 objects of automatic storage duration are indeterminate if they meet all the following conditions:

- 15262 • They are local to the function containing the corresponding *setjmp()* invocation.
- 15263 • They do not have volatile-qualified type.
- 15264 • They are changed between the *setjmp()* invocation and *longjmp()* call.

15265 As it bypasses the usual function call and return mechanisms, *longjmp()* will execute correctly in
 15266 contexts of interrupts, signals and any of their associated functions. However, if *longjmp()* is
 15267 invoked from a nested signal handler (that is, from a function invoked as a result of a signal
 15268 raised during the handling of another signal), the behaviour is undefined.

15269 The effect of a call to *longjmp()* where initialisation of the *jmp_buf* structure was not performed
 15270 in the calling thread is undefined.

15271 **RETURN VALUE**

15272 After *longjmp()* is completed, program execution continues as if the corresponding invocation of
 15273 *setjmp()* had just returned the value specified by *val*. The *longjmp()* function cannot cause
 15274 *setjmp()* to return 0; if *val* is 0, *setjmp()* returns 1.

15275 **ERRORS**

15276 No errors are defined.

15277 **EXAMPLES**

15278 None.

15279 **APPLICATION USAGE**

15280 Applications whose behaviour depends on the value of the signal mask should not use
 15281 *longjmp()* and *setjmp()*, since their effect on the signal mask is unspecified, but should instead
 15282 use the *siglongjmp()* and *sigsetjmp()* functions (which can save and restore the signal mask under
 15283 application control).

15284 **FUTURE DIRECTIONS**

15285 None.

15286 **SEE ALSO**15287 *setjmp()*, *sigaction()*, *siglongjmp()*, *sigsetjmp()*, <setjmp.h>.15288 **CHANGE HISTORY**

15289 First released in Issue 1.

15290 Derived from Issue 1 of the SVID.

15291 **Issue 4**

15292 The following change is incorporated for alignment with the ISO C standard:

- 15293 • Mention of volatile-qualified types is added to the DESCRIPTION. |

15294 Another change is incorporated as follows:

- 15295 • The APPLICATION USAGE section is deleted. |

15296 **Issue 4, Version 2**

15297 The DESCRIPTION is updated for X/OPEN UNIX conformance and discusses valid possibilities
15298 for the resulting state of the signal mask. |

15299 **Issue 5**

15300 The DESCRIPTION is updated for alignment with the POSIX Threads Extension. |

15301 **NAME**

15302 lrand48 — generate uniformly distributed pseudo-random non-negative long integers

15303 **SYNOPSIS**

15304 EX #include <stdlib.h>

15305 long int lrand48(void);

15306

15307 **DESCRIPTION**

15308 Refer to *drand48()*.

15309 **CHANGE HISTORY**

15310 First released in Issue 1.

15311 Derived from Issue 1 of the SVID.

15312 **Issue 4**

15313 The following changes are incorporated in this issue:

- 15314 • The <stdlib.h> header is now included in the SYNOPSIS section.
- 15315 • The argument list now contains **void**.

15316 **NAME**

15317 lsearch, lfind — linear search and update

15318 **SYNOPSIS**

15319 EX #include <search.h>

```

15320 void *lsearch(const void *key, void *base, size_t *nel, size_t width,
15321             int (*compar)(const void *, const void *));
15322 void *lfind(const void *key, const void *base, size_t *nel,
15323            size_t width, int (*compar)(const void *, const void *));
15324

```

15325 **DESCRIPTION**

15326 The *lsearch()* function is a linear search routine. It returns a pointer into a table indicating where
 15327 an entry may be found. If the entry does not occur, it is added at the end of the table. The *key*
 15328 argument points to the entry to be sought in the table. The *base* argument points to the first
 15329 element in the table. The *width* argument is the size of an element in bytes. The *nel* argument
 15330 points to an integer containing the current number of elements in the table. The integer to which
 15331 *nel* points is incremented if the entry is added to the table. The *compar* argument points to a
 15332 comparison function which the user must supply (*strcmp()*, for example). It is called with two
 15333 arguments that point to the elements being compared. The function must return 0 if the
 15334 elements are equal and non-zero otherwise.

15335 The *lfind()* function is the same as *lsearch()* except that if the entry is not found, it is not added to
 15336 the table. Instead, a null pointer is returned.

15337 **RETURN VALUE**

15338 If the searched for entry is found, both *lsearch()* and *lfind()* return a pointer to it. Otherwise,
 15339 *lfind()* returns a null pointer and *lsearch()* returns a pointer to the newly added element.

15340 Both functions return a null pointer in case of error.

15341 **ERRORS**

15342 No errors are defined.

15343 **EXAMPLES**

15344 This fragment will read in less than or equal to TABSIZE strings of length less than or equal to
 15345 ELSIZE and store them in a table, eliminating duplicates.

```

15346 #include <stdio.h>
15347 #include <string.h>
15348 #include <search.h>
15349 #define TABSIZE 50
15350 #define ELSIZE 120
15351 ...
15352     char line[ELSIZE], tab[TABSIZE][ELSIZE];
15353     size_t nel = 0;
15354     ...
15355     while (fgets(line, ELSIZE, stdin) != NULL && nel < TABSIZE)
15356         (void) lsearch(line, tab, &nel,
15357                       ELSIZE, (int (*)(const void *, const void *)) strcmp);
15358     ...

```

15359 **APPLICATION USAGE**

15360 The comparison function need not compare every byte, so arbitrary data may be contained in
 15361 the elements in addition to the values being compared.

- 15362 Undefined results can occur if there is not enough room in the table to add a new item. |
- 15363 **FUTURE DIRECTIONS** |
- 15364 None.
- 15365 **SEE ALSO**
- 15366 *bsearch()*, *hsearch()*, *tsearch()*, <**search.h**>.
- 15367 **CHANGE HISTORY**
- 15368 First released in Issue 1.
- 15369 Derived from Issue 1 of the SVID.
- 15370 **Issue 4**
- 15371 The following changes are incorporated in this issue:
- 15372 • In the SYNOPSIS section, the type of argument *key* in the declaration of *lsearch()* is changed |
 - 15373 from **void*** to **const void***, the type arguments *key* and *base* have been changed from **void*** to
 - 15374 **const void*** in the declaration of *lfind()*, and the arguments to *compar()* are defined for both
 - 15375 functions.
 - 15376 • In the EXAMPLES section, the sample code is updated to use ISO C syntax. |
 - 15377 • Warnings about the casting of various arguments are removed from the APPLICATION |
 - 15378 USAGE section, as casting requirements are now clear from the function definitions.

15379 **NAME**

15380 lseek — move the read/write file offset

15381 **SYNOPSIS**

15382 OH #include <sys/types.h>

15383 #include <unistd.h>

15384 off_t lseek(int *fildes*, off_t *offset*, int *whence*);15385 **DESCRIPTION**15386 The *lseek()* function will set the file offset for the open file description associated with the file
15387 descriptor *fildes*, as follows:

- 15388 • If *whence* is SEEK_SET the file offset is set to *offset* bytes.
- 15389 • If *whence* is SEEK_CUR the file offset is set to its current location plus *offset*.
- 15390 • If *whence* is SEEK_END the file offset is set to the size of the file plus *offset*.

15391 The symbolic constants SEEK_SET, SEEK_CUR and SEEK_END are defined in the header
15392 <unistd.h>.15393 The behaviour of *lseek()* on devices which are incapable of seeking is implementation-
15394 dependent. The value of the file offset associated with such a device is undefined.15395 The *lseek()* function will allow the file offset to be set beyond the end of the existing data in the
15396 file. If data is later written at this point, subsequent reads of data in the gap will return bytes
15397 with the value 0 until data is actually written into the gap.15398 The *lseek()* function will not, by itself, extend the size of a file.15399 RT If *fildes* refers to a shared memory object, the result of the *lseek()* function is unspecified.15400 **RETURN VALUE**15401 Upon successful completion, the resulting offset, as measured in bytes from the beginning of the
15402 file, is returned. Otherwise, (off_t)−1 is returned, *errno* is set to indicate the error and the file
15403 offset will remain unchanged.15404 **ERRORS**15405 The *lseek()* function will fail if:

- 15406 [EBADF] The *fildes* argument is not an open file descriptor.
- 15407 [EINVAL] The *whence* argument is not a proper value, or the resulting file offset would
15408 be invalid.
- 15409 EX [EOVERFLOW] The resulting file offset would be a value which cannot be represented
15410 correctly in an object of type **off_t**.
- 15411 [ESPIPE] The *fildes* argument is associated with a pipe or FIFO.

15412 **EXAMPLES**

15413 None.

15414 **APPLICATION USAGE**

15415 None.

15416 **FUTURE DIRECTIONS**

15417 None.

15418 **SEE ALSO**15419 *open()*, <sys/types.h>, <unistd.h>.

15420 **CHANGE HISTORY**

15421 First released in Issue 1.

15422 Derived from Issue 1 of the SVID.

15423 **Issue 4**

15424 The following changes are incorporated in this issue:

15425 • The `<sys/types.h>` header is now marked as optional (OH); this header need not be included |
15426 on XSI-conformant systems.15427 • The APPLICATION USAGE section is removed, as the ISO POSIX-1 standard now requires |
15428 that `off_t` be signed.15429 **Issue 5**

15430 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension. |

15431 Large File Summit extensions added.

15432 **NAME**

15433 lstat — get symbolic link status

15434 **SYNOPSIS**

15435 EX #include <sys/stat.h>

15436 int lstat(const char *path, struct stat *buf);

15437

15438 **DESCRIPTION**

15439 The *lstat()* function has the same effect as *stat()*, except when *path* refers to a symbolic link. In
 15440 that case *lstat()* returns information about the link, while *stat()* returns information about the
 15441 file the link references.

15442 For symbolic links, the **st_mode** member will contain meaningful information when used with
 15443 the file type macros, and the **st_size** member will contain the length of the pathname contained
 15444 in the symbolic link. File mode bits and the contents of the remaining members of the *stat*
 15445 structure are unspecified. The value returned in the **st_size** member is the length of the contents
 15446 of the symbolic link, and does not count any trailing null.

15447 **RETURN VALUE**

15448 Upon successful completion, *lstat()* returns 0. Otherwise, it returns -1 and sets *errno* to indicate
 15449 the error.

15450 **ERRORS**15451 The *lstat()* function will fail if:

15452 [EACCES] A component of the path prefix denies search permission.

15453 [EIO] An error occurred while reading from the file system.

15454 [ELOOP] Too many symbolic links were encountered in resolving *path*.

15455 [ENAMETOOLONG]

15456 The length of a pathname exceeds {PATH_MAX}, or pathname component is
 15457 longer than {NAME_MAX}.

15458 [ENOTDIR] A component of the path prefix is not a directory.

15459 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

15460 EX [E_OVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file
 15461 serial number cannot be represented correctly in the structure pointed to by
 15462 *buf*.

15463 The *lstat()* function may fail if:

15464 [ENAMETOOLONG]

15465 Pathname resolution of a symbolic link produced an intermediate result
 15466 whose length exceeds {PATH_MAX}.

15467 [E_OVERFLOW] One of the members is too large to store into the structure pointed to by the
 15468 *buf* argument.

15469 **EXAMPLES**

15470 None.

15471 **APPLICATION USAGE**

15472 None.

15473 **FUTURE DIRECTIONS**

15474 None.

15475 **SEE ALSO**15476 *fstat()*, *readlink()*, *stat()*, *symlink()*, <sys/stat.h>.15477 **CHANGE HISTORY**

15478 First released in Issue 4, Version 2.

15479 **Issue 5**

15480 Moved from X/OPEN UNIX extension to BASE.

15481 Large File Summit extensions added.

15482 **NAME**

15483 makecontext, swapcontext — manipulate user contexts

15484 **SYNOPSIS**

15485 EX #include <ucontext.h>

15486 void makecontext(ucontext_t *ucp, (void *func)(), int argc, ...);

15487 int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);

15488

15489 **DESCRIPTION**

15490 The *makecontext()* function modifies the context specified by *ucp*, which has been initialised
 15491 using *getcontext()*. When this context is resumed using *swapcontext()* or *setcontext()*, program
 15492 execution continues by calling *func*, passing it the arguments that follow *argc* in the
 15493 *makecontext()* call.

15494 Before a call is made to *makecontext()*, the context being modified should have a stack allocated
 15495 for it. The value of *argc* must match the number of integer arguments passed to *func*, otherwise
 15496 the behaviour is undefined.

15497 The *uc_link* member is used to determine the context that will be resumed when the context
 15498 being modified by *makecontext()* returns. The *uc_link* member should be initialised prior to the
 15499 call to *makecontext()*.

15500 The *swapcontext()* function saves the current context in the context structure pointed to by *oucp*
 15501 and sets the context to the context structure pointed to by *ucp*.

15502 **RETURN VALUE**

15503 On successful completion, *swapcontext()* returns 0. Otherwise, -1 is returned and *errno* is set to
 15504 indicate the error.

15505 **ERRORS**15506 The *swapcontext()* function will fail if:

15507 [ENOMEM] The *ucp* argument does not have enough stack left to complete the operation.

15508 **EXAMPLES**

15509 None.

15510 **APPLICATION USAGE**

15511 None.

15512 **FUTURE DIRECTIONS**

15513 None.

15514 **SEE ALSO**15515 *exit()*, *getcontext()*, *sigaction()*, *sigprocmask()*, <ucontext.h>.15516 **CHANGE HISTORY**

15517 First released in Issue 4, Version 2.

15518 **Issue 5**

15519 Moved from X/OPEN UNIX extension to BASE.

15520 In the ERRORS section, the description of [ENOMEM] is changed to apply to *swapcontext()* only.

15521 NAME

15522 **malloc** — a memory allocator

15523 SYNOPSIS

15524 `#include <stdlib.h>`

15525 `void *malloc(size_t size);`

15526 DESCRIPTION

15527 The *malloc()* function allocates unused space for an object whose size in bytes is specified by *size*
15528 and whose value is indeterminate.

15529 The order and contiguity of storage allocated by successive calls to *malloc()* is unspecified. The
15530 pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a
15531 pointer to any type of object and then used to access such an object in the space allocated (until
15532 the space is explicitly freed or reallocated). Each such allocation will yield a pointer to an object
15533 disjoint from any other object. The pointer returned points to the start (lowest byte address) of
15534 the allocated space. If the space cannot be allocated, a null pointer is returned. If the size of the
15535 space requested is 0, the behaviour is implementation-dependent; the value returned will be
15536 either a null pointer or a unique pointer.

15537 RETURN VALUE

15538 Upon successful completion with *size* not equal to 0, *malloc()* returns a pointer to the allocated
15539 space. If *size* is 0, either a null pointer or a unique pointer that can be successfully passed to
15540 EX *free()* will be returned. Otherwise, it returns a null pointer and sets *errno* to indicate the error.

15541 ERRORS

15542 The *malloc()* function will fail if:

15543 EX [ENOMEM] Insufficient storage space is available.

15544 EXAMPLES

15545 None.

15546 APPLICATION USAGE

15547 None.

15548 FUTURE DIRECTIONS

15549 None.

15550 SEE ALSO

15551 *calloc()*, *free()*, *realloc()*, *<stdlib.h>*.

15552 CHANGE HISTORY

15553 First released in Issue 1.

15554 Derived from Issue 1 of the SVID.

15555 Issue 4

15556 The following change is incorporated for alignment with the ISO C standard:

- 15557 • The RETURN VALUE section is updated to indicate what will be returned if *size* is 0.

15558 Other changes are incorporated as follows:

- 15559 • The setting of *errno* and the [ENOMEM] error are marked as extensions.
- 15560 • The APPLICATION USAGE section is changed to record that *<malloc.h>* need no longer be
- 15561 supported on XSI-conformant systems.

15562 **NAME**

15563 mblen — get number of bytes in a character

15564 **SYNOPSIS**

15565 #include <stdlib.h>

15566 int mblen(const char *s, size_t n);

15567 **DESCRIPTION**

15568 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:

15570 mbtowc((wchar_t *)0, s, n);

15571 The implementation will behave as if no function defined in this document calls *mblen()*.

15572 The behaviour 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.

15580 **RETURN VALUE**

15581 If *s* is a null pointer, *mblen()* returns 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()* either returns 0 (if *s* points to the null byte), or returns the number of bytes that constitute the character (if the next *n* or fewer bytes form a valid character), or returns -1 (if they do not form a valid character) and may set *errno* to indicate the error. In no case will the value returned be greater than *n* or the value of the MB_CUR_MAX macro.

15587 **ERRORS**15588 The *mblen()* function may fail if:

15589 EX [EILSEQ] Invalid character sequence is detected.

15590 **EXAMPLES**

15591 None.

15592 **APPLICATION USAGE**

15593 None.

15594 **FUTURE DIRECTIONS**

15595 None.

15596 **SEE ALSO**15597 *mbtowc()*, *mbstowcs()*, *wctomb()*, *wcstombs()*, <stdlib.h>.15598 **CHANGE HISTORY**

15599 First released in Issue 4.

15600 Aligned with the ISO C standard.

15601 NAME

15602 mbrlen — get number of bytes in a character (restartable)

15603 SYNOPSIS

15604 #include <wchar.h>

15605 size_t mbrlen(const char *s, size_t n, mbstate_t *ps);

15606 DESCRIPTION

15607 If *s* is not a null pointer, *mbrlen()* determines the number of bytes constituting the character pointed to by *s*. It is equivalent to:

15609 mbstate_t internal;

15610 mbrtowc(NULL, s, n, ps != NULL ? ps : &internal);

15611 If *ps* is a null pointer, the *mbrlen()* function uses its own internal **mbstate_t** object, which is initialised at program startup 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 will behave as if no function defined in this specification calls *mbrlen()*.

15616 The behaviour of this function is affected by the LC_CTYPE category of the current locale.

15617 RETURN VALUE

15618 The *mbrlen()* function returns the first of the following that applies:

15619 0 If the next *n* or fewer bytes complete the character that corresponds to the null wide-character.

15621 *positive* If the next *n* or fewer bytes complete a valid character; the value returned is the number of bytes that complete the character.

15623 (**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).

15628 (**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 is stored in *errno* and the conversion state is undefined.

15631 ERRORS

15632 The *mbrlen()* function may fail if:

15633 [EINVAL] *ps* points to an object that contains an invalid conversion state.

15634 [EILSEQ] Invalid character sequence is detected.

15635 EXAMPLES

15636 None.

15637 APPLICATION USAGE

15638 None.

15639 FUTURE DIRECTIONS

15640 None.

15641 SEE ALSO

15642 *mbsinit()*, *mbrtowc()*, <wchar.h>.

15643	CHANGE HISTORY	
15644	First released in Issue 5.	
15645	Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).	

15646 **NAME**

15647 mbrtowc — convert a character to a wide-character code (restartable)

15648 **SYNOPSIS**

15649 #include <wchar.h>

15650 size_t mbrtowc(wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);

15651 **DESCRIPTION**15652 If *s* is a null pointer, the *mbrtowc()* function is equivalent to the call:

15653 mbrtowc(NULL, '', 1, ps)

15654 In this case, the values of the arguments *pwc* and *n* are ignored.

15655 If *s* is not a null pointer, the *mbrtowc()* function inspects at most *n* bytes beginning at the byte pointed to by *s* to determine the number of bytes needed to complete the next character (including any shift sequences). If the function determines that the next character is completed, it determines the value of the corresponding wide-character and then, if *pwc* is not a null pointer, stores that value in the object pointed to by *pwc*. If the corresponding wide-character is the null wide-character, the resulting state described is the initial conversion state.

15661 If *ps* is a null pointer, the *mbrtowc()* function uses its own internal **mbstate_t** object, which is initialised at program startup 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 will behave as if no function defined in this specification calls *mbrtowc()*.

15666 The behaviour of this function is affected by the LC_CTYPE category of the current locale.

15667 **RETURN VALUE**15668 The *mbrtowc()* function returns the first of the following that applies:

15669 0 If the next *n* or fewer bytes complete the character that corresponds to the null
15670 wide-character (which is the value stored).

15671 *positive* If the next *n* or fewer bytes complete a valid character (which is the value
15672 stored); the value returned is the number of bytes that complete the character.

15673 (**size_t**)-2 If the next *n* bytes contribute to an incomplete but potentially valid character,
15674 and all *n* bytes have been processed (no value is stored). When *n* has at least
15675 the value of the MB_CUR_MAX macro, this case can only occur if *s* points at a
15676 sequence of redundant shift sequences (for implementations with state-
15677 dependent encodings).

15678 (**size_t**)-1 If an encoding error occurs, in which case the next *n* or fewer bytes do not
15679 contribute to a complete and valid character (no value is stored). In this case,
15680 EILSEQ is stored in *errno* and the conversion state is undefined.

15681 **ERRORS**15682 The *mbrtowc()* function may fail if:15683 [EINVAL] *ps* points to an object that contains an invalid conversion state.

15684 [EILSEQ] Invalid character sequence is detected.

15685 **EXAMPLES**

15686 None.

15687 **APPLICATION USAGE**

15688 None.

15689 **FUTURE DIRECTIONS**

15690 None.

15691 **SEE ALSO**15692 *mbsinit()*, <wchar.h>.15693 **CHANGE HISTORY**

15694 First released in Issue 5.

15695 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

15696 **NAME**

15697 mbsinit — determine conversion object status

15698 **SYNOPSIS**

15699 #include <wchar.h>

15700 int mbsinit(const mbstate_t *ps);

15701 **DESCRIPTION**

15702 If *ps* is not a null pointer, the *mbsinit()* function determines whether the object pointed to by *ps*
15703 describes an initial conversion state.

15704 **RETURN VALUE**

15705 The *mbsinit()* function returns non-zero if *ps* is a null pointer, or if the pointed-to object describes
15706 an initial conversion state; otherwise, it returns zero.

15707 If an **mbstate_t** object is altered by any of the functions described as "restartable", and is then
15708 used with a different character sequence, or in the other conversion direction, or with a different
15709 LC_CTYPE category setting than on earlier function calls, the behaviour is undefined.

15710 **ERRORS**

15711 No errors are defined.

15712 **EXAMPLES**

15713 None.

15714 **APPLICATION USAGE**

15715 The **mbstate_t** object is used to describe the current conversion state from a particular character
15716 sequence to a wide-character sequence (or vice versa) under the rules of a particular setting of
15717 the LC_CTYPE category of the current locale.

15718 The initial conversion state corresponds, for a conversion in either direction, to the beginning of
15719 a new character sequence in the initial shift state. A zero valued **mbstate_t** object is at least one
15720 way to describe an initial conversion state. A zero valued **mbstate_t** object can be used to
15721 initiate conversion involving any character sequence, in any LC_CTYPE category setting.

15722 **FUTURE DIRECTIONS**

15723 None.

15724 **SEE ALSO**

15725 **mbrlen()**, **mbrtowc()**, **wcrtomb()**, **mbsrtowcs()**, **wcsrtombs()**, **<wchar.h>**.

15726 **CHANGE HISTORY**

15727 First released in Issue 5.

15728 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

15729 **NAME**15730 `mbsrtowcs` — convert a character string to a wide-character string (restartable)15731 **SYNOPSIS**15732 `#include <wchar.h>`15733 `size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len,`
15734 `mbstate_t *ps);`15735 **DESCRIPTION**

15736 The `mbsrtowcs()` function converts a sequence of characters, beginning in the conversion state
 15737 described by the object pointed to by `ps`, from the array indirectly pointed to by `src` into a
 15738 sequence of corresponding wide-characters. If `dst` is not a null pointer, the converted characters
 15739 are stored into the array pointed to by `dst`. Conversion continues up to and including a
 15740 terminating null character, which is also stored. Conversion stops early in either of the
 15741 following cases:

- 15742 • When a sequence of bytes is encountered that does not form a valid character.
- 15743 • When `len` codes have been stored into the array pointed to by `dst` (and `dst` is not a null
 15744 pointer).

15745 Each conversion takes place as if by a call to the `mbrtowc()` function.

15746 If `dst` is not a null pointer, the pointer object pointed to by `src` is assigned either a null pointer (if
 15747 conversion stopped due to reaching a terminating null character) or the address just past the last
 15748 character converted (if any). If conversion stopped due to reaching a terminating null character,
 15749 and if `dst` is not a null pointer, the resulting state described is the initial conversion state.

15750 If `ps` is a null pointer, the `mbsrtowcs()` function uses its own internal `mbstate_t` object, which is
 15751 initialised at program startup to the initial conversion state. Otherwise, the `mbstate_t` object
 15752 pointed to by `ps` is used to completely describe the current conversion state of the associated
 15753 character sequence. The implementation will behave as if no function defined in this
 15754 specification calls `mbsrtowcs()`.

15755 The behaviour of this function is affected by the `LC_CTYPE` category of the current locale.15756 **RETURN VALUE**

15757 If the input conversion encounters a sequence of bytes that do not form a valid character, an
 15758 encoding error occurs. In this case, the `mbsrtowcs()` function stores the value of the macro
 15759 `EILSEQ` in `errno` and returns `(size_t)-1`; the conversion state is undefined. Otherwise, it returns
 15760 the number of characters successfully converted, not including the terminating null (if any).

15761 **ERRORS**15762 The `mbsrtowcs()` function may fail if:

- 15763 `[EINVAL]` `ps` points to an object that contains an invalid conversion state.
- 15764 `[EILSEQ]` Invalid character sequence is detected.

15765 **EXAMPLES**

15766 None.

15767 **APPLICATION USAGE**

15768 None.

15769 **FUTURE DIRECTIONS**

15770 None.

15771 **SEE ALSO**

15772 *mbsinit()*, *mbrtowc()*, **<wchar.h>**.

15773 **CHANGE HISTORY**

15774 First released in Issue 5.

15775 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

15776 **NAME**

15777 mbstowcs — convert a character string to a wide-character string

15778 **SYNOPSIS**

15779 #include <stdlib.h>

15780 size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);

15781 **DESCRIPTION**

15782 The *mbstowcs()* function converts a sequence of characters that begins in the initial shift state
 15783 from the array pointed to by *s* into a sequence of corresponding wide-character codes and stores
 15784 not more than *n* wide-character codes into the array pointed to by *pwcs*. No characters that
 15785 follow a null byte (which is converted into a wide-character code with value 0) will be examined
 15786 or converted. Each character is converted as if by a call to *mbtowc()*, except that the shift state of
 15787 *mbtowc()* is not affected.

15788 No more than *n* elements will be modified in the array pointed to by *pwcs*. If copying takes
 15789 place between objects that overlap, the behaviour is undefined.

15790 EX The behaviour of this function is affected by the LC_CTYPE category of the current locale. If
 15791 *pwcs* is a null pointer, *mbstowcs()* returns the length required to convert the entire array
 15792 regardless of the value of *n*, but no values are stored.

15793 **RETURN VALUE**

15794 If an invalid character is encountered, *mbstowcs()* returns (**size_t**)−1 and may set *errno* to indicate
 15795 the error. Otherwise, *mbstowcs()* returns the number of the array elements modified (or required
 15796 if *pwcs* is null), not including a terminating 0 code, if any. The array will not be zero-terminated
 15797 if the value returned is *n*.

15798 **ERRORS**15799 The *mbstowcs()* function may fail if:

15800 EX [EILSEQ] Invalid byte sequence is detected.

15801 **EXAMPLES**

15802 None.

15803 **APPLICATION USAGE**

15804 None.

15805 **FUTURE DIRECTIONS**

15806 None.

15807 **SEE ALSO**15808 *mblen()*, *mbtowc()*, *wctomb()*, *wcstombs()*, <stdlib.h>.15809 **CHANGE HISTORY**

15810 First released in Issue 4.

15811 Aligned with the ISO C standard.

15812 **NAME**

15813 mbtowc — convert a character to a wide-character code

15814 **SYNOPSIS**

15815 #include <stdlib.h>

15816 int mbtowc(wchar_t *pwc, const char *s, size_t n);

15817 **DESCRIPTION**

15818 If *s* is not a null pointer, *mbtowc()* determines the number of the bytes that constitute the
 15819 character pointed to by *s*. It then determines the wide-character code for the value of type
 15820 **wchar_t** that corresponds to that character. (The value of the wide-character code
 15821 corresponding to the null byte is 0.) If the character is valid and *pwc* is not a null pointer,
 15822 *mbtowc()* stores the wide-character code in the object pointed to by *pwc*.

15823 The behaviour of this function is affected by the LC_CTYPE category of the current locale. For a
 15824 state-dependent encoding, this function is placed into its initial state by a call for which its
 15825 character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null
 15826 pointer cause the internal state of the function to be altered as necessary. A call with *s* as a null
 15827 pointer causes this function to return a non-zero value if encodings have state dependency, and
 15828 0 otherwise. If the implementation employs special bytes to change the shift state, these bytes
 15829 do not produce separate wide-character codes, but are grouped with an adjacent character.
 15830 Changing the LC_CTYPE category causes the shift state of this function to be indeterminate. At
 15831 most *n* bytes of the array pointed to by *s* will be examined.

15832 The implementation will behave as if no function defined in this specification calls *mbtowc()*.15833 **RETURN VALUE**

15834 If *s* is a null pointer, *mbtowc()* returns a non-zero or 0 value, if character encodings, respectively,
 15835 do or do not have state-dependent encodings. If *s* is not a null pointer, *mbtowc()* either returns 0
 15836 (if *s* points to the null byte), or returns the number of bytes that constitute the converted
 15837 character (if the next *n* or fewer bytes form a valid character), or returns -1 and may set *errno* to
 15838 indicate the error (if they do not form a valid character).

15839 In no case will the value returned be greater than *n* or the value of the MB_CUR_MAX macro.15840 **ERRORS**15841 The *mbtowc()* function may fail if:

15842 EX [EILSEQ] Invalid character sequence is detected.

15843 **EXAMPLES**

15844 None.

15845 **APPLICATION USAGE**

15846 None.

15847 **FUTURE DIRECTIONS**

15848 None.

15849 **SEE ALSO**15850 *mblen()*, *mbstowcs()*, *wctomb()*, *wcstombs()*, <stdlib.h>.15851 **CHANGE HISTORY**

15852 First released in Issue 4.

15853 Aligned with the ISO C standard.

15854 **NAME**

15855 memccpy — copy bytes in memory

15856 **SYNOPSIS**15857 EX `#include <string.h>`15858 `void *memccpy(void *s1, const void *s2, int c, size_t n);`

15859

15860 **DESCRIPTION**

15861 The *memccpy()* function copies bytes from memory area *s2* into *s1*, stopping after the first
 15862 occurrence of byte *c* (converted to an **unsigned char**) is copied, or after *n* bytes are copied,
 15863 whichever comes first. If copying takes place between objects that overlap, the behaviour is
 15864 undefined.

15865 **RETURN VALUE**

15866 The *memccpy()* function returns a pointer to the byte after the copy of *c* in *s1*, or a null pointer if *c*
 15867 was not found in the first *n* bytes of *s2*.

15868 **ERRORS**

15869 No errors are defined.

15870 **EXAMPLES**

15871 None.

15872 **APPLICATION USAGE**15873 The *memccpy()* function does not check for the overflow of the receiving memory area.15874 **FUTURE DIRECTIONS**

15875 None.

15876 **SEE ALSO**15877 `<string.h>`.15878 **CHANGE HISTORY**

15879 First released in Issue 1.

15880 Derived from Issue 1 of the SVID.

15881 **Issue 4**

15882 The following changes are incorporated in this issue:

- 15883 • The type of argument *s2* is changed from **void*** to **const void***.
- 15884 • Reference to use of the `<memory.h>` header is removed from the APPLICATION USAGE
- 15885 section.
- 15886 • The FUTURE DIRECTIONS section is removed.

15887 **NAME**

15888 memchr — find byte in memory

15889 **SYNOPSIS**

15890 #include <string.h>

15891 void *memchr(const void *s, int c, size_t n);

15892 **DESCRIPTION**

15893 The *memchr()* function locates the first occurrence of *c* (converted to an **unsigned char**) in the
15894 initial *n* bytes (each interpreted as **unsigned char**) of the object pointed to by *s*.

15895 **RETURN VALUE**

15896 The *memchr()* function returns a pointer to the located byte, or a null pointer if the byte does not
15897 occur in the object.

15898 **ERRORS**

15899 No errors are defined.

15900 **EXAMPLES**

15901 None.

15902 **APPLICATION USAGE**

15903 None.

15904 **FUTURE DIRECTIONS**

15905 None.

15906 **SEE ALSO**

15907 <string.h>.

15908 **CHANGE HISTORY**

15909 First released in Issue 1.

15910 Derived from Issue 1 of the SVID.

15911 **Issue 4**

15912 The following changes are incorporated for alignment with the ISO C standard:

- 15913 • The function is no longer marked as an extension.
- 15914 • The type of argument *s* is changed from **void*** to **const void***.

15915 Another change is incorporated as follows:

- 15916 • The APPLICATION USAGE section is removed.

15917 **NAME**

15918 memcmp — compare bytes in memory

15919 **SYNOPSIS**

15920 #include <string.h>

15921 int memcmp(const void *s1, const void *s2, size_t n);

15922 **DESCRIPTION**15923 The *memcmp()* function compares the first *n* bytes (each interpreted as **unsigned char**) of the
15924 object pointed to by *s1* to the first *n* bytes of the object pointed to by *s2*.15925 The sign of a non-zero return value is determined by the sign of the difference between the
15926 values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the objects
15927 being compared.15928 **RETURN VALUE**15929 The *memcmp()* function returns an integer greater than, equal to or less than 0, if the object
15930 pointed to by *s1* is greater than, equal to or less than the object pointed to by *s2* respectively.15931 **ERRORS**

15932 No errors are defined.

15933 **EXAMPLES**

15934 None.

15935 **APPLICATION USAGE**

15936 None.

15937 **FUTURE DIRECTIONS**

15938 None.

15939 **SEE ALSO**

15940 <string.h>.

15941 **CHANGE HISTORY**

15942 First released in Issue 1.

15943 Derived from Issue 1 of the SVID.

15944 **Issue 4**

15945 The following changes are incorporated for alignment with the ISO C standard:

- 15946
- The function is no longer marked as an extension.
 - The type of arguments *s1* and *s2* are changed from **void*** to **const void***.

15948 Other changes are incorporated as follows:

- 15949
- The RETURN VALUE section is clarified.
 - The APPLICATION USAGE section is removed.

15951 **NAME**

15952 memcpy — copy bytes in memory

15953 **SYNOPSIS**

15954 #include <string.h>

15955 void *memcpy(void *s1, const void *s2, size_t n);

15956 **DESCRIPTION**15957 The *memcpy()* function copies *n* bytes from the object pointed to by *s2* into the object pointed to
15958 by *s1*. If copying takes place between objects that overlap, the behaviour is undefined.15959 **RETURN VALUE**15960 The *memcpy()* function returns *s1*; no return value is reserved to indicate an error.15961 **ERRORS**

15962 No errors are defined.

15963 **EXAMPLES**

15964 None.

15965 **APPLICATION USAGE**15966 The *memcpy()* function does not check for the overflowing of the receiving memory area.15967 **FUTURE DIRECTIONS**

15968 None.

15969 **SEE ALSO**

15970 <string.h>.

15971 **CHANGE HISTORY**

15972 First released in Issue 1.

15973 Derived from Issue 1 of the SVID.

15974 **Issue 4**

15975 The following changes are incorporated for alignment with the ISO C standard:

- 15976
- The function is no longer marked as an extension.
 - The type of argument *s2* is changed from **void*** to **const void***.

15978 Other changes are incorporated as follows:

- 15979
- Reference to use of the <**memory.h**> header is removed from the APPLICATION USAGE
-
- 15980 section, and a note about overflow checking has been added.
-
- 15981
- The FUTURE DIRECTIONS section is removed.

15982 **NAME**

15983 memmove — copy bytes in memory with overlapping areas

15984 **SYNOPSIS**

15985 #include <string.h>

15986 void *memmove(void *s1, const void *s2, size_t n);

15987 **DESCRIPTION**

15988 The *memmove()* function copies *n* bytes from the object pointed to by *s2* into the object pointed to
15989 by *s1*. Copying takes place as if the *n* bytes from the object pointed to by *s2* are first copied into
15990 a temporary array of *n* bytes that does not overlap the objects pointed to by *s1* and *s2*, and then
15991 the *n* bytes from the temporary array are copied into the object pointed to by *s1*.

15992 **RETURN VALUE**15993 The *memmove()* function returns *s1*; no return value is reserved to indicate an error.15994 **ERRORS**

15995 No errors are defined.

15996 **EXAMPLES**

15997 None.

15998 **APPLICATION USAGE**

15999 None.

16000 **SEE ALSO**

16001 <string.h>.

16002 **CHANGE HISTORY**

16003 First released in Issue 4.

16004 Derived from the ANSI C standard.

16005 **NAME**

16006 memset — set bytes in memory

16007 **SYNOPSIS**

16008 #include <string.h>

16009 void *memset(void *s, int c, size_t n);

16010 **DESCRIPTION**

16011 The *memset()* function copies *c* (converted to an **unsigned char**) into each of the first *n* bytes of
16012 the object pointed to by *s*.

16013 **RETURN VALUE**16014 The *memset()* function returns *s*; no return value is reserved to indicate an error.16015 **ERRORS**

16016 No errors are defined.

16017 **EXAMPLES**

16018 None.

16019 **APPLICATION USAGE**

16020 None.

16021 **SEE ALSO**

16022 <string.h>.

16023 **CHANGE HISTORY**

16024 First released in Issue 1.

16025 Derived from Issue 1 of the SVID.

16026 **Issue 4**

16027 The following change is incorporated for alignment with the ISO C standard:

- 16028 • The function is no longer marked as an extension.

16029 Another change is incorporated as follows:

- 16030 • The APPLICATION USAGE section is removed.

16031 **NAME**

16032 mkdir — make a directory

16033 **SYNOPSIS**

16034 OH #include <sys/types.h>

16035 #include <sys/stat.h>

16036 int mkdir(const char *path, mode_t mode);

16037 **DESCRIPTION**

16038 The *mkdir()* function creates a new directory with name *path*. The file permission bits of the new
 16039 directory are initialised from *mode*. These file permission bits of the *mode* argument are modified
 16040 by the process' file creation mask.

16041 When bits in *mode* other than the file permission bits are set, the meaning of these additional bits
 16042 is implementation-dependent.

16043 The directory's user ID is set to the process' effective user ID. The directory's group ID is set to
 16044 the group ID of the parent directory or to the effective group ID of the process.

16045 The newly created directory will be an empty directory.

16046 Upon successful completion, *mkdir()* will mark for update the *st_atime*, *st_ctime* and *st_mtime*
 16047 fields of the directory. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the
 16048 new entry are marked for update.

16049 **RETURN VALUE**

16050 Upon successful completion, *mkdir()* returns 0. Otherwise, -1 is returned, no directory is created
 16051 and *errno* is set to indicate the error.

16052 **ERRORS**

16053 The *mkdir()* function will fail if:

16054 [EACCES] Search permission is denied on a component of the path prefix, or write
 16055 permission is denied on the parent directory of the directory to be created.

16056 [EEXIST] The named file exists.

16057 EX [ELOOP] Too many symbolic links were encountered in resolving *path*.

16058 [EMLINK] The link count of the parent directory would exceed {LINK_MAX}.

16059 FIPS [ENAMETOOLONG]

16060 The length of the *path* argument exceeds {PATH_MAX} or a pathname
 16061 component is longer than {NAME_MAX}.

16062 [ENOENT] A component of the path prefix specified by *path* does not name an existing
 16063 directory or *path* is an empty string.

16064 [ENOSPC] The file system does not contain enough space to hold the contents of the new
 16065 directory or to extend the parent directory of the new directory.

16066 [ENOTDIR] A component of the path prefix is not a directory.

16067 [EROFS] The parent directory resides on a read-only file system.

16068 The *mkdir()* function may fail if:

16069 EX [ENAMETOOLONG]

16070 Pathname resolution of a symbolic link produced an intermediate result
 16071 whose length exceeds {PATH_MAX}.

16072 **EXAMPLES**

16073 None.

16074 **APPLICATION USAGE**

16075 None.

16076 **SEE ALSO**

16077 *umask()*, *<sys/stat.h>*, *<sys/types.h>*.

16078 **CHANGE HISTORY**

16079 First released in Issue 3.

16080 Entry included for alignment with the POSIX.1-1988 standard.

16081 **Issue 4**

16082 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 16083 • The type of argument *path* is changed from **char *** to **const char ***.

16084 The following changes are incorporated for alignment with the FIPS requirements:

- 16085 • In the **ERRORS** section, the condition whereby **[ENAMETOOLONG]** will be returned if a
16086 pathname component is larger than **{NAME_MAX}** is now defined as mandatory and
16087 marked as an extension.

16088 Another change is incorporated as follows:

- 16089 • The *<sys/types.h>* header is now marked as optional (OH); this header need not be included
16090 on XSI-conformant systems.

16091 **Issue 4, Version 2**

16092 The **ERRORS** section is updated for X/OPEN UNIX conformance as follows:

- 16093 • It states that **[ELOOP]** will be returned if too many symbolic links are encountered during
16094 pathname resolution.
- 16095 • A second **[ENAMETOOLONG]** condition is defined that may report excessive length of an
16096 intermediate result of pathname resolution of a symbolic link.

16097 **NAME**

16098 mkfifo — make a FIFO special file

16099 **SYNOPSIS**

16100 OH #include <sys/types.h>

16101 #include <sys/stat.h>

16102 int mkfifo(const char *path, mode_t mode);

16103 **DESCRIPTION**

16104 The *mkfifo()* function creates a new FIFO special file named by the pathname pointed to by *path*.
 16105 The file permission bits of the new FIFO are initialised from *mode*. The file permission bits of the
 16106 *mode* argument are modified by the process' file creation mask.

16107 When bits in *mode* other than the file permission bits are set, the effect is implementation-
 16108 dependent.

16109 The FIFO's user ID will be set to the process' effective user ID. The FIFO's group ID will be set
 16110 to the group ID of the parent directory or to the effective group ID of the process.

16111 Upon successful completion, *mkfifo()* will mark for update the *st_atime*, *st_ctime* and *st_mtime*
 16112 fields of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new
 16113 entry are marked for update.

16114 **RETURN VALUE**

16115 Upon successful completion, 0 is returned. Otherwise, -1 is returned, no FIFO is created and
 16116 *errno* is set to indicate the error.

16117 **ERRORS**16118 The *mkfifo()* function will fail if:

16119 [EACCES] A component of the path prefix denies search permission, or write permission
 16120 is denied on the parent directory of the FIFO to be created.

16121 [EEXIST] The named file already exists.

16122 EX [ELOOP] Too many symbolic links were encountered in resolving *path*.

16123 FIPS [ENAMETOOLONG]

16124 The length of the *path* argument exceeds {PATH_MAX} or a pathname
 16125 component is longer than {NAME_MAX}.

16126 [ENOENT] A component of the path prefix specified by *path* does not name an existing
 16127 directory or *path* is an empty string.

16128 [ENOSPC] The directory that would contain the new file cannot be extended or the file
 16129 system is out of file-allocation resources.

16130 [ENOTDIR] A component of the path prefix is not a directory.

16131 [EROFS] The named file resides on a read-only file system.

16132 The *mkfifo()* function may fail if:

16133 EX [ENAMETOOLONG]

16134 Pathname resolution of a symbolic link produced an intermediate result
 16135 whose length exceeds {PATH_MAX}.

16136 **EXAMPLES**

16137 None.

16138 **APPLICATION USAGE**

16139 None.

16140 **SEE ALSO**16141 `umask()`, `<sys/stat.h>`, `<sys/types.h>`.16142 **CHANGE HISTORY**

16143 First released in Issue 3.

16144 Entry included for alignment with the POSIX.1-1988 standard.

16145 **Issue 4**

16146 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- 16147 • The type of argument *path* is changed from **char *** to **const char ***.
- 16148 • The description of [EACCES] is updated to indicate that this error will also be returned if
- 16149 write permission is denied to the parent directory.

16150 The following changes are incorporated for alignment with the FIPS requirements:

- 16151 • In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a
- 16152 pathname component is larger than {NAME_MAX} is now defined as mandatory and marked
- 16153 as an extension.

16154 Another change is incorporated as follows:

- 16155 • The `<sys/types.h>` header is now marked as optional (OH); this header need not be included
- 16156 on XSI-conformant systems.

16157 **Issue 4, Version 2**

16158 The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- 16159 • It states that [ELOOP] will be returned if too many symbolic links are encountered during
- 16160 pathname resolution.
- 16161 • A second [ENAMETOOLONG] condition is defined that may report excessive length of an
- 16162 intermediate result of pathname resolution of a symbolic link.

16163 **NAME**

16164 mknod — make a directory, a special or regular file

16165 **SYNOPSIS**

16166 EX #include <sys/stat.h>

16167 int mknod(const char *path, mode_t mode, dev_t dev);

16168

16169 **DESCRIPTION**16170 The *mknod()* function creates a new file named by the pathname to which the argument *path*
16171 points.16172 The file type for *path* is OR-ed into the *mode* argument, and must be selected from one of the
16173 following symbolic constants:

16174

16175

16176

16177

16178

16179

16180

Name	Description
S_IFIFO	FIFO-special
S_IFCHR	Character-special (non-portable)
S_IFDIR	Directory (non-portable)
S_IFBLK	Block-special (non-portable)
S_IFREG	Regular (non-portable)

16181 The only portable use of *mknod()* is to create a FIFO-special file. If *mode* is not S_IFIFO or *dev* is
16182 not 0, the behaviour of *mknod()* is unspecified.16183 The permissions for the new file are OR-ed into the *mode* argument, and may be selected from
16184 any combination of the following symbolic constants:

16185

16186

16187

16188

16189

16190

16191

16192

16193

16194

16195

16196

16197

16198

16199

16200

16201

Name	Description
S_ISUID	Set user ID on execution.
S_ISGID	Set group ID on execution.
S_IRWXU	Read, write or execute (search) by owner.
S_IRUSR	Read by owner.
S_IWUSR	Write by owner.
S_IXUSR	Execute (search) by owner.
S_IRWXG	Read, write or execute (search) by group.
S_IRGRP	Read by group.
S_IWGRP	Write by group.
S_IXGRP	Execute (search) by group.
S_IRWXO	Read, write or execute (search) by others.
S_IROTH	Read by others.
S_IWOTH	Write by others.
S_IXOTH	Execute (search) by others.
S_ISVTX	On directories, restricted deletion flag.

16202 The user ID of the file is initialised to the effective user ID of the process. The group ID of the
16203 file is initialised to either the effective group ID of the process or the group ID of the parent
16204 directory.16205 The owner, group, and other permission bits of *mode* are modified by the file mode creation
16206 mask of the process. The *mknod()* function clears each bit whose corresponding bit in the file
16207 mode creation mask of the process is set.

16208 Upon successful completion, *mknod()* marks for update the *st_atime*, *st_ctime* and *st_mtime* fields
 16209 of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new entry are
 16210 marked for update.

16211 Only a process with appropriate privileges may invoke *mknod()* for file types other than FIFO-
 16212 special.

16213 RETURN VALUE

16214 Upon successful completion, *mknod()* returns 0. Otherwise, it returns -1, the new file is not
 16215 created, and *errno* is set to indicate the error.

16216 ERRORS

16217 The *mknod()* function will fail if:

16218 [EPERM] The invoking process does not have appropriate privileges and the file type is
 16219 not FIFO-special.

16220 [ENOTDIR] A component of the path prefix is not a directory.

16221 [ENOENT] A component of the path prefix specified by *path* does not name an existing
 16222 directory or *path* is an empty string.

16223 [EACCES] A component of the path prefix denies search permission, or write permission
 16224 is denied on the parent directory.

16225 [EROFS] The directory in which the file is to be created is located on a read-only file
 16226 system.

16227 [EEXIST] The named file exists.

16228 [EIO] An I/O error occurred while accessing the file system.

16229 [EINVAL] An invalid argument exists.

16230 [ENOSPC] The directory that would contain the new file cannot be extended or the file
 16231 system is out of file allocation resources.

16232 [ELOOP] Too many symbolic links were encountered in resolving *path*.

16233 [ENAMETOOLONG]

16234 The length of a pathname exceeds {PATH_MAX}, or pathname component is
 16235 longer than {NAME_MAX}.

16236 The *mknod()* function may fail if:

16237 [ENAMETOOLONG]

16238 Pathname resolution of a symbolic link produced an intermediate result
 16239 whose length exceeds {PATH_MAX}.

16240 EXAMPLES

16241 None.

16242 APPLICATION USAGE

16243 For portability to implementations conforming to earlier versions of this specification, *mkfifo()* is
 16244 preferred over this function for making FIFO special files.

16245 FUTURE DIRECTIONS

16246 None.

16247 SEE ALSO

16248 *chmod()*, *creat()*, *exec*, *mkdir()*, *mkfifo()*, *open()*, *stat()*, *umask()*, <sys/stat.h>.

16249 **CHANGE HISTORY**

16250 First released in Issue 4, Version 2.

16251 **Issue 5**

16252 Moved from X/OPEN UNIX extension to BASE.

|
|

16253 **NAME**

16254 mkstemp — make a unique file name

16255 **SYNOPSIS**16256 EX `#include <stdlib.h>`16257 `int mkstemp(char *template);`

16258

16259 **DESCRIPTION**

16260 The *mkstemp()* function replaces the contents of the string pointed to by *template* by a unique file
 16261 name, and returns a file descriptor for the file open for reading and writing. The function thus
 16262 prevents any possible race condition between testing whether the file exists and opening it for
 16263 use. The string in *template* should look like a file name with six trailing 'X's; *mkstemp()* replaces
 16264 each 'X' with a character from the portable file name character set. The characters are chosen
 16265 such that the resulting name does not duplicate the name of an existing file.

16266 **RETURN VALUE**

16267 Upon successful completion, *mkstemp()* returns an open file descriptor. Otherwise `-1` is
 16268 returned if no suitable file could be created.

16269 **ERRORS**

16270 No errors are defined.

16271 **EXAMPLES**

16272 None.

16273 **APPLICATION USAGE**

16274 It is possible to run out of letters.

16275 The *mkstemp()* function need not check to determine whether the file name part of *template*
 16276 exceeds the maximum allowable file name length.

16277 For portability with previous versions of this document, *tmpfile()* is preferred over this function.16278 **FUTURE DIRECTIONS**

16279 None.

16280 **SEE ALSO**16281 *getpid()*, *open()*, *tmpfile()*, *tmpnam()*, `<stdlib.h>`.16282 **CHANGE HISTORY**

16283 First released in Issue 4, Version 2.

16284 **Issue 5**

16285 Moved from X/OPEN UNIX extension to BASE.

16286 **NAME**

16287 mktemp — make a unique filename

16288 **SYNOPSIS**16289 EX `#include <stdlib.h>`16290 `char *mktemp(char *template);`

16291

16292 **DESCRIPTION**

16293 The *mktemp()* function replaces the contents of the string pointed to by *template* by a unique
16294 filename and returns *template*. The application must initialise *template* to be a filename with six
16295 trailing 'X's; *mktemp()* replaces each 'X' with a single byte character from the portable filename
16296 character set.

16297 **RETURN VALUE**

16298 The *mktemp()* function returns the pointer *template*. If a unique name cannot be created, *template*
16299 points to a null string.

16300 **ERRORS**

16301 No errors are defined.

16302 **EXAMPLES**

16303 None.

16304 **APPLICATION USAGE**

16305 Between the time a pathname is created and the file opened, it is possible for some other process
16306 to create a file with the same name. The *mkstemp()* function avoids this problem.

16307 For portability with previous versions of this document, *tmpnam()* is preferred over this
16308 function.

16309 **FUTURE DIRECTIONS**

16310 None.

16311 **SEE ALSO**16312 *mkstemp()*, *tmpfile()*, *tmpnam()*, *<stdlib.h>*.16313 **CHANGE HISTORY**

16314 First released in Issue 4, Version 2.

16315 **Issue 5**

16316 Moved from X/OPEN UNIX extension to BASE.

16317 **NAME**

16318 mktime — convert broken-down time into time since the Epoch

16319 **SYNOPSIS**

16320 #include <time.h>

16321 time_t mktime(struct tm *timeptr);

16322 **DESCRIPTION**

16323 The *mktime()* function converts the broken-down time, expressed as local time, in the structure pointed to by *timeptr*, into a time since the Epoch value with the same encoding as that of the values returned by *time()*. The original values of the *tm_wday* and *tm_yday* components of the structure are ignored, and the original values of the other components are not restricted to the ranges described in the <time.h> entry.

16328 A positive or 0 value for *tm_isdst* causes *mktime()* to presume initially that Daylight Savings Time, respectively, is or is not in effect for the specified time. A negative value for *tm_isdst* causes *mktime()* to attempt to determine whether Daylight Saving Time is in effect for the specified time.

16332 Local timezone information is set as though *mktime()* called *tzset()*.

16333 Upon successful completion, the values of the *tm_wday* and *tm_yday* components of the structure are set appropriately, and the other components are set to represent the specified time since the Epoch, but with their values forced to the ranges indicated in the <time.h> entry; the final value of *tm_mday* is not set until *tm_mon* and *tm_year* are determined.

16337 **RETURN VALUE**

16338 The *mktime()* function returns the specified time since the Epoch encoded as a value of type *time_t*. If the time since the Epoch cannot be represented, the function returns the value (*time_t*)-1.

16341 **ERRORS**

16342 No errors are defined.

16343 **EXAMPLES**

16344 What day of the week is July 4, 2001?

```

16345       #include <stdio.h>
16346       #include <time.h>
16347       struct tm time_str;
16348       char daybuf[20];
16349       int main(void)
16350       {
16351           time_str.tm_year = 2001 - 1900;
16352           time_str.tm_mon = 7 - 1;
16353           time_str.tm_mday = 4;
16354           time_str.tm_hour = 0;
16355           time_str.tm_min = 0;
16356           time_str.tm_sec = 1;
16357           time_str.tm_isdst = -1;
16358           if (mktime(&time_str) == -1)
16359               (void)puts("-unknown-");
16360           else {
16361               (void)strftime(daybuf, sizeof(daybuf), "%A", &time_str);
16362               (void)puts(daybuf);
16363           }

```



```

16364         return 0;
16365     }

```

16366 APPLICATION USAGE

16367 None.

16368 FUTURE DIRECTIONS

16369 None.

16370 SEE ALSO

16371 *asctime()*, *clock()*, *ctime()*, *difftime()*, *gmtime()*, *localtime()*, *strftime()*, *strptime()*, *time()*, *utime()*,
 16372 **<time.h>**.

16373 CHANGE HISTORY

16374 First released in Issue 3.

16375 Entry included for alignment with the POSIX.1-1988 standard and ANSI C standard.

16376 Issue 4

16377 The following changes are incorporated in this issue:

- 16378 • In the DESCRIPTION, a paragraph is added indicating the possible settings of *tm_isdst*, and
 16379 reference to setting of *tm_sec* for leap seconds or double leap seconds is removed (although
 16380 this functionality is still supported).
- 16381 • In the EXAMPLES section, the sample code is updated to use ISO C syntax.

16382 NAME

16383 mlock, munlock — lock or unlock a range of process address space (**REALTIME**)

16384 SYNOPSIS

16385 RT `#include <sys/mman.h>`16386 `int mlock(const void * addr, size_t len);`16387 `int munlock(const void * addr, size_t len);`

16388

16389 DESCRIPTION

16390 The function *mlock()* causes those whole pages containing any part of the address space of the
 16391 process starting at address *addr* and continuing for *len* bytes to be memory resident until
 16392 unlocked or until the process exits or *execs* another process image. The implementation may
 16393 require that *addr* be a multiple of {PAGESIZE}.

16394 The function *munlock()* unlocks those whole pages containing any part of the address space of
 16395 the process starting at address *addr* and continuing for *len* bytes, regardless of how many times
 16396 *mlock()* has been called by the process for any of the pages in the specified range. The
 16397 implementation may require that *addr* be a multiple of the {PAGESIZE}.

16398 If any of the pages in the range specified to a call to *munlock()* are also mapped into the address
 16399 spaces of other processes, any locks established on those pages by another process are
 16400 unaffected by the call of this process to *munlock()*. If any of the pages in the range specified by a
 16401 call to *munlock()* are also mapped into other portions of the address space of the calling process
 16402 outside the range specified, any locks established on those pages via the other mappings are also
 16403 unaffected by this call.

16404 Upon successful return from *mlock()*, pages in the specified range will be locked and memory
 16405 resident. Upon successful return from *munlock()*, pages in the specified range will be unlocked
 16406 with respect to the address space of the process. Memory residency of unlocked pages is
 16407 unspecified.

16408 The appropriate privilege is required to lock process memory with *mlock()*.

16409 RETURN VALUE

16410 Upon successful completion, the *mlock()* and *munlock()* functions return a value of zero.
 16411 Otherwise, no change is made to any locks in the address space of the process, and the function
 16412 returns a value of `-1` and sets *errno* to indicate the error.

16413 ERRORS

16414 The *mlock()* and *munlock()* functions will fail if:

16415 [ENOMEM] Some or all of the address range specified by the *addr* and *len* arguments does
 16416 not correspond to valid mapped pages in the address space of the process.

16417 [ENOSYS] The implementation does not support this memory locking interface.

16418 The *mlock()* functions will fail if:

16419 [EAGAIN] Some or all of the memory identified by the operation could not be locked
 16420 when the call was made.

16421 The *mlock()* and *munlock()* functions may fail if:

16422 [EINVAL] The *addr* argument is not a multiple of {PAGESIZE}.

16423 The *mlock()* function may fail if:

16424 [ENOMEM] Locking the pages mapped by the specified range would exceed an
 16425 implementation-dependent limit on the amount of memory that the process

16426		may lock.	
16427	[EPERM]	The calling process does not have the appropriate privilege to perform the	
16428		requested operation.	
16429	EXAMPLES		
16430	None.		
16431	APPLICATION USAGE		
16432	None.		
16433	SEE ALSO		
16434	<i>exec</i> , <i>_exit()</i> , <i>fork()</i> , <i>mlockall()</i> , <i>munmap()</i> , <i><sys/mman.h></i> .		
16435	CHANGE HISTORY		
16436	First released in Issue 5.		
16437	Included for alignment with the POSIX Realtime Extension.		

16438 NAME

16439 mlockall, munlockall — lock/unlock the address space of a process (**REALTIME**)

16440 SYNOPSIS

16441 RT `#include <sys/mman.h>`

16442 `int mlockall(int flags);`

16443 `int munlockall(void);`

16444

16445 DESCRIPTION

16446 The function *mlockall()* causes all of the pages mapped by the address space of a process to be
 16447 memory resident until unlocked or until the process exits or *execs* another process image. The
 16448 *flags* argument determines whether the pages to be locked are those currently mapped by the
 16449 address space of the process, those that will be mapped in the future, or both. The *flags*
 16450 argument is constructed from the inclusive OR of one or more of the following symbolic
 16451 constants, defined in `<sys/mman.h>`:

16452 MCL_CURRENT Lock all of the pages currently mapped into the address space of the process.

16453 MCL_FUTURE Lock all of the pages that become mapped into the address space of the
 16454 process in the future, when those mappings are established.

16455 If MCL_FUTURE is specified, and the automatic locking of future mappings eventually causes
 16456 the amount of locked memory to exceed the amount of available physical memory or any other
 16457 implementation-dependent limit, the behaviour is implementation-dependent. The manner in
 16458 which the implementation informs the application of these situations is also implementation-
 16459 dependent.

16460 The *munlockall()* function unlocks all currently mapped pages of the address space of the
 16461 process. Any pages that become mapped into the address space of the process after a call to
 16462 *munlockall()* will not be locked, unless there is an intervening call to *mlockall()* specifying
 16463 MCL_FUTURE or a subsequent call to *mlockall()* MCL_CURRENT. If pages mapped into the
 16464 address space of the process are also mapped into the address spaces of other processes and are
 16465 locked by those processes, the locks established by the other processes are unaffected by a call
 16466 by this process to *munlockall()*.

16467 Upon successful return from the *mlockall()* function that specifies MCL_CURRENT, all currently
 16468 mapped pages of the process's address space will be memory resident and locked. Upon return
 16469 from the *munlockall()* function, all currently mapped pages of the process's address space will be
 16470 unlocked with respect to the process's address space. The memory residency of unlocked pages
 16471 is unspecified.

16472 The appropriate privilege is required to lock process memory with *mlockall()*.

16473 RETURN VALUE

16474 Upon successful completion, the *mlockall()* function returns a value of zero. Otherwise, no
 16475 additional memory is locked, and the function returns a value of -1 and sets *errno* to indicate the
 16476 error. The effect of failure of *mlockall()* on previously existing locks in the address space is
 16477 unspecified.

16478 If it is supported by the implementation, the *munlockall()* function always returns a value of
 16479 zero. Otherwise, the function returns a value of -1 and sets *errno* to indicate the error.

16480 ERRORS

16481 The *mlockall()* and *munlockall()* functions will fail if:

16482 [ENOSYS] The implementation does not support this memory locking interface.

16483	The <i>mlockall()</i> function will fail if:		
16484	[EAGAIN]	Some or all of the memory identified by the operation could not be locked	
16485		when the call was made.	
16486	[EINVAL]	The <i>flags</i> argument is zero, or includes unimplemented flags.	
16487	The <i>mlockall()</i> function may fail if:		
16488	[ENOMEM]	Locking all of the pages currently mapped into the address space of the	
16489		process would exceed an implementation-dependent limit on the amount of	
16490		memory that the process may lock.	
16491	[EPERM]	The calling process does not have the appropriate privilege to perform the	
16492		requested operation.	
16493	EXAMPLES		
16494	None.		
16495	APPLICATION USAGE		
16496	None.		
16497	SEE ALSO		
16498	<i>exec</i> , <i>_exit()</i> , <i>fork()</i> , <i>mlock()</i> , <i>munmap()</i> , <i><sys/mman.h></i> .		
16499	CHANGE HISTORY		
16500	First released in Issue 5.		
16501	Included for alignment with the POSIX Realtime Extension.		

16502 NAME

16503 mmap — map pages of memory

16504 SYNOPSIS

16505 #include <sys/mman.h>

```
16506 void *mmap(void *addr, size_t len, int prot, int flags,
16507            int fildes, off_t off);
```

16508 DESCRIPTION

16509 RT The *mmap()* function establishes a mapping between a process' address space and a file or
 16510 shared memory object. The format of the call is as follows:

```
16511 pa=mmap(addr, len, prot, flags, fildes, off);
```

16512 The *mmap()* function establishes a mapping between the address space of the process at an
 16513 address *pa* for *len* bytes to the memory object represented by the file descriptor *fildes* at offset *off*
 16514 for *len* bytes. The value of *pa* is an implementation-dependent function of the parameter *addr*
 16515 and the values of *flags*, further described below. A successful *mmap()* call returns *pa* as its result.
 16516 The address range starting at *pa* and continuing for *len* bytes will be legitimate for the possible
 16517 (not necessarily current) address space of the process. The range of bytes starting at *off* and
 16518 continuing for *len* bytes will be legitimate for the possible (not necessarily current) offsets in the
 16519 RT file or shared memory object represented by *fildes*.

16520 The mapping established by *mmap()* replaces any previous mappings for those whole pages
 16521 containing any part of the address space of the process starting at *pa* and continuing for *len*
 16522 bytes.

16523 If the size of the mapped file changes after the call to *mmap()* as a result of some other operation
 16524 on the mapped file, the effect of references to portions of the mapped region that correspond to
 16525 added or removed portions of the file is unspecified.

16526 RT The *mmap()* function is supported for regular files and shared memory objects. Support for any
 16527 other type of file is unspecified.

16528 The parameter *prot* determines whether read, write, execute, or some combination of accesses
 16529 are permitted to the data being mapped. The *prot* should be either PROT_NONE or the bitwise
 16530 inclusive OR of one or more of the other flags in the following table, defined in the header
 16531 <sys/mman.h>.

16532

16533

16534

16535

16536

16537

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.

16538 If an implementation cannot support the combination of access types specified by *prot*, the call
 16539 to *mmap()* fails. An implementation may permit accesses other than those specified by *prot*;
 16540 however, the implementation will not permit a write to succeed where PROT_WRITE has not
 16541 been set or permit any access where PROT_NONE alone has been set. The implementation will
 16542 support at least the following values of *prot*: PROT_NONE, PROT_READ, PROT_WRITE, and
 16543 the inclusive OR of PROT_READ and PROT_WRITE. The file descriptor *fildes* will have been
 16544 opened with read permission, regardless of the protection options specified. If PROT_WRITE is
 16545 specified, the application must have opened the file descriptor *fildes* with write permission
 16546 unless MAP_PRIVATE is specified in the *flags* parameter as described below.

16547 The parameter *flags* provides other information about the handling of the mapped data. The
 16548 value of *flags* is the bitwise inclusive OR of these options, defined in `<sys/mman.h>`:

16549
 16550
 16551
 16552
 16553

Symbolic Constant	Description
MAP_SHARED	Changes are shared.
MAP_PRIVATE	Changes are private.
MAP_FIXED	Interpret <i>addr</i> exactly.

16554 MAP_SHARED and MAP_PRIVATE describe the disposition of write references to the memory
 16555 object. If MAP_SHARED is specified, write references change the underlying object. If
 16556 MAP_PRIVATE is specified, modifications to the mapped data by the calling process will be
 16557 visible only to the calling process and will not change the underlying object. It is unspecified
 16558 whether modifications to the underlying object done after the MAP_PRIVATE mapping is
 16559 established are visible through the MAP_PRIVATE mapping. Either MAP_SHARED or
 16560 MAP_PRIVATE can be specified, but not both. The mapping type is retained across *fork()*.

16561 When MAP_FIXED is set in the *flags* argument, the implementation is informed that the value of
 16562 EX *pa* must be *addr*, exactly. If MAP_FIXED is set, *mmap()* may return `(void *)-1` and set *errno* to
 16563 [EINVAL]. If a MAP_FIXED request is successful, the mapping established by *mmap()* replaces
 16564 any previous mappings for the process' pages in the range [*pa*, *pa + len*).

16565 When MAP_FIXED is not set, the implementation uses *addr* in an unspecified manner to arrive
 16566 at *pa*. The *pa* so chosen will be an area of the address space that the implementation deems
 16567 suitable for a mapping of *len* bytes to the file. All implementations interpret an *addr* value of 0 as
 16568 granting the implementation complete freedom in selecting *pa*, subject to constraints described
 16569 below. A non-zero value of *addr* is taken to be a suggestion of a process address near which the
 16570 mapping should be placed. When the implementation selects a value for *pa*, it never places a
 16571 mapping at address 0, nor does it replace any extant mapping.

16572 The *off* argument is constrained to be aligned and sized according to the value returned by
 16573 *sysconf()* when passed `_SC_PAGESIZE` or `_SC_PAGE_SIZE`. When MAP_FIXED is specified, the
 16574 argument *addr* must also meet these constraints. The implementation performs mapping
 16575 operations over whole pages. Thus, while the argument *len* need not meet a size or alignment
 16576 constraint, the implementation will include, in any mapping operation, any partial page
 16577 specified by the range [*pa*, *pa + len*).

16578 The system always zero-fills any partial page at the end of an object. Further, the system never
 16579 writes out any modified portions of the last page of an object that are beyond its end. References
 16580 within the address range starting at *pa* and continuing for *len* bytes to whole pages following the
 16581 end of an object result in delivery of a SIGBUS signal.

16582 An implementation may deliver SIGBUS signals when a reference would cause an error in the
 16583 mapped object, such as out-of-space condition.

16584 EX The *mmap()* function adds an extra reference to the file associated with the file descriptor *fd*es
 16585 which is not removed by a subsequent *close()* on that file descriptor. This reference is removed
 16586 when there are no more mappings to the file.

16587 The *st_atime* field of the mapped file may be marked for update at any time between the
 16588 *mmap()* call and the corresponding *munmap()* call. The initial read or write reference to a
 16589 mapped region will cause the file's *st_atime* field to be marked for update if it has not already
 16590 been marked for update.

16591 The *st_ctime* and *st_mtime* fields of a file that is mapped with MAP_SHARED and
 16592 PROT_WRITE, will be marked for update at some point in the interval between a write reference
 16593 to the mapped region and the next call to *msync()* with MS_ASYNC or MS_SYNC for that

16594 portion of the file by any process. If there is no such call, these fields may be marked for update
 16595 at any time after a write reference if the underlying file is modified as a result.

16596 EX There may be implementation-dependent limits on the number of memory regions that can be
 16597 mapped (per process or per system). If such a limit is imposed, whether the number of memory
 16598 regions that can be mapped by a process is decreased by the use of *shmat()* is implementation-
 16599 dependent.

16600 RETURN VALUE

16601 Upon successful completion, the *mmap()* function returns the address at which the mapping was
 16602 placed (pa); otherwise, it returns a value of MAP_FAILED and sets *errno* to indicate the error.
 16603 The symbol MAP_FAILED is defined in the header <sys/mman.h>. No successful return from
 16604 *mmap()* will return the value MAP_FAILED.

16605 If *mmap()* fails for reasons other than [EBADF], [EINVAL] or [ENOTSUP], some of the mappings
 16606 in the address range starting at *addr* and continuing for *len* bytes may have been unmapped.

16607 ERRORS

16608 The *mmap()* function will fail if:

16609 [EACCES] The *fildev* argument is not open for read, regardless of the protection specified,
 16610 or *fildev* is not open for write and PROT_WRITE was specified for a
 16611 MAP_SHARED type mapping.

16612 RT [EAGAIN] The mapping could not be locked in memory, if required by *mlockall()*, due to
 16613 a lack of resources.

16614 [EBADF] The *fildev* argument is not a valid open file descriptor.

16615 EX [EINVAL] The *addr* argument (if MAP_FIXED was specified) or *off* is not a multiple of
 16616 the page size as returned by *sysconf()*, or are considered invalid by the
 16617 implementation.

16618 EX [EINVAL] The value of *flags* is invalid (neither MAP_PRIVATE nor MAP_SHARED is
 16619 set).

16620 EX [EMFILE] The number of mapped regions would exceed an implementation-dependent
 16621 limit (per process or per system).

16622 [ENODEV] The *fildev* argument refers to a file whose type is not supported by *mmap()*.

16623 [ENOMEM] MAP_FIXED was specified, and the range [*addr*, *addr* + *len*) exceeds that
 16624 allowed for the address space of a process; or if MAP_FIXED was not
 16625 specified and there is insufficient room in the address space to effect the
 16626 mapping.

16627 RT [ENOMEM] The mapping could not be locked in memory, if required by *mlockall()*,
 16628 because it would require more space than the system is able to supply.

16629 [ENOTSUP] The implementation does not support the combination of accesses requested
 16630 in the *prot* argument.

16631 [ENXIO] Addresses in the range [*off*, *off* + *len*) are invalid for the object specified by
 16632 *fildev*.

16633 [ENXIO] MAP_FIXED was specified in *flags* and the combination of *addr*, *len* and *off* is
 16634 invalid for the object specified by *fildev*.

16635 EX [EOVERFLOW] The file is a regular file and the value of *off* plus *len* exceeds the offset
 16636 maximum established in the open file description associated with *fildev*.

16637 **EXAMPLES**

16638 None.

16639 **APPLICATION USAGE**16640 Use of *mmap()* may reduce the amount of memory available to other memory allocation
16641 functions.16642 Use of MAP_FIXED may result in unspecified behaviour in further use of *brk()*, *sbrk()*, *malloc()*
16643 and *shmat()*. The use of MAP_FIXED is discouraged, as it may prevent an implementation from
16644 making the most effective use of resources.16645 The application must ensure correct synchronisation when using *mmap()* in conjunction with
16646 any other file access method, such as *read()* and *write()*, standard input/output, and *shmat()*.16647 The *mmap()* function allows access to resources via address space manipulations, instead of
16648 *read()/write()*. Once a file is mapped, all a process has to do to access it is use the data at the
16649 address to which the file was mapped. So, using pseudo-code to illustrate the way in which an
16650 existing program might be changed to use *mmap()*, the following:16651 fildes = open(...)
16652 lseek(fildes, some_offset)
16653 read(fildes, buf, len)
16654 /* use data in buf */

16655 becomes:

16656 fildes = open(...)
16657 address = mmap(0, len, PROT_READ, MAP_PRIVATE, fildes, some_offset)
16658 /* use data at address */16659 The [EINVAL] error above is marked EX because it is defined as an optional error in the POSIX
16660 Realtime Extension.16661 **FUTURE DIRECTIONS**

16662 None.

16663 **SEE ALSO**16664 *brk()*, *exec*, *fcntl()*, *fork()*, *lockf()*, *msync()*, *munmap()*, *mprotect()*, *sbrk()*, *shmat()*, *sysconf()*,
16665 <sys/mman.h>.16666 **CHANGE HISTORY**

16667 First released in Issue 4, Version 2.

16668 **Issue 5**16669 Moved from X/OPEN UNIX extension to BASE and aligned with *mmap()* in the POSIX Realtime
16670 Extension. Specifically, the DESCRIPTION is extensively reworded, [EAGAIN] and [ENOTSUP]
16671 are added to the mandatory errors, and new cases of [ENOMEM] and [ENXIO] are added to the
16672 mandatory errors. Also the value returned on failure is the value of the constant MAP_FAILED;
16673 this was previously defined as -1.

16674 Large File Summit extensions added.

16675 **NAME**

16676 modf — decompose a floating-point number

16677 **SYNOPSIS**

16678 #include <math.h>

16679 double modf(double *x*, double **iptr*);16680 **DESCRIPTION**

16681 The *modf()* function breaks the argument *x* into integral and fractional parts, each of which has
 16682 the same sign as the argument. It stores the integral part as a double in the object pointed to by
 16683 *iptr*.

16684 An application wishing to check for error situations should set *errno* to 0 before calling *modf()*. If
 16685 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

16686 **RETURN VALUE**16687 Upon successful completion, *modf()* returns the signed fractional part of *x*.16688 EX If *x* is NaN, NaN is returned, *errno* may be set to [EDOM] and **iptr* is set to NaN.16689 If the correct value would cause underflow, 0 is returned and *errno* may be set to [ERANGE].16690 **ERRORS**16691 The *modf()* function may fail if:16692 EX [EDOM] The value of *x* is NaN.

16693 [ERANGE] The result underflows.

16694 EX No other errors will occur.

16695 **EXAMPLES**

16696 None.

16697 **APPLICATION USAGE**

16698 None.

16699 **SEE ALSO**16700 *frexp()*, *isnan()*, *ldexp()*, <math.h>.16701 **CHANGE HISTORY**

16702 First released in Issue 1.

16703 Derived from Issue 1 of the SVID.

16704 **Issue 4**

16705 The following changes are incorporated in this issue:

- 16706 • Removed references to *matherr()*.
- 16707 • The name of the first argument is changed from *value* to *x*.
- 16708 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
- 16709 the ISO C standard and to rationalise error handling in the mathematics functions.
- 16710 • The return value specified for [EDOM] is marked as an extension.

16711 **Issue 5**

16712 The DESCRIPTION is updated to indicate how an application should check for an error. This
16713 text was previously published in the APPLICATION USAGE section.

16714 **NAME**

16715 mprotect — set protection of memory mapping

16716 **SYNOPSIS**

16717 #include <sys/mman.h>

16718 int mprotect(void *addr, size_t len, int prot);

16719 **DESCRIPTION**

16720 The function *mprotect()* changes the access protections to be that specified by *prot* for those
 16721 whole pages containing any part of the address space of the process starting at address *addr* and
 16722 continuing for *len* bytes. The parameter *prot* determines whether read, write, execute, or some
 16723 combination of accesses are permitted to the data being mapped. The *prot* argument should be
 16724 either PROT_NONE or the bitwise inclusive OR of one or more of PROT_READ, PROT_WRITE
 16725 and PROT_EXEC.

16726 If an implementation cannot support the combination of access types specified by *prot*, the call
 16727 to *mprotect()* fails.

16728 An implementation may permit accesses other than those specified by *prot*; however, no
 16729 implementation permits a write to succeed where PROT_WRITE has not been set or permits any
 16730 access where PROT_NONE alone has been set. Implementations will support at least the
 16731 following values of *prot*: PROT_NONE, PROT_READ, PROT_WRITE, and the inclusive OR of
 16732 PROT_READ and PROT_WRITE. If PROT_WRITE is specified, the application must have
 16733 opened the mapped objects in the specified address range with write permission, unless
 16734 MAP_PRIVATE was specified in the original mapping, regardless of whether the file descriptors
 16735 used to map the objects have since been closed.

16736 EX The implementation will require that *addr* be a multiple of the page size as returned by *sysconf()*.

16737 The behaviour of this function is unspecified if the mapping was not established by a call to
 16738 *mmap()*.

16739 When *mprotect()* fails for reasons other than [EINVAL], the protections on some of the pages in
 16740 the range [*addr*, *addr* + *len*) may have been changed.

16741 **RETURN VALUE**

16742 Upon successful completion, *mprotect()* returns 0. Otherwise, it returns -1 and sets *errno* to
 16743 indicate the error.

16744 **ERRORS**16745 The *mprotect()* function will fail if:

16746 [EACCES] The *prot* argument specifies a protection that violates the access permission
 16747 the process has to the underlying memory object.

16748 [EAGAIN] The *prot* argument specifies PROT_WRITE over a MAP_PRIVATE mapping
 16749 and there are insufficient memory resources to reserve for locking the private
 16750 page.

16751 EX [EINVAL] The *addr* argument is not a multiple of the page size as returned by *sysconf()*.

16752 [ENOMEM] Addresses in the range [*addr*, *addr* + *len*) are invalid for the address space of a
 16753 process, or specify one or more pages which are not mapped.

16754 [ENOMEM] The *prot* argument specifies PROT_WRITE on a MAP_PRIVATE mapping, and
 16755 it would require more space than the system is able to supply for locking the
 16756 private pages, if required.

16757 [ENOTSUP] The implementation does not support the combination of accesses requested
16758 in the *prot* argument.

16759 **EXAMPLES**

16760 None.

16761 **APPLICATION USAGE**

16762 The EINVAL error above is marked EX because it is defined as an optional error in the POSIX
16763 Realtime Extension.

16764 **FUTURE DIRECTIONS**

16765 None.

16766 **SEE ALSO**

16767 *mmap()*, *sysconf()*, <sys/mman.h>.

16768 **CHANGE HISTORY**

16769 First released in Issue 4, Version 2.

16770 **Issue 5**

16771 Moved from X/OPEN UNIX extension to BASE and aligned with *mprotect()* in the POSIX
16772 Realtime Extension. Specifically, the DESCRIPTION is largely reworded, [ENOTSUP] and a
16773 second form of [ENOMEM] are added to the mandatory errors, [EAGAIN] is moved from the
16774 optional to the mandatory errors.

16775 **NAME**16776 mq_close — close a message queue (**REALTIME**)16777 **SYNOPSIS**

16778 RT #include <mqueue.h>

16779 int mq_close(mqd_t mqdes);

16780

16781 **DESCRIPTION**

16782 The *mq_close()* function removes the association between the message queue descriptor, *mqdes*,
 16783 and its message queue. The results of using this message queue descriptor after successful
 16784 return from this *mq_close()*, and until the return of this message queue descriptor from a
 16785 subsequent *mq_open()*, are undefined.

16786 If the process has successfully attached a notification request to the message queue via this
 16787 *mqdes*, this attachment will be removed, and the message queue is available for another process
 16788 to attach for notification.

16789 **RETURN VALUE**

16790 Upon successful completion, the *mq_close()* function returns a value of zero; otherwise, the
 16791 function returns a value of -1 and sets *errno* to indicate the error.

16792 **ERRORS**16793 The *mq_close()* function will fail if:

16794 [EBADF] The *mqdes* argument is not a valid message queue descriptor.

16795 [ENOSYS] The function *mq_close()* is not supported by this implementation.

16796 **EXAMPLES**

16797 None.

16798 **APPLICATION USAGE**

16799 None.

16800 **SEE ALSO**16801 *mq_open()*, *mq_unlink()*, <mqueue.h>, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*.16802 **CHANGE HISTORY**

16803 First released in Issue 5.

16804 Included for alignment with the POSIX Realtime Extension.

16805 **NAME**16806 mq_getattr — get message queue attributes (**REALTIME**)16807 **SYNOPSIS**

16808 RT #include <mqqueue.h>

16809 int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);

16810

16811 **DESCRIPTION**

16812 The *mqdes* argument specifies a message queue descriptor. The *mq_getattr()* function is used to
 16813 get status information and attributes of the message queue and the open message queue
 16814 description associated with the message queue descriptor. The results are returned in the
 16815 *mq_attr* structure referenced by the *mqstat* argument.

16816 Upon return, the following members will have the values associated with the open message
 16817 queue description as set when the message queue was opened and as modified by subsequent
 16818 *mq_setattr()* calls:

16819 mq_flags

16820 The following attributes of the message queue are returned as set at message queue creation.

16821 mq_maxmsg

16822 mq_msgsize

16823 mq_curmsgs The number of messages currently on the queue.

16824 **RETURN VALUE**

16825 Upon successful completion, the *mq_getattr()* function returns zero. Otherwise, the function
 16826 returns -1 and sets *errno* to indicate the error.

16827 **ERRORS**16828 The *mq_getattr()* function will fail if:16829 [EBADF] The *mqdes* argument is not a valid message queue descriptor.16830 [ENOSYS] The function *mq_getattr()* is not supported by this implementation.16831 **EXAMPLES**

16832 None.

16833 **APPLICATION USAGE**

16834 None.

16835 **SEE ALSO**16836 *mq_open()*, *mq_send()*, *mq_setattr()* <mqqueue.h>, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*.16837 **CHANGE HISTORY**

16838 First released in Issue 5.

16839 Included for alignment with the POSIX Realtime Extension.

16840 **NAME**16841 mq_notify — notify process that a message is available (**REALTIME**)16842 **SYNOPSIS**16843 RT `#include <mqueue.h>`16844 `int mq_notify(mqd_t mqdes, const struct sigevent *notification);`

16845

16846 **DESCRIPTION**

16847 If the argument *notification* is not NULL, this function registers the calling process to be notified
 16848 of message arrival at an empty message queue associated with the specified message queue
 16849 descriptor, *mqdes*. The notification specified by the *notification* argument will be sent to the
 16850 process when the message queue transitions from empty to non-empty. At any time, only one
 16851 process may be registered for notification by a message queue. If the calling process or any
 16852 other process has already registered for notification of message arrival at the specified message
 16853 queue, subsequent attempts to register for that message queue fail.

16854 If *notification* is NULL and the process is currently registered for notification by the specified
 16855 message queue, the existing registration is removed.

16856 When the notification is sent to the registered process, its registration will be removed. The
 16857 message queue will then be available for registration.

16858 If a process has registered for notification of message arrival at a message queue and some
 16859 thread is blocked in *mq_receive()* waiting to receive a message when a message arrives at the
 16860 queue, the arriving message satisfies the appropriate *mq_receive()*. The resulting behaviour is as
 16861 if the message queue remains empty, and no notification is sent.

16862 **RETURN VALUE**

16863 Upon successful completion, the *mq_notify()* function returns a value of zero; otherwise, the
 16864 function returns a value of -1 and sets *errno* to indicate the error.

16865 **ERRORS**16866 The *mq_notify()* function will fail if:

- | | | |
|-------|----------|--|
| 16867 | [EBADF] | The <i>mqdes</i> argument is not a valid message queue descriptor. |
| 16868 | [EBUSY] | A process is already registered for notification by the message queue. |
| 16869 | [ENOSYS] | The function <i>mq_notify()</i> is not supported by this implementation. |

16870 **EXAMPLES**

16871 None.

16872 **APPLICATION USAGE**

16873 None.

16874 **SEE ALSO**16875 *mq_open()*, *mq_send()*, **<mqueue.h>**, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*.16876 **CHANGE HISTORY**

16877 First released in Issue 5.

16878 Included for alignment with the POSIX Realtime Extension.

16879 **NAME**16880 `mq_open` — open a message queue (**REALTIME**)16881 **SYNOPSIS**16882 RT `#include <mqueue.h>`16883 `mqd_t mq_open(const char *name, int oflag, ...);`

16884

16885 **DESCRIPTION**

16886 The `mq_open()` function establishes the connection between a process and a message queue with
 16887 a message queue descriptor. It creates a open message queue description that refers to the
 16888 message queue, and a message queue descriptor that refers to that open message queue
 16889 description. The message queue descriptor is used by other functions to refer to that message
 16890 queue. The *name* argument points to a string naming a message queue. It is unspecified
 16891 whether the name appears in the file system and is visible to other functions that take
 16892 pathnames as arguments. The *name* argument conforms to the construction rules for a
 16893 pathname. If *name* begins with the slash character, then processes calling `mq_open()` with the
 16894 same value of *name* refer to the same message queue object, as long as that name has not been
 16895 removed. If *name* does not begin with the slash character, the effect is implementation-
 16896 dependent. The interpretation of slash characters other than the leading slash character in *name*
 16897 is implementation-dependent. If the *name* argument is not the name of an existing message
 16898 queue and creation is not requested, `mq_open()` fails and returns an error.

16899 The *oflag* argument requests the desired receive and/or send access to the message queue. The
 16900 requested access permission to receive messages or send messages is granted if the calling
 16901 process would be granted read or write access, respectively, to an equivalently protected file.

16902 The value of *oflag* is the bitwise inclusive OR of values from the following list. Applications
 16903 specify exactly one of the first three values (access modes) below in the value of *oflag*:

16904 **O_RDONLY** Open the message queue for receiving messages. The process can use the
 16905 returned message queue descriptor with `mq_receive()`, but not `mq_send()`. A
 16906 message queue may be open multiple times in the same or different processes
 16907 for receiving messages.

16908 **O_WRONLY** Open the queue for sending messages. The process can use the returned
 16909 message queue descriptor with `mq_send()` but not `mq_receive()`. A message
 16910 queue may be open multiple times in the same or different processes for
 16911 sending messages.

16912 **O_RDWR** Open the queue for both receiving and sending messages. The process can
 16913 use any of the functions allowed for **O_RDONLY** and **O_WRONLY**. A
 16914 message queue may be open multiple times in the same or different processes
 16915 for sending messages.

16916 Any combination of the remaining flags may be specified in the value of *oflag*:

16917 **O_CREAT** This option is used to create a message queue, and it requires two additional
 16918 arguments: *mode*, which is of type **mode_t**, and *attr*, which is a pointer to a
 16919 **mq_attr** structure. If the pathname, *name*, has already been used to create a
 16920 message queue that still exists, then this flag has no effect, except as noted
 16921 under **O_EXCL**. Otherwise, a message queue is created without any messages
 16922 in it. The user ID of the message queue is set to the effective user ID of the
 16923 process, and the group ID of the message queue is set to the effective group ID
 16924 of the process. The file permission bits are set to the value of *mode*. When bits
 16925 in *mode* other than file permission bits are set, the effect is implementation-

16926		dependent. If <i>attr</i> is NULL, the message queue is created with
16927		implementation-dependent default message queue attributes. If <i>attr</i> is non-
16928		NULL and the calling process has the appropriate privilege on <i>name</i> , the
16929		message queue <i>mq_maxmsg</i> and <i>mq_msgsize</i> attributes are set to the values of
16930		the corresponding members in the mq_attr structure referred to by <i>attr</i> . If <i>attr</i>
16931		is non-NULL, but the calling process does not have the appropriate privilege
16932		on <i>name</i> , the <i>mq_open()</i> function fails and returns an error without creating the
16933		message queue.
16934	O_EXCL	If O_EXCL and O_CREAT are set, <i>mq_open()</i> fails if the message queue <i>name</i>
16935		exists. The check for the existence of the message queue and the creation of
16936		the message queue if it does not exist are atomic with respect to other
16937		processes executing <i>mq_open()</i> naming the same <i>name</i> with O_EXCL and
16938		O_CREAT set. If O_EXCL is set and O_CREAT is not set, the result is
16939		undefined.
16940	O_NONBLOCK	The setting of this flag is associated with the open message queue description
16941		and determines whether a <i>mq_send()</i> or <i>mq_receive()</i> waits for resources or
16942		messages that are not currently available, or fails with <i>errno</i> set to [EAGAIN].
16943		See <i>mq_send()</i> and <i>mq_receive()</i> for details.
16944	The <i>mq_open()</i> function does not add or remove messages from the queue.	
16945	RETURN VALUE	
16946	Upon successful completion, the function returns a message queue descriptor. Otherwise, the	
16947	function returns (mqd_t)−1 and sets <i>errno</i> to indicate the error.	
16948	ERRORS	
16949	The <i>mq_open()</i> function will fail if:	
16950	[EACCES]	The message queue exists and the permissions specified by <i>oflag</i> are denied, or
16951		the message queue does not exist and permission to create the message queue
16952		is denied.
16953	[EEXIST]	O_CREAT and O_EXCL are set and the named message queue already exists.
16954	[EINTR]	The <i>mq_open()</i> operation was interrupted by a signal.
16955	[EINVAL]	The <i>mq_open()</i> operation is not supported for the given name.
16956	[EINVAL]	O_CREAT was specified in <i>oflag</i> , the value of <i>attr</i> is not NULL, and either
16957		<i>mq_maxmsg</i> or <i>mq_msgsize</i> was less than or equal to zero.
16958	[EMFILE]	Too many message queue descriptors or file descriptors are currently in use by
16959		this process.
16960	[ENAMETOOLONG]	
16961		The length of the <i>name</i> string exceeds {PATH_MAX}, or a pathname
16962		component is longer than {NAME_MAX} while _POSIX_NO_TRUNC is in
16963		effect.
16964	[ENFILE]	Too many message queues are currently open in the system.
16965	[ENOENT]	O_CREAT is not set and the named message queue does not exist.
16966	[ENOSPC]	There is insufficient space for the creation of the new message queue.
16967	[ENOSYS]	The function <i>mq_open()</i> is not supported by this implementation.

16968 **EXAMPLES**

16969 None.

16970 **APPLICATION USAGE**

16971 None.

16972 **SEE ALSO**

16973 *mq_close()*, *mq_receive()*, *mq_send()*, *mq_setattr()*, *mq_getattr()*, *mq_unlink()*, *<mqueue.h>*,
16974 *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*.

16975 **CHANGE HISTORY**

16976 First released in Issue 5.

16977 Included for alignment with the POSIX Realtime Extension.

16978 NAME

16979 mq_receive — receive a message from a message queue (**REALTIME**)

16980 SYNOPSIS

16981 RT `#include <mqueue.h>`

16982 `ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,`
 16983 `unsigned int *msg_prio);`
 16984

16985 DESCRIPTION

16986 The *mq_receive()* function is used to receive the oldest of the highest priority message(s) from the
 16987 message queue specified by *mqdes*. If the size of the buffer in bytes, specified by the *msg_len*
 16988 argument, is less than the *mq_msgsize* attribute of the message queue, the function fails and
 16989 returns an error. Otherwise, the selected message is removed from the queue and copied to the
 16990 buffer pointed to by the *msg_ptr* argument.

16991 EX If the value of *maxsize* is greater than {SSIZE_MAX}, the result is implementation-dependent.

16992 If the argument *msg_prio* is not NULL, the priority of the selected message is stored in the
 16993 location referenced by *msg_prio*.

16994 If the specified message queue is empty and O_NONBLOCK is not set in the message queue
 16995 description associated with *mqdes*, *mq_receive()* blocks until a message is enqueued on the
 16996 message queue or until *mq_receive()* is interrupted by a signal. If more than one thread is
 16997 waiting to receive a message when a message arrives at an empty queue and the Priority
 16998 Scheduling option is supported, then the thread of highest priority that has been waiting the
 16999 longest will be selected to receive the message. Otherwise, it is unspecified which waiting
 17000 thread receives the message. If the specified message queue is empty and O_NONBLOCK is set
 17001 in the message queue description associated with *mqdes*, no message is removed from the queue,
 17002 and *mq_receive()* returns an error.

17003 RETURN VALUE

17004 Upon successful completion, *mq_receive()* returns the length of the selected message in bytes and
 17005 the message is removed from the queue. Otherwise, no message is removed from the queue, the
 17006 function returns a value of -1, and sets *errno* to indicate the error.

17007 ERRORS

17008 The *mq_receive()* function will fail if:

17009 [EAGAIN] O_NONBLOCK was set in the message description associated with *mqdes*,
 17010 and the specified message queue is empty.

17011 [EBADF] The *mqdes* argument is not a valid message queue descriptor open for reading.

17012 [EMSGSIZE] The specified message buffer size, *msg_len*, is less than the message size
 17013 attribute of the message queue.

17014 [EINTR] The *mq_receive()* operation was interrupted by a signal.

17015 [ENOSYS] The *mq_receive()* function is not supported by this implementation.

17016 The *mq_receive()* function may fail if:

17017 [EBADMSG] The implementation has detected a data corruption problem with the
 17018 message.

17019 EXAMPLES

17020 None.

17021 **APPLICATION USAGE**

17022 None.

17023 **SEE ALSO**17024 *mq_send()*, *<mqqueue.h>*, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*.17025 **CHANGE HISTORY**

17026 First released in Issue 5.

17027 Included for alignment with the POSIX Realtime Extension.

17028 **NAME**17029 mq_send — send a message to a message queue (**REALTIME**)17030 **SYNOPSIS**

17031 RT #include <mqueue.h>

```
17032       int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
17033                   unsigned int msg_prio);
```

17034

17035 **DESCRIPTION**

17036 The *mq_send()* function adds the message pointed to by the argument *msg_ptr* to the message
 17037 queue specified by *mqdes*. The *msg_len* argument specifies the length of the message in bytes
 17038 pointed to by *msg_ptr*. The value of *msg_len* is less than or equal to the *mq_msgsize* attribute of
 17039 the message queue, or *mq_send()* fails.

17040 If the specified message queue is not full, *mq_send()* behaves as if the message is inserted into
 17041 the message queue at the position indicated by the *msg_prio* argument. A message with a larger
 17042 numeric value of *msg_prio* is inserted before messages with lower values of *msg_prio*. A message
 17043 will be inserted after other messages in the queue, if any, with equal *msg_prio*. The value of
 17044 *msg_prio* will be less than MQ_PRIO_MAX.

17045 If the specified message queue is full and O_NONBLOCK is not set in the message queue
 17046 description associated with *mqdes*, *mq_send()* blocks until space becomes available to enqueue
 17047 the message, or until *mq_send()* is interrupted by a signal. If more than one thread is waiting to
 17048 send when space becomes available in the message queue and the Priority Scheduling option is
 17049 supported, then the thread of the highest priority that has been waiting the longest will be
 17050 unblocked to send its message. Otherwise, it is unspecified which waiting thread is unblocked.
 17051 If the specified message queue is full and O_NONBLOCK is set in the message queue
 17052 description associated with *mqdes*, the message is not queued and *mq_send()* returns an error.

17053 **RETURN VALUE**

17054 Upon successful completion, the *mq_send()* function returns a value of zero. Otherwise, no
 17055 message is enqueued, the function returns -1, and is set to indicate the error.

17056 **ERRORS**17057 The *mq_send()* function will fail if:

17058 [EAGAIN] The O_NONBLOCK flag is set in the message queue description associated
 17059 with *mqdes*, and the specified message queue is full.

17060 [EBADF] The *mqdes* argument is not a valid message queue descriptor open for writing.

17061 [EINTR] A signal interrupted the call to *mq_send()*.

17062 [EINVAL] The value of *msg_prio* was outside the valid range.

17063 [EMSGSIZE] The specified message length, *msg_len*, exceeds the message size attribute of
 17064 the message queue.

17065 [ENOSYS] The function *mq_send()* is not supported by this implementation.

17066 **EXAMPLES**

17067 None.

17068 **APPLICATION USAGE**

17069 None.

17070 **SEE ALSO**17071 *mq_receive()*, *mq_setattr()*, <mqueue.h>.

17072 **CHANGE HISTORY**

17073 First released in Issue 5.

17074 Included for alignment with the POSIX Realtime Extension.

|

17075 **NAME**17076 mq_setattr — set message queue attributes (**REALTIME**)17077 **SYNOPSIS**

17078 RT #include <mqueue.h>

```
17079       int mq_setattr(mqd_t mqdes, const struct mq_attr *mqstat,
17080                      struct mq_attr *omqstat);
```

17081

17082 **DESCRIPTION**

17083 The *mq_setattr()* function is used to set attributes associated with the open message queue
 17084 description referenced by the message queue descriptor specified by *mqdes*.

17085 The message queue attributes corresponding to the following members defined in the **mq_attr**
 17086 structure are set to the specified values upon successful completion of *mq_setattr()*:

17087 mq_flags The value of this member is the bitwise logical OR of zero or more of
 17088 O_NONBLOCK and any implementation-dependent flags.

17089 The values of the *mq_maxmsg*, *mq_msgsize* and *mq_curmsgs* members of the **mq_attr** structure are
 17090 ignored by *mq_setattr()*.

17091 If *omqstat* is non-NULL, the function *mq_setattr()* stores, in the location referenced by *omqstat*, the
 17092 previous message queue attributes and the current queue status. These values are the same as
 17093 would be returned by a call to *mq_getattr()* at that point.

17094 **RETURN VALUE**

17095 Upon successful completion, the function returns a value of zero and the attributes of the
 17096 message queue will have been changed as specified. Otherwise, the message queue attributes
 17097 are unchanged, and the function returns a value of -1 and sets *errno* to indicate the error.

17098 **ERRORS**17099 The *mq_setattr()* function will fail if:

17100 [EBADF] The *mqdes* argument is not a valid message queue descriptor.

17101 [ENOSYS] The function *mq_setattr()* is not supported by this implementation.

17102 **EXAMPLES**

17103 None.

17104 **APPLICATION USAGE**

17105 None.

17106 **SEE ALSO**17107 *mq_open()*, *mq_send()*, <mqueue.h>, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*.17108 **CHANGE HISTORY**

17109 First released in Issue 5.

17110 Included for alignment with the POSIX Realtime Extension.

17111 **NAME**17112 mq_unlink — remove a message queue (**REALTIME**)17113 **SYNOPSIS**

17114 RT #include <mqueue.h>

17115 int mq_unlink(const char *name);

17116

17117 **DESCRIPTION**

17118 The *mq_unlink()* function removes the message queue named by the pathname *name*. After a
 17119 successful call to *mq_unlink()* with *name*, a call to *mq_open()* with *name* fails if the flag *O_CREAT*
 17120 is not set in *flags*. If one or more processes have the message queue open when *mq_unlink()* is
 17121 called, destruction of the message queue is postponed until all references to the message queue
 17122 have been closed. Calls to *mq_open()* to re-create the message queue may fail until the message
 17123 queue is actually removed. However, the *mq_unlink()* call need not block until all references
 17124 have been closed; it may return immediately.

17125 **RETURN VALUE**

17126 Upon successful completion, the function returns a value of zero. Otherwise, the named
 17127 message queue is changed by this function call, and the function returns a value of *-1* and sets
 17128 *errno* to indicate the error.

17129 **ERRORS**17130 The *mq_unlink()* function will fail if:

17131 [EACCES] Permission is denied to unlink the named message queue.

17132 [ENAMETOOLONG]

17133 The length of the *name* string exceeds {NAME_MAX} while
 17134 _POSIX_NO_TRUNC is in effect.

17135 [ENOENT] The named message queue does not exist.

17136 [ENOSYS] The function *mq_unlink()* is not supported by this implementation.17137 **EXAMPLES**

17138 None.

17139 **APPLICATION USAGE**

17140 None.

17141 **SEE ALSO**17142 *mq_close()*, *mq_open()*, <mqueue.h>, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*.17143 **CHANGE HISTORY**

17144 First released in Issue 5.

17145 Included for alignment with the POSIX Realtime Extension.

17146 **NAME**

17147 mrnd48 — generate uniformly distributed pseudo-random signed long integers

17148 **SYNOPSIS**

17149 EX #include <stdlib.h>

17150 long int mrnd48(void);

17151

17152 **DESCRIPTION**

17153 Refer to *drand48()*.

17154 **CHANGE HISTORY**

17155 First released in Issue 1.

17156 Derived from Issue 1 of the SVID.

17157 **Issue 4**

17158 The following changes are incorporated in this issue:

- 17159 • The <stdlib.h> header is now required.
- 17160 • The *mrnd48()* function is now defined to return **long int**.
- 17161 • The argument list now includes **void**.

17162 **NAME**

17163 msgctl — message control operations

17164 **SYNOPSIS**17165 EX

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

17166

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

17167

17168 **DESCRIPTION**17169 The *msgctl()* function provides message control operations as specified by *cmd*. The following
17170 values for *cmd*, and the message control operations they specify, are:17171 **IPC_STAT** Place the current value of each member of the **msqid_ds** data structure
17172 associated with *msqid* into the structure pointed to by *buf*. The contents of this
17173 structure are defined in *<sys/msg.h>*.17174 **IPC_SET** Set the value of the following members of the **msqid_ds** data structure
17175 associated with *msqid* to the corresponding value found in the structure
17176 pointed to by *buf*:17177 msg_perm.uid
17178 msg_perm.gid
17179 msg_perm.mode
17180 msg_qbytes17181 **IPC_SET** can only be executed by a process with appropriate privileges or that
17182 has an effective user ID equal to the value of **msg_perm.cuid** or
17183 **msg_perm.uid** in the **msqid_ds** data structure associated with *msqid*. Only a
17184 process with appropriate privileges can raise the value of *msg_qbytes*.17185 **IPC_RMID** Remove the message queue identifier specified by *msqid* from the system and
17186 destroy the message queue and **msqid_ds** data structure associated with it.
17187 **IPC_RMD** can only be executed by a process with appropriate privileges or
17188 one that has an effective user ID equal to the value of **msg_perm.cuid** or
17189 **msg_perm.uid** in the **msqid_ds** data structure associated with *msqid*.17190 **RETURN VALUE**17191 Upon successful completion, *msgctl()* returns 0. Otherwise, it returns -1 and *errno* will be set to
17192 indicate the error.17193 **ERRORS**17194 The *msgctl()* function will fail if:17195 **[EACCES]** The argument *cmd* is **IPC_STAT** and the calling process does not have read
17196 permission, see Section 2.6 on page 36.17197 **[EINVAL]** The value of *msqid* is not a valid message queue identifier; or the value of *cmd*
17198 is not a valid command.17199 **[EPERM]** The argument *cmd* is **IPC_RMID** or **IPC_SET** and the effective user ID of the
17200 calling process is not equal to that of a process with appropriate privileges
17201 and it is not equal to the value of **msg_perm.cuid** or **msg_perm.uid** in the data
17202 structure associated with *msqid*.17203 **[EPERM]** The argument *cmd* is **IPC_SET**, an attempt is being made to increase to the
17204 value of *msg_qbytes*, and the effective user ID of the calling process does not
17205 have appropriate privileges.

17206 **EXAMPLES**

17207 None.

17208 **APPLICATION USAGE**

17209 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
 17210 Application developers who need to use IPC should design their applications so that modules
 17211 using the IPC routines described in Section 2.6 on page 36 can be easily modified to use the
 17212 alternative interfaces.

17213 **FUTURE DIRECTIONS**

17214 None.

17215 **SEE ALSO**

17216 *mq_close()*, *mq_getattr()*, *mq_notify()*, *mq_open()*, *mq_receive()*, *mq_send()*, *mq_setattr()*,
 17217 *mq_unlink()*, *msgget()*, *msgrcv()*, *msgsnd()*, **<sys/msg.h>**, Section 2.6 on page 36.

17218 **CHANGE HISTORY**

17219 First released in Issue 2.

17220 Derived from Issue 2 of the SVID.

17221 **Issue 4**

17222 The following changes are incorporated in this issue:

- 17223 • The interface is no longer marked as OPTIONAL FUNCTIONALITY.
- 17224 • Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the SYNOPSIS
 17225 section.
- 17226 • A FUTURE DIRECTIONS section is added warning application developers about migration
 17227 to IEEE 1003.4 interfaces for interprocess communication.
- 17228 • The [ENOSYS] error is removed from the ERRORS section.

17229 **Issue 5**

17230 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
 17231 DIRECTIONS to a new APPLICATION USAGE section.

17232 **NAME**

17233 msgget — get the message queue identifier

17234 **SYNOPSIS**

17235 EX #include <sys/msg.h>

17236 int msgget(key_t key, int msgflg);

17237

17238 **DESCRIPTION**17239 The *msgget()* function returns the message queue identifier associated with the argument *key*.17240 A message queue identifier, associated message queue and data structure, see <sys/msg.h>, are
17241 created for the argument *key* if one of the following is true:

- 17242 • The argument *key* is equal to IPC_PRIVATE.
- 17243 • The argument *key* does not already have a message queue identifier associated with it, and
17244 (*msgflg* & IPC_CREAT) is non-zero.

17245 Upon creation, the data structure associated with the new message queue identifier is initialised
17246 as follows:

- 17247 • *msg_perm.cuid*, *msg_perm.uid*, *msg_perm.cgid* and *msg_perm.gid* are set equal to the
17248 effective user ID and effective group ID, respectively, of the calling process.
- 17249 • The low-order 9 bits of *msg_perm.mode* are set equal to the low-order 9 bits of *msgflg*.
- 17250 • *msg_qnum*, *msg_lspid*, *msg_lrpid*, *msg_stime* and *msg_rtime* are set equal to 0.
- 17251 • *msg_ctime* is set equal to the current time.
- 17252 • *msg_qbytes* is set equal to the system limit.

17253 **RETURN VALUE**17254 Upon successful completion, *msgget()* returns a non-negative integer, namely a message queue
17255 identifier. Otherwise, it returns -1 and *errno* is set to indicate the error.17256 **ERRORS**17257 The *msgget()* function will fail if:

- | | | |
|-------------------------|----------|--|
| 17258
17259
17260 | [EACCES] | A message queue identifier exists for the argument <i>key</i> , but operation permission as specified by the low-order 9 bits of <i>msgflg</i> would not be granted, see Section 2.6 on page 36. |
| 17261
17262 | [EEXIST] | A message queue identifier exists for the argument <i>key</i> but ((<i>msgflg</i> & IPC_CREAT) && (<i>msgflg</i> & IPC_EXCL)) is non-zero. |
| 17263
17264 | [ENOENT] | A message queue identifier does not exist for the argument <i>key</i> and (<i>msgflg</i> & IPC_CREAT) is 0. |
| 17265
17266
17267 | [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. |

17268 **EXAMPLES**

17269 None.

17270 **APPLICATION USAGE**

17271 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
 17272 Application developers who need to use IPC should design their applications so that modules
 17273 using the IPC routines described in Section 2.6 on page 36 can be easily modified to use the

17274 alternative interfaces.

17275 FUTURE DIRECTIONS

17276 None.

17277 SEE ALSO

17278 *mq_close()*, *mq_getattr()*, *mq_notify()*, *mq_open()*, *mq_receive()*, *mq_send()*, *mq_setattr()*,
17279 *mq_unlink()*, *msgctl()*, *msgrcv()*, *msgsnd()*, **<sys/msg.h>**, Section 2.6 on page 36.

17280 CHANGE HISTORY

17281 First released in Issue 2.

17282 Derived from Issue 2 of the SVID.

17283 Issue 4

17284 The following changes are incorporated in this issue:

- 17285 • The interface is no longer marked as OPTIONAL FUNCTIONALITY.
- 17286 • Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the SYNOPSIS
17287 section.
- 17288 • The [ENOSYS] error is removed from the ERRORS section.

17289 Issue 5

17290 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
17291 DIRECTIONS to a new APPLICATION USAGE section.

17292 NAME

17293 msgrcv — message receive operation

17294 SYNOPSIS

17295 EX

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

```
17296 ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long int msgtyp,
17297               int msgflg);
17298
```

17299 DESCRIPTION

17300 The *msgrcv()* function reads a message from the queue associated with the message queue
 17301 identifier specified by *msqid* and places it in the user-defined buffer pointed to by *msgp*.

17302 The argument *msgp* points to a user-defined buffer that must contain first a field of type **long int**
 17303 that will specify the type of the message, and then a data portion that will hold the data bytes of
 17304 the message. The structure below is an example of what this user-defined buffer might look like:

```
17305     struct mymsg {
17306         long int    mtype;      /* message type */
17307         char        mtext[1];  /* message text */
17308     }
```

17309 The structure member **mtype** is the received message's type as specified by the sending process.

17310 The structure member **mtext** is the text of the message.

17311 The argument *msgsz* specifies the size in bytes of **mtext**. The received message is truncated to
 17312 *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & MSG_NOERROR) is non-zero. The truncated
 17313 part of the message is lost and no indication of the truncation is given to the calling process.

17314 If the value of *msgsz* is greater than {SSIZE_MAX}, the result is implementation-dependent.

17315 The argument *msgtyp* specifies the type of message requested as follows:

- 17316 • If *msgtyp* is 0, the first message on the queue is received.
- 17317 • If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.
- 17318 • If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the
 17319 absolute value of *msgtyp* is received.

17320 The argument *msgflg* specifies the action to be taken if a message of the desired type is not on the
 17321 queue. These are as follows:

- 17322 • If (*msgflg* & IPC_NOWAIT) is non-zero, the calling thread will return immediately with a
 17323 return value of -1 and *errno* set to [ENOMSG].
- 17324 • If (*msgflg* & IPC_NOWAIT) is 0, the calling thread will suspend execution until one of the
 17325 following occurs:
 - 17326 — A message of the desired type is placed on the queue.
 - 17327 — The message queue identifier *msqid* is removed from the system; when this occurs, *errno* is
 17328 set equal to [EIDRM] and -1 is returned.
 - 17329 — The calling thread receives a signal that is to be caught; in this case a message is not
 17330 received and the calling thread resumes execution in the manner prescribed in *sigaction()*.

17331 Upon successful completion, the following actions are taken with respect to the data structure
 17332 associated with *msqid*:

- 17333 • **msg_qnum** is decremented by 1.
- 17334 • **msg_lrp**id is set equal to the process ID of the calling process.
- 17335 • **msg_rtime** is set equal to the current time.

17336 RETURN VALUE

17337 Upon successful completion, *msgrcv()* returns a value equal to the number of bytes actually
 17338 placed into the buffer *mtext*. Otherwise, no message will be received, *msgrcv()* will return
 17339 (**ssize_t**)−1 and *errno* will be set to indicate the error.

17340 ERRORS

17341 The *msgrcv()* function will fail if:

- 17342 [E2BIG] The value of **mtext** is greater than *msgsz* and (*msgflg* & MSG_NOERROR) is 0.
- 17343 [EACCES] Operation permission is denied to the calling process. See Section 2.6 on page
 17344 36.
- 17345 [EIDRM] The message queue identifier *msqid* is removed from the system.
- 17346 [EINTR] The *msgrcv()* function was interrupted by a signal.
- 17347 [EINVAL] *msqid* is not a valid message queue identifier; or the value of *msgsz* is less than
 17348 0.
- 17349 [ENOMSG] The queue does not contain a message of the desired type and (*msgflg* &
 17350 IPC_NOWAIT) is non-zero.

17351 EXAMPLES

17352 None.

17353 APPLICATION USAGE

17354 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
 17355 Application developers who need to use IPC should design their applications so that modules
 17356 using the IPC routines described in Section 2.6 on page 36 can be easily modified to use the
 17357 alternative interfaces.

17358 FUTURE DIRECTIONS

17359 None.

17360 SEE ALSO

17361 *mq_close()*, *mq_getattr()*, *mq_notify()*, *mq_open()*, *mq_receive()*, *mq_send()*, *mq_setattr()*,
 17362 *mq_unlink()*, *msgctl()*, *msgget()*, *msgsnd()*, *sigaction()*, <sys/msg.h>, Section 2.6 on page 36.

17363 CHANGE HISTORY

17364 First released in Issue 2.

17365 Derived from Issue 2 of the SVID.

17366 Issue 4

17367 The following changes are incorporated in this issue:

- 17368 • The interface is no longer marked as OPTIONAL FUNCTIONALITY.
- 17369 • Inclusion of the <sys/types.h> and <sys/ipc.h> headers is removed from the SYNOPSIS
 17370 section.
- 17371 • The [ENOSYS] error is removed from the ERRORS section.

- | | | |
|-------|---|--|
| 17372 | • A FUTURE DIRECTIONS section is added warning application developers about migration | |
| 17373 | to IEEE 1003.4 interfaces for interprocess communication. | |
| 17374 | Issue 5 | |
| 17375 | The type of the return value is changed from int to ssize_t , and a warning is added to the | |
| 17376 | DESCRIPTION about values of <i>msgsz</i> larger the {SSIZE_MAX}. | |
| 17377 | The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE | |
| 17378 | DIRECTIONS to the APPLICATION USAGE section. | |

17379 NAME

17380 msgsnd — message send operation

17381 SYNOPSIS

17382 EX

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

17383

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

17384

17385 DESCRIPTION

17386 The *msgsnd()* function is used to send a message to the queue associated with the message
17387 queue identifier specified by *msqid*.17388 The argument *msgp* points to a user-defined buffer that must contain first a field of type **long int**
17389 that will specify the type of the message, and then a data portion that will hold the data bytes of
17390 the message. The structure below is an example of what this user-defined buffer might look like:17391

```
struct mymsg {  
17392     long int    mtype;        /* message type */  
17393     char        mtext[1];    /* message text */  
17394 }
```

17395 The structure member **mtype** is a non-zero positive type **long int** that can be used by the
17396 receiving process for message selection.17397 The structure member **mtext** is any text of length *msgsz* bytes. The argument *msgsz* can range
17398 from 0 to a system-imposed maximum.17399 The argument *msgflg* specifies the action to be taken if one or more of the following are true:

- 17400
- 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.

17403 These actions are as follows:

- 17404
- If (*msgflg* & **IPC_NOWAIT**) is non-zero, the message will not be sent and the calling thread
17405 will return immediately.
 - If (*msgflg* & **IPC_NOWAIT**) is 0, the calling thread will suspend execution until one of the
17406 following occurs:
17407
 - The condition responsible for the suspension no longer exists, in which case the message
17408 is sent.
 - The message queue identifier *msqid* is removed from the system; when this occurs, *errno* is
17410 set equal to **[EIDRM]** and **-1** is returned.
 - The calling thread receives a signal that is to be caught; in this case the message is not
17412 sent and the calling thread resumes execution in the manner prescribed in *sigaction()*.

17414 Upon successful completion, the following actions are taken with respect to the data structure
17415 associated with *msqid*, see **<sys/msg.h>**:

- 17416
- **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.

17419 **RETURN VALUE**

17420 Upon successful completion, *msgsnd()* returns 0. Otherwise, no message will be sent, *msgsnd()*
 17421 will return -1 and *errno* will be set to indicate the error.

17422 **ERRORS**

17423 The *msgsnd()* function will fail if:

- | | | |
|-------------------------|----------|---|
| 17424
17425 | [EACCES] | Operation permission is denied to the calling process. See Section 2.6 on page 36. |
| 17426
17427 | [EAGAIN] | The message cannot be sent for one of the reasons cited above and (<i>msgflg</i> & <i>IPC_NOWAIT</i>) is non-zero. |
| 17428 | [EIDRM] | The message queue identifier <i>msgid</i> is removed from the system. |
| 17429 | [EINTR] | The <i>msgsnd()</i> function was interrupted by a signal. |
| 17430
17431
17432 | [EINVAL] | The value of <i>msgid</i> is not a valid message queue identifier, or the value of mtype is less than 1; or the value of <i>msgsz</i> is less than 0 or greater than the system-imposed limit. |

17433 **EXAMPLES**

17434 None.

17435 **APPLICATION USAGE**

17436 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
 17437 Application developers who need to use IPC should design their applications so that modules
 17438 using the IPC routines described in Section 2.6 on page 36 can be easily modified to use the
 17439 alternative interfaces.

17440 **FUTURE DIRECTIONS**

17441 None.

17442 **SEE ALSO**

17443 *mq_close()*, *mq_getattr()*, *mq_notify()*, *mq_open()*, *mq_receive()*, *mq_send()*, *mq_setattr()*,
 17444 *mq_unlink()*, *msgctl()*, *msgget()*, *msgrcv()*, *sigaction()*, <**sys/msg.h**>, Section 2.6 on page 36.

17445 **CHANGE HISTORY**

17446 First released in Issue 2.

17447 Derived from Issue 2 of the SVID.

17448 **Issue 4**

17449 The following changes are incorporated in this issue:

- | | |
|----------------|---|
| 17450 | • The interface is no longer marked as OPTIONAL FUNCTIONALITY. |
| 17451
17452 | • Inclusion of the < sys/types.h > and < sys/ipc.h > headers is removed from the SYNOPSIS section. Also the type of argument <i>msgp</i> is changed from void* to const void* . |
| 17453 | • In the DESCRIPTION, the example of a message buffer is changed: |
| 17454 | — explicitly to define the first member as being of type long int |
| 17455 | — to define the size of the message array <i>mtext</i> . |
| 17456 | • The [ENOSYS] error is removed from the ERRORS section. |
| 17457
17458 | • A FUTURE DIRECTIONS section is added warning application developers about migration to IEEE 1003.4 interfaces for interprocess communication. |

17459 **Issue 5**

17460 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
17461 DIRECTIONS to a new APPLICATION USAGE section.

17462 **NAME**

17463 msync — synchronise memory with physical storage

17464 **SYNOPSIS**

17465 #include <sys/mman.h>

17466 int msync(void *addr, size_t len, int flags);

17467 **DESCRIPTION**

17468 The *msync()* function writes all modified data to permanent storage locations, if any, in those
 17469 whole pages containing any part of the address space of the process starting at address *addr* and
 17470 continuing for *len* bytes. If no such storage exists, *msync()* need not have any effect. If
 17471 requested, the *msync()* function then invalidates cached copies of data.

17472 EX The implementation will require that *addr* be a multiple of the page size as returned by *sysconf()*.

17473 For mappings to files, the *msync()* function ensures that all write operations are completed as
 17474 defined for synchronised I/O data integrity completion. It is unspecified whether the
 17475 implementation also writes out other file attributes. When the *msync()* function is called on
 17476 MAP_PRIVATE mappings, any modified data will not be written to the underlying object and
 17477 will not cause such data to be made visible to other processes. It is unspecified whether data in
 17478 RT MAP_PRIVATE mappings has any permanent storage locations. The effect of *msync()* on shared
 17479 memory objects is unspecified.

17480 The *flags* argument is constructed from the bitwise inclusive OR of one or more of the following
 17481 flags defined in the header <sys/mman.h>:

17482

17483

17484

17485

17486

Symbolic Constant	Description
MS_ASYNC	Perform asynchronous writes.
MS_SYNC	Perform synchronous writes.
MS_INVALIDATE	Invalidate cached data.

17487 When MS_ASYNC is specified, *msync()* returns immediately once all the write operations are
 17488 initiated or queued for servicing; when MS_SYNC is specified, *msync()* will not return until all
 17489 write operations are completed as defined for synchronised I/O data integrity completion.
 17490 Either MS_ASYNC or MS_SYNC is specified, but not both.

17491 When MS_INVALIDATE is specified, *msync()* invalidates all cached copies of mapped data that
 17492 are inconsistent with the permanent storage locations such that subsequent references obtain
 17493 data that was consistent with the permanent storage locations sometime between the call to
 17494 *msync()* and the first subsequent memory reference to the data.

17495 The behaviour of this function is unspecified if the mapping was not established by a call to
 17496 *mmap()*.

17497 If *msync()* causes any write to a file, the file's *st_ctime* and *st_mtime* fields are marked for update.

17498 **RETURN VALUE**

17499 Upon successful completion, *msync()* returns 0. Otherwise, it returns -1 and sets *errno* to
 17500 indicate the error.

17501 **ERRORS**

17502 The *msync()* function will fail if:

17503 RT [EBUSY] Some or all of the addresses in the range starting at *addr* and continuing for *len*
 17504 bytes are locked, and MS_INVALIDATE is specified.

17505	[EINVAL]	The value in <i>flags</i> is invalid.	
17506 EX	[EINVAL]	The value of <i>addr</i> is not a multiple of the page size, {PAGESIZE}.	
17507	[ENOMEM]	The addresses in the range starting at <i>addr</i> and continuing for <i>len</i> bytes are	
17508		outside the range allowed for the address space of a process or specify one or	
17509		more pages that are not mapped.	
17510	EXAMPLES		
17511	None.		
17512	APPLICATION USAGE		
17513	The <i>msync()</i> function should be used by programs that require a memory object to be in a		
17514	known state, for example in building transaction facilities.		
17515	Normal system activity can cause pages to be written to disk. Therefore, there are no guarantees		
17516	that <i>msync()</i> is the only control over when pages are or are not written to disk.		
17517	The second form of [EINVAL] above is marked EX because it is defined as an optional error in		
17518	the POSIX Realtime Extension.		
17519	FUTURE DIRECTIONS		
17520	None.		
17521	SEE ALSO		
17522	<i>mmap()</i> , <i>sysconf()</i> , <sys/mman.h>.		
17523	CHANGE HISTORY		
17524	First released in Issue 4, Version 2.		
17525	Issue 5		
17526	Moved from X/OPEN UNIX extension to BASE and aligned with <i>msync()</i> in the POSIX Realtime		
17527	Extension. Specifically, the DESCRIPTION is extensively reworded, [EBUSY] and a new form of		
17528	[EINVAL] are added to the mandatory errors.		

17529 **NAME**

17530 munlock — unlock a range of process address space

17531 **SYNOPSIS**

17532 RT #include <sys/mman.h>

17533 int munlock(const void * *addr*, size_t *len*);

17534

17535 **DESCRIPTION**17536 Refer to *mlock()*.17537 **CHANGE HISTORY**

17538 First released in Issue 5.

17539 Included for alignment with the POSIX Realtime Extension.

17540 **NAME**

17541 munlockall — unlock the address space of a process

17542 **SYNOPSIS**

17543 RT #include <sys/mman.h>

17544 int munlockall(void);

17545

17546 **DESCRIPTION**17547 Refer to *mlockall()*.17548 **CHANGE HISTORY**

17549 First released in Issue 5.

17550 Included for alignment with the POSIX Realtime Extension.

17551 **NAME**

17552 munmap — unmap pages of memory

17553 **SYNOPSIS**

17554 #include <sys/mman.h>

17555 int munmap(void *addr, size_t len);

17556 **DESCRIPTION**

17557 The function *munmap()* removes any mappings for those entire pages containing any part of the
 17558 address space of the process starting at *addr* and continuing for *len* bytes. Further references to
 17559 these pages result in the generation of a SIGSEGV signal to the process. If there are no mappings
 17560 in the specified address range, then *munmap()* has no effect.

17561 EX The implementation will require that *addr* be a multiple of the page size {PAGESIZE}.

17562 If a mapping to be removed was private, any modifications made in this address range will be
 17563 discarded.

17564 RT Any memory locks (see *mlock()* and *mlockall()*) associated with this address range will be
 17565 removed, as if by an appropriate call to *munlock()*.

17566 The behaviour of this function is unspecified if the mapping was not established by a call to
 17567 *mmap()*.

17568 **RETURN VALUE**

17569 Upon successful completion, *munmap()* returns 0. Otherwise, it returns -1 and sets *errno* to
 17570 indicate the error.

17571 **ERRORS**17572 The *munmap()* function will fail if:

17573 [EINVAL] Addresses in the range [*addr*, *addr* + *len*) are outside the valid range for the
 17574 address space of a process.

17575 EX [EINVAL] The *len* argument is 0.

17576 EX [EINVAL] The *addr* argument is not a multiple of the page size as returned by *sysconf()*.

17577 **EXAMPLES**

17578 None.

17579 **APPLICATION USAGE**

17580 The third form of EINVAL above is marked EX because it is defined as an optional error in the
 17581 POSIX Realtime Extension.

17582 **FUTURE DIRECTIONS**

17583 None.

17584 **SEE ALSO**17585 *mmap()*, *sysconf()*, <signal.h>, <sys/mman.h>.17586 **CHANGE HISTORY**

17587 First released in Issue 4, Version 2.

17588 **Issue 5**

17589 Moved from X/OPEN UNIX extension to BASE and aligned with *munmap()* in the POSIX
 17590 Realtime Extension. Specifically, the DESCRIPTION is extensively reworded and the SIGBUS
 17591 error is no longer permitted to be generated.

17592 **NAME**17593 nanosleep — high resolution sleep (**REALTIME**)17594 **SYNOPSIS**

17595 RT #include <time.h>

17596 int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);

17597

17598 **DESCRIPTION**

17599 The *nanosleep()* function causes the current thread to be suspended from execution until either
 17600 the time interval specified by the *rqtp* argument has elapsed or a signal is delivered to the calling
 17601 thread and its action is to invoke a signal-catching function or to terminate the process. The
 17602 suspension time may be longer than requested because the argument value is rounded up to an
 17603 integer multiple of the sleep resolution or because of the scheduling of other activity by the
 17604 system. But, except for the case of being interrupted by a signal, the suspension time will not be
 17605 less than the time specified by *rqtp*, as measured by the system clock, **CLOCK_REALTIME**.

17606 The use of the *nanosleep()* function has no effect on the action or blockage of any signal.

17607 **RETURN VALUE**

17608 If the *nanosleep()* function returns because the requested time has elapsed, its return value is
 17609 zero.

17610 If the *nanosleep()* function returns because it has been interrupted by a signal, the function
 17611 returns a value of -1 and sets *errno* to indicate the interruption. If the *rmtp* argument is non-
 17612 NULL, the **timespec** structure referenced by it is updated to contain the amount of time
 17613 remaining in the interval (the requested time minus the time actually slept). If the *rmtp*
 17614 argument is NULL, the remaining time is not returned.

17615 If *nanosleep()* fails, it returns a value of -1 and sets *errno* to indicate the error.

17616 **ERRORS**

17617 The *nanosleep()* function will fail if:

17618	[EINTR]	The <i>nanosleep()</i> function was interrupted by a signal.
17619	[EINVAL]	The <i>rqtp</i> argument specified a nanosecond value less than zero or greater than
17620		or equal to 1000 million.
17621	[ENOSYS]	The <i>nanosleep()</i> function is not supported by this implementation.

17622 **EXAMPLES**

17623 None.

17624 **APPLICATION USAGE**

17625 None.

17626 **FUTURE DIRECTIONS**

17627 None.

17628 **SEE ALSO**

17629 *sleep()*, <time.h>.

17630 **CHANGE HISTORY**

17631 First released in Issue 5.

17632 Included for alignment with the POSIX Realtime Extension.

17633 **NAME**

17634 nextafter — next representable double-precision floating-point number

17635 **SYNOPSIS**

17636 EX #include <math.h>

17637 double nextafter(double x, double y);

17638

17639 **DESCRIPTION**

17640 The *nextafter()* function computes the next representable double-precision floating-point value
17641 following *x* in the direction of *y*. Thus, if *y* is less than *x*, *nextafter()* returns the largest
17642 representable floating-point number less than *x*.

17643 An application wishing to check for error situations should set *errno* to 0 before calling
17644 *nextafter()*. If *errno* is non-zero on return, or the value NaN is returned, an error has occurred.

17645 **RETURN VALUE**

17646 The *nextafter()* function returns the next representable double-precision floating-point value
17647 following *x* in the direction of *y*.

17648 If *x* or *y* is NaN, then *nextafter()* returns NaN and may set *errno* to [EDOM].

17649 If *x* is finite and the correct function value would overflow, HUGE_VAL is returned and *errno* is
17650 set to [ERANGE].

17651 **ERRORS**

17652 The *nextafter()* function will fail if:

17653 [ERANGE] The correct value would overflow.

17654 The *nextafter()* function may fail if:

17655 [EDOM] The *x* or *y* argument is NaN.

17656 **EXAMPLES**

17657 None.

17658 **APPLICATION USAGE**

17659 None.

17660 **FUTURE DIRECTIONS**

17661 None.

17662 **SEE ALSO**

17663 <math.h>.

17664 **CHANGE HISTORY**

17665 First released in Issue 4, Version 2.

17666 **Issue 5**

17667 Moved from X/OPEN UNIX extension to BASE.

17668 **NAME**

17669 nftw — walk a file tree

17670 **SYNOPSIS**17671 EX

```
#include <ftw.h>
```

```
17672 int nftw(const char *path, int (*fn)(const char *,
17673     const struct stat *, int, struct FTW *), int depth, int flags);
17674
```

17675 **DESCRIPTION**

17676 The *nftw()* function recursively descends the directory hierarchy rooted in *path*. The *nftw()*
 17677 function has a similar effect to *ftw()* except that it takes an additional argument *flags*, which is a
 17678 bitwise inclusive-OR of zero or more of the following flags:

17679 17680 17681	FTW_CHDIR	If set, <i>nftw()</i> will change the current working directory to each directory as it reports files in that directory. If clear, <i>nftw()</i> will not change the current working directory.
17682 17683 17684	FTW_DEPTH	If set, <i>nftw()</i> will report all files in a directory before reporting the directory itself. If clear, <i>nftw()</i> will report any directory before reporting the files in that directory.
17685 17686	FTW_MOUNT	If set, <i>nftw()</i> will only report files in the same file system as <i>path</i> . If clear, <i>nftw()</i> will report all files encountered during the walk.
17687 17688 17689	FTW_PHYS	If set, <i>nftw()</i> performs a physical walk and does not follow symbolic links. If clear, <i>nftw()</i> will follow links instead of reporting them, and will not report the same file twice.

17690 At each file it encounters, *nftw()* calls the user-supplied function *fn()* with four arguments:

- 17691 • The first argument is the pathname of the object.
- 17692 • The second argument is a pointer to the **stat** buffer containing information on the object.
- 17693 • The third argument is an integer giving additional information. Its value is one of the
 17694 following:

17695	FTW_F	The object is a file.
17696	FTW_D	The object is a directory.
17697 17698	FTW_DP	The object is a directory and subdirectories have been visited. (This condition will only occur if the FTW_DEPTH flag is included in <i>flags</i> .)
17699 17700	FTW_SL	The object is a symbolic link. (This condition will only occur if the FTW_PHYS flag is included in <i>flags</i> .)
17701 17702	FTW_SLN	The object is a symbolic link that does not name an existing file. (This condition will only occur if the FTW_PHYS flag is not included in <i>flags</i> .)
17703 17704	FTW_DNR	The object is a directory that cannot be read. The <i>fn()</i> function will not be called for any of its descendants.
17705 17706 17707	FTW_NS	The <i>stat()</i> function failed on the object because of lack of appropriate permission. The stat buffer passed to <i>fn()</i> is undefined. Failure of <i>stat()</i> for any other reason is considered an error and <i>nftw()</i> returns -1 .
- 17708 • The fourth argument is a pointer to an **FTW** structure. The value of **base** is the offset of the
 17709 object's filename in the pathname passed as the first argument to *fn()*. The value of **level**
 17710 indicates depth relative to the root of the walk, where the root level is 0.

17711 The argument *depth* sets the maximum number of file descriptors that will be used by *nftw()*
 17712 while traversing the file tree. At most one file descriptor will be used for each directory level.

17713 RETURN VALUE

17714 The *nftw()* function continues until the first of the following conditions occurs:

- 17715 • An invocation of *fn()* returns a non-zero value, in which case *nftw()* returns that value.
- 17716 • The *nftw()* function detects an error other than [EACCES] (see FTW_DNR and FTW_NS
 17717 above), in which case *nftw()* returns -1 and sets *errno* to indicate the error.
- 17718 • The tree is exhausted, in which case *nftw()* returns 0.

17719 ERRORS

17720 The *nftw()* function will fail if:

17721 [EACCES] Search permission is denied for any component of *path* or read permission is
 17722 denied for *path*, or *fn()* returns -1 and does not reset *errno*.

17723 [ENAMETOOLONG]

17724 The length of the *path* string exceeds {PATH_MAX}, or a pathname component
 17725 is longer than {NAME_MAX}.

17726 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

17727 [ENOTDIR] A component of *path* is not a directory.

17728 The *nftw()* function may fail if:

17729 [ELOOP] Too many symbolic links were encountered in resolving *path*.

17730 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

17731 [ENAMETOOLONG]

17732 Pathname resolution of a symbolic link produced an intermediate result
 17733 whose length exceeds {PATH_MAX}.

17734 [ENFILE] Too many files are currently open in the system.

17735 In addition, *errno* may be set if the function pointed to by *fn()* causes *errno* to be set.

17736 EXAMPLES

17737 None.

17738 APPLICATION USAGE

17739 None.

17740 FUTURE DIRECTIONS

17741 None.

17742 SEE ALSO

17743 *lstat()*, *opendir()*, *readdir()*, *stat()*, <ftw.h>.

17744 CHANGE HISTORY

17745 First released in Issue 4, Version 2.

17746 Issue 5

17747 Moved from X/OPEN UNIX extension to BASE.

17748 In the DESCRIPTION, the definition of the *depth* argument is clarified.

17749 **NAME**

17750 nice — change nice value of a process

17751 **SYNOPSIS**

17752 EX #include <unistd.h>

17753 int nice(int *incr*);

17754

17755 **DESCRIPTION**

17756 The *nice()* function adds the value of *incr* to the nice value of the calling process. A process' nice
17757 value is a non-negative number for which a more positive value results in less favourable
17758 scheduling.

17759 A maximum nice value of 2 * {NZERO} -1 and a minimum nice value of 0 are imposed by the
17760 system. Requests for values above or below these limits result in the nice value being set to the
17761 corresponding limit. Only a process with appropriate privileges can lower the nice value.

17762 RT Calling the *nice()* function has no effect on the priority of processes or threads with policy
17763 SCHED_FIFO or SCHED_RR. The effect on processes or threads with other scheduling policies
17764 is implementation-dependent.

17765 The nice value set with *nice()* is applied to the process. If the process is multi-threaded, the nice
17766 value affects all system scope threads in the process.

17767 **RETURN VALUE**

17768 Upon successful completion, *nice()* returns the new nice value minus {NZERO}. Otherwise, -1
17769 is returned, the process' nice value is not changed, and *errno* is set to indicate the error.

17770 **ERRORS**

17771 The *nice()* function will fail if:

17772 [EPERM] The *incr* argument is negative and the calling process does not have
17773 appropriate privileges.

17774 **EXAMPLES**

17775 None.

17776 **APPLICATION USAGE**

17777 As -1 is a permissible return value in a successful situation, an application wishing to check for
17778 error situations should set *errno* to 0, then call *nice()*, and if it returns -1, check to see if *errno* is
17779 non-zero.

17780 **FUTURE DIRECTIONS**

17781 None.

17782 **SEE ALSO**

17783 <limits.h>, <unistd.h>.

17784 **CHANGE HISTORY**

17785 First released in Issue 1.

17786 Derived from Issue 1 of the SVID.

17787 **Issue 4**

17788 The following changes are incorporated in this issue:

- 17789 • The <unistd.h> header is added to the SYNOPSIS section.
- 17790 • A statement is added to the DESCRIPTION indicating that the nice value can only be
- 17791 lowered by a process with appropriate privileges.

17792 **Issue 4, Version 2**

17793 The RETURN VALUE section is updated for X/OPEN UNIX conformance to define that the
17794 process' *nice* value is not changed if an error is detected.

17795 **Issue 5**

17796 A statement is added to the description indicating the effects of this function on the different
17797 scheduling policies and multi-threaded processes.

17798 **NAME**

17799 nl_langinfo — language information

17800 **SYNOPSIS**

17801 EX #include <langinfo.h>

17802 char *nl_langinfo(nl_item item);

17803

17804 **DESCRIPTION**

17805 The *nl_langinfo()* function returns a pointer to a string containing information relevant to the
 17806 particular language or cultural area defined in the program's locale (see <langinfo.h>). The
 17807 manifest constant names and values of *item* are defined in <langinfo.h>. For example:

17808 nl_langinfo (ABDAY_1)

17809 would return a pointer to the string “Dom” if the identified language was Portuguese, and
 17810 “Sun” if the identified language was English.

17811 Calls to *setlocale()* with a category corresponding to the category of *item* (see <langinfo.h>), or to
 17812 the category LC_ALL, may overwrite the array pointed to by the return value.

17813 This interface need not be reentrant.

17814 **RETURN VALUE**

17815 In a locale where *langinfo* data is not defined, *nl_langinfo()* returns a pointer to the corresponding
 17816 string in the POSIX locale. In all locales, *nl_langinfo()* returns a pointer to an empty string if *item*
 17817 contains an invalid setting.

17818 This pointer may point to static data that may be overwritten on the next call.

17819 **ERRORS**

17820 No errors are defined.

17821 **EXAMPLES**

17822 None.

17823 **APPLICATION USAGE**

17824 The array pointed to by the return value should not be modified by the program, but may be
 17825 modified by further calls to *nl_langinfo()*.

17826 **FUTURE DIRECTIONS**

17827 None.

17828 **SEE ALSO**17829 *setlocale()*, <langinfo.h>, <nl_types.h>, the XBD specification, Chapter 5, Locale.17830 **CHANGE HISTORY**

17831 First released in Issue 2.

17832 **Issue 4**

17833 The <nl_types.h> header is removed from the SYNOPSIS section.

17834 **Issue 5**

17835 The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section.

17836 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

17837 **NAME**

17838 nrand48 — generate uniformly distributed pseudo-random non-negative long integers

17839 **SYNOPSIS**

17840 EX #include <stdlib.h>

17841 long int nrand48(unsigned short int xsubi[3]);

17842

17843 **DESCRIPTION**

17844 Refer to *drand48()*.

17845 **CHANGE HISTORY**

17846 First released in Issue 1.

17847 Derived from Issue 1 of the SVID.

17848 **Issue 4**

17849 The declaration of *xsubi* is expanded to **unsigned short int**.

17850 NAME

17851 open — open a file

17852 SYNOPSIS

17853 OH #include <sys/types.h>

17854 #include <sys/stat.h>

17855 #include <fcntl.h>

17856 int open(const char *path, int oflag, ...);

17857 DESCRIPTION

17858 The *open()* function establishes the connection between a file and a file descriptor. It creates an
 17859 open file description that refers to a file and a file descriptor that refers to that open file
 17860 description. The file descriptor is used by other I/O functions to refer to that file. The *path*
 17861 argument points to a pathname naming the file.

17862 The *open()* function will return a file descriptor for the named file that is the lowest file
 17863 descriptor not currently open for that process. The open file description is new, and therefore
 17864 the file descriptor does not share it with any other process in the system. The FD_CLOEXEC file
 17865 descriptor flag associated with the new file descriptor will be cleared.

17866 The file offset used to mark the current position within the file is set to the beginning of the file.

17867 The file status flags and file access modes of the open file description will be set according to the
 17868 value of *oflag*.

17869 Values for *oflag* are constructed by a bitwise-inclusive-OR of flags from the following list,
 17870 defined in <fcntl.h>. Applications must specify exactly one of the first three values (file access
 17871 modes) below in the value of *oflag*:

17872 O_RDONLY Open for reading only.

17873 O_WRONLY Open for writing only.

17874 O_RDWR Open for reading and writing. The result is undefined if this flag is applied to
 17875 a FIFO.

17876 Any combination of the following may be used:

17877 O_APPEND If set, the file offset will be set to the end of the file prior to each write.

17878 O_CREAT If the file exists, this flag has no effect except as noted under O_EXCL below.
 17879 Otherwise, the file is created; the user ID of the file is set to the effective user
 17880 FIPS ID of the process; the group ID of the file is set to the group ID of the file's
 17881 parent directory or to the effective group ID of the process; and the access
 17882 permission bits (see <sys/stat.h>) of the file mode are set to the value of the
 17883 third argument taken as type **mode_t** modified as follows: a bitwise-AND is
 17884 performed on the file-mode bits and the corresponding bits in the complement
 17885 of the process' file mode creation mask. Thus, all bits in the file mode whose
 17886 corresponding bit in the file mode creation mask is set are cleared. When bits
 17887 other than the file permission bits are set, the effect is unspecified. The third
 17888 argument does not affect whether the file is open for reading, writing or for
 17889 both.

17890 RT O_DSYNC Write I/O operations on the file descriptor complete as defined by
 17891 synchronised I/O data integrity completion

17892 O_EXCL If O_CREAT and O_EXCL are set, *open()* will fail if the file exists. The check
 17893 for the existence of the file and the creation of the file if it does not exist will be
 17894 atomic with respect to other processes executing *open()* naming the same

17895		filename in the same directory with O_EXCL and O_CREAT set. If O_CREAT
17896		is not set, the effect is undefined.
17897	O_NOCTTY	If set and <i>path</i> identifies a terminal device, <i>open()</i> will not cause the terminal
17898		device to become the controlling terminal for the process.
17899	O_NONBLOCK	When opening a FIFO with O_RDONLY or O_WRONLY set:
17900		If O_NONBLOCK is set:
17901		An <i>open()</i> for reading only will return without delay. An <i>open()</i> for
17902		writing only will return an error if no process currently has the file open
17903		for reading.
17904		If O_NONBLOCK is clear:
17905		An <i>open()</i> for reading only will block the calling thread until a thread
17906		opens the file for writing. An <i>open()</i> for writing only will block the calling
17907		thread until a thread opens the file for reading.
17908		When opening a block special or character special file that supports non-
17909		blocking opens:
17910		If O_NONBLOCK is set:
17911		The <i>open()</i> function will return without blocking for the device to be ready
17912		or available. Subsequent behaviour of the device is device-specific.
17913		If O_NONBLOCK is clear:
17914		The <i>open()</i> function will block the calling thread until the device is ready
17915		or available before returning.
17916		Otherwise, the behaviour of O_NONBLOCK is unspecified.
17917 RT	O_RSYNC	Read I/O operations on the file descriptor complete at the same level of
17918		integrity as specified by the O_DSYNC and O_SYNC flags. If both O_DSYNC
17919		and O_RSYNC are set in <i>oflag</i> , all I/O operations on the file descriptor
17920		complete as defined by synchronised I/O data integrity completion. If both
17921		O_SYNC and O_RSYNC are set in flags, all I/O operations on the file
17922		descriptor complete as defined by synchronised I/O file integrity completion.
17923	O_SYNC	Write I/O operations on the file descriptor complete as defined by
17924		synchronised I/O file integrity completion.
17925	O_TRUNC	If the file exists and is a regular file, and the file is successfully opened
17926		O_RDWR or O_WRONLY, its length is truncated to 0 and the mode and
17927		owner are unchanged. It will have no effect on FIFO special files or terminal
17928		device files. Its effect on other file types is implementation-dependent. The
17929		result of using O_TRUNC with O_RDONLY is undefined.
17930		If O_CREAT is set and the file did not previously exist, upon successful completion, <i>open()</i> will
17931		mark for update the <i>st_atime</i> , <i>st_ctime</i> and <i>st_mtime</i> fields of the file and the <i>st_ctime</i> and <i>st_mtime</i>
17932		fields of the parent directory.
17933		If O_TRUNC is set and the file did previously exist, upon successful completion, <i>open()</i> will
17934		mark for update the <i>st_ctime</i> and <i>st_mtime</i> fields of the file.
17935 RT		If both the O_SYNC and O_DSYNC flags are set, the effect is as if only the O_SYNC flag was set.
17936		

17937 EX	If <i>path</i> refers to a STREAMS file, <i>oflag</i> 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.	
17938		
17939		
17940		
17941		
17942	If <i>path</i> names the master side of a pseudo-terminal device, then it is unspecified whether <i>open()</i> locks the slave side so that it cannot be opened. Portable applications must call <i>unlockpt()</i> before opening the slave side.	
17943		
17944		
17945	The largest value that can be represented correctly in an object of type off_t will be established as the offset maximum in the open file description.	
17946		
17947	RETURN VALUE	
17948	Upon successful completion, the function will open the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, -1 is returned and <i>errno</i> is set to indicate the error. No files will be created or modified if the function returns -1.	
17949		
17950		
17951	ERRORS	
17952	The <i>open()</i> function will fail if:	
17953	[EACCES]	Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by <i>oflag</i> 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.
17954		
17955		
17956		
17957	[EEXIST]	O_CREAT and O_EXCL are set, and the named file exists.
17958	[EINTR]	A signal was caught during <i>open()</i> .
17959 RT	[EINVAL]	The implementation does not support synchronised I/O for this file.
17960 EX	[EIO]	The <i>path</i> argument names a STREAMS file and a hangup or error occurred during the <i>open()</i> .
17961		
17962	[EISDIR]	The named file is a directory and <i>oflag</i> includes O_WRONLY or O_RDWR.
17963 EX	[ELOOP]	Too many symbolic links were encountered in resolving <i>path</i> .
17964	[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
17965	[ENAMETOOLONG]	
17966 FIPS		The length of the <i>path</i> argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
17967		
17968	[ENFILE]	The maximum allowable number of files is currently open in the system.
17969	[ENOENT]	O_CREAT is not set and the named file does not exist; or O_CREAT is set and either the path prefix does not exist or the <i>path</i> argument points to an empty string.
17970		
17971		
17972 EX	[ENOSR]	The <i>path</i> argument names a STREAMS-based file and the system is unable to allocate a STREAM.
17973		
17974	[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.
17975		
17976	[ENOTDIR]	A component of the path prefix is not a directory.

17977	[ENXIO]	O_NONBLOCK is set, the named file is a FIFO, O_WRONLY is set and no process has the file open for reading.	
17978			
17979 EX	[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.	
17980			
17981 EX	[EOVERFLOW]	The named file is a regular file and the size of the file cannot be represented correctly in an object of type <code>off_t</code> .	
17982			
17983	[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 <i>oflag</i> argument.	
17984			
17985			
17986		The <i>open()</i> function may fail if:	
17987 EX	[EAGAIN]	The <i>path</i> argument names the slave side of a pseudo-terminal device that is locked.	
17988			
17989 EX	[EINVAL]	The value of the <i>oflag</i> argument is not valid.	
17990 EX	[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.	
17991			
17992			
17993 EX	[ENOMEM]	The <i>path</i> argument names a STREAMS file and the system is unable to allocate resources.	
17994			
17995 EX	[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and <i>oflag</i> is O_WRONLY or O_RDWR.	
17996			
17997	EXAMPLES		
17998	None.		
17999	APPLICATION USAGE		
18000	None.		
18001	FUTURE DIRECTIONS		
18002	None.		
18003	SEE ALSO		
18004	<i>chmod()</i> , <i>close()</i> , <i>creat()</i> , <i>dup()</i> , <i>fcntl()</i> , <i>lseek()</i> , <i>read()</i> , <i>umask()</i> , <i>unlockpt()</i> , <i>write()</i> , <code><fcntl.h></code> ,		
18005	<code><sys/stat.h></code> , <code><sys/types.h></code> .		
18006	CHANGE HISTORY		
18007	First released in Issue 1.		
18008	Derived from Issue 1 of the SVID.		
18009	Issue 4		
18010	The following changes are incorporated for alignment with the ISO POSIX-1 standard:		
18011	<ul style="list-style-type: none"> The type of argument <i>path</i> is changed from char * to const char *. 		
18012	<ul style="list-style-type: none"> Various wording changes are made to the DESCRIPTION to improve clarity and to align the text with the ISO POSIX-1 standard. 		
18013			
18014	The following changes are incorporated for alignment with the FIPS requirements:		
18015	<ul style="list-style-type: none"> In the DESCRIPTION, the description of O_CREAT is amended and the relevant part marked as an extension. 		
18016			
18017	<ul style="list-style-type: none"> In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a pathname component is larger than {NAME_MAX} is now defined as mandatory and marked 		
18018			

18019 as an extension.

18020 Other changes are incorporated as follows:

18021 • The `<sys/types.h>` and `<sys/stat.h>` headers are now marked as optional (OH); these headers
18022 do not need to be included on XSI-conformant systems.

18023 • `O_NDELAY` is removed from the list of *oflag* values (this flag was marked WITHDRAWN in
18024 Issue 3).

18025 • The `[ENXIO]` error (for the condition where the file is a character or block special file and the
18026 associated device does not exist) and the `[EINVAL]` error are marked as extensions.

18027 **Issue 4, Version 2**

18028 The following changes are incorporated for X/OPEN UNIX conformance:

18029 • The `DESCRIPTION` is updated to define the use of open flags with STREAMS files, and to
18030 identify special considerations when opening the master side of a pseudo-terminal.

18031 • The `[EIO]`, `[ELOOP]` and `[ENOSR]` errors are added to the `ERRORS` section as mandatory
18032 errors; `[EAGAIN]`, `[ENAMETOOLONG]` and `[ENOMEM]` are added as optional errors.

18033 **Issue 5**

18034 The `DESCRIPTION` is updated for alignment with the POSIX Realtime Extension and the POSIX
18035 Threads Extension.

18036 Large File Summit extensions added.

18037 **NAME**18038 `opendir` — open a directory18039 **SYNOPSIS**18040 OH `#include <sys/types.h>`18041 `#include <dirent.h>`18042 `DIR *opendir(const char *dirname);`18043 **DESCRIPTION**

18044 The `opendir()` function opens a directory stream corresponding to the directory named by the
 18045 `dirname` argument. The directory stream is positioned at the first entry. If the type **DIR**, is
 18046 implemented using a file descriptor, applications will only be able to open up to a total of
 18047 {OPEN_MAX} files and directories. A successful call to any of the `exec` functions will close any
 18048 directory streams that are open in the calling process.

18049 **RETURN VALUE**

18050 Upon successful completion, `opendir()` returns a pointer to an object of type **DIR**. Otherwise, a
 18051 null pointer is returned and `errno` is set to indicate the error.

18052 **ERRORS**18053 The `opendir()` function will fail if:

18054 [EACCES] Search permission is denied for the component of the path prefix of `dirname` or
 18055 read permission is denied for `dirname`.

18056 EX [ELOOP] Too many symbolic links were encountered in resolving *path*.

18057 FIPS [ENAMETOOLONG]

18058 The length of the `dirname` argument exceeds {PATH_MAX}, or a pathname
 18059 component is longer than {NAME_MAX}.

18060 [ENOENT] A component of `dirname` does not name an existing directory or `dirname` is an
 18061 empty string.

18062 [ENOTDIR] A component of `dirname` is not a directory.

18063 The `opendir()` function may fail if:

18064 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

18065 EX [ENAMETOOLONG]

18066 Pathname resolution of a symbolic link produced an intermediate result
 18067 whose length exceeds {PATH_MAX}.

18068 [ENFILE] Too many files are currently open in the system.

18069 **EXAMPLES**

18070 None.

18071 **APPLICATION USAGE**

18072 The `opendir()` function should be used in conjunction with `readdir()`, `closedir()` and `rewinddir()` to
 18073 examine the contents of the directory (see the EXAMPLES section in `readdir()`). This method is
 18074 recommended for portability.

18075 **FUTURE DIRECTIONS**

18076 None.

18077 **SEE ALSO**18078 `closedir()`, `lstat()`, `readdir()`, `rewinddir()`, `symlink()`, `<dirent.h>`, `<limits.h>`, `<sys/types.h>`.

18079 **CHANGE HISTORY**

18080 First released in Issue 2.

18081 **Issue 4**

18082 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- 18083 • The type of argument *dirname* is changed from **char *** to **const char ***.
- 18084 • The generation of an [ENOENT] error when *dirname* points to an empty string is made
- 18085 mandatory.

18086 The following change is incorporated for alignment with the FIPS requirements:

- 18087 • In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a
- 18088 pathname component is larger than {NAME_MAX} is now defined as mandatory and marked
- 18089 as an extension.

18090 Other changes are incorporated as follows:

- 18091 • The **<sys/types.h>** header is now marked as optional (OH); this header need not be included
- 18092 on XSI-conformant systems.

- 18093 • In the DESCRIPTION, the following sentence is moved to the **XBD** specification:

18094 The type **DIR**, which is defined in **<dirent.h>**, represents a *directory stream*, which is an

18095 ordered sequence of all directory entries in a particular directory.

18096 **Issue 4, Version 2**

18097 The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- 18098 • It states that [ELOOP] will be returned if too many symbolic links are encountered during
- 18099 pathname resolution.
- 18100 • A second [ENAMETOOLONG] condition is defined that may report excessive length of an
- 18101 intermediate result of pathname resolution of a symbolic link.

18102 **NAME**

18103 openlog — open a connection to the logging facility

18104 **SYNOPSIS**

18105 EX #include <syslog.h>

18106 void openlog(const char **ident*, int *logopt*, int *facility*);

18107

18108 **DESCRIPTION**18109 Refer to *closelog()*.18110 **CHANGE HISTORY**

18111 First released in Issue 4, Version 2.

18112 **Issue 5**

18113 Moved from X/OPEN UNIX extension to BASE.

18114 **NAME**

18115 optarg, opterr, optind, optopt — options parsing variables

18116 **SYNOPSIS**

18117 #include <stdio.h>

18118 extern char *optarg;

18119 extern int opterr, optind, optopt;

18120 **DESCRIPTION**18121 Refer to *getopt()*.18122 **CHANGE HISTORY**

18123 First released in Issue 1.

18124 Originally derived from Issue 1 of the SVID.

18125 **Issue 4**18126 Entry derived from *getopt()* in Issue 3, with the following change:

- 18127
- Item *optopt* is added to the list of external data items.

18128 **NAME**

18129 pathconf — get configurable pathname variables

18130 **SYNOPSIS**

18131 #include <unistd.h>

18132 long int pathconf(const char *path, int name);

18133 **DESCRIPTION**18134 Refer to *fpathconf()*.18135 **CHANGE HISTORY**

18136 First released in Issue 3.

18137 Entry included for alignment with the POSIX.1-1988 standard.

18138 **Issue 4**

18139 The following changes have been made for alignment with the ISO POSIX-1 standard:

- 18140 • The type of argument *path* is changed from **char *** to **const char ***. Also the return value of
- 18141 both functions is changed from **long** to **long int**.
- 18142 • In the DESCRIPTION, the words “The behaviour is undefined if” have been replaced by “it
- 18143 is unspecified whether an implementation supports an association of the variable name with
- 18144 the specified file” in notes 2, 4 and 6.
- 18145 • In the RETURN VALUE section, errors associated with the use of *path* and *fildev*, when an
- 18146 implementation does not support the requested association, are now specified separately.
- 18147 • The requirement that *errno* be set to indicate the error is added.

18148 The following change is incorporated for alignment with the FIPS requirements:

- 18149 • In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a
- 18150 pathname component is larger than {NAME_MAX} is now defined as mandatory and marked
- 18151 as an extension.

18152 **Issue 4, Version 2**

18153 The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- 18154 • It states that [ELOOP] will be returned if too many symbolic links are encountered during
- 18155 pathname resolution.
- 18156 • A second [ENAMETOOLONG] condition is defined that may report excessive length of an
- 18157 intermediate result of pathname resolution of a symbolic link.

18158 **Issue 5**

18159 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

18160 Large File Summit extensions added.

18161 **NAME**

18162 `pause` — suspend the thread until signal is received

18163 **SYNOPSIS**

18164 `#include <unistd.h>`

18165 `int pause(void);`

18166 **DESCRIPTION**

18167 The `pause()` function suspends the calling thread until delivery of a signal whose action is either
18168 to execute a signal-catching function or to terminate the process.

18169 If the action is to terminate the process, `pause()` will not return.

18170 If the action is to execute a signal-catching function, `pause()` will return after the signal-catching
18171 function returns.

18172 **RETURN VALUE**

18173 Since `pause()` suspends thread execution indefinitely unless interrupted by a signal, there is no
18174 successful completion return value. A value of `-1` is returned and `errno` is set to indicate the
18175 error.

18176 **ERRORS**

18177 The `pause()` function will fail if:

18178 [EINTR] A signal is caught by the calling process and control is returned from the
18179 signal-catching function.

18180 **EXAMPLES**

18181 None.

18182 **APPLICATION USAGE**

18183 None.

18184 **FUTURE DIRECTIONS**

18185 None.

18186 **SEE ALSO**

18187 `sigsuspend()`, `<unistd.h>`.

18188 **CHANGE HISTORY**

18189 First released in Issue 1.

18190 Derived from Issue 1 of the SVID.

18191 **Issue 4**

18192 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 18193 • The argument list is explicitly defined as **void**.

18194 Other changes are incorporated as follows:

- 18195 • The `<unistd.h>` header is added to the SYNOPSIS section.
- 18196 • In the RETURN VALUE section, the text is expanded to indicate that process execution is
18197 suspended indefinitely “unless interrupted by a signal”.

18198 **Issue 5**

18199 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

18200 **NAME**

18201 pclose — close a pipe stream to or from a process

18202 **SYNOPSIS**

18203 #include <stdio.h>

18204 int pclose(FILE *stream);

18205 **DESCRIPTION**

18206 The *pclose()* function closes a stream that was opened by *popen()*, waits for the command to
 18207 terminate, and returns the termination status of the process that was running the command
 18208 language interpreter. However, if a call caused the termination status to be unavailable to
 18209 *pclose()*, then *pclose()* returns *-1* with *errno* set to [ECHILD] to report this situation; this can
 18210 happen if the application calls one of the following functions:

- 18211 • *wait()*
- 18212 • *waitpid()* with a *pid* argument less than or equal to 0 or equal to the process ID of the
 18213 command line interpreter
- 18214 • any other function not defined in this specification that could do one of the above.

18215 In any case, *pclose()* will not return before the child process created by *popen()* has terminated.

18216 If the command language interpreter cannot be executed, the child termination status returned
 18217 by *pclose()* will be as if the command language interpreter terminated using *exit(127)* or
 18218 *_exit(127)*.

18219 The *pclose()* function will not affect the termination status of any child of the calling process
 18220 other than the one created by *popen()* for the associated stream.

18221 If the argument *stream* to *pclose()* is not a pointer to a stream created by *popen()*, the result of
 18222 *pclose()* is undefined.

18223 **RETURN VALUE**

18224 Upon successful return, *pclose()* returns the termination status of the command language
 18225 interpreter. Otherwise, *pclose()* returns *-1* and sets *errno* to indicate the error.

18226 **ERRORS**

18227 The *pclose()* function will fail if:

- 18228 [ECHILD] The status of the child process could not be obtained, as described above.

18229 **EXAMPLES**

18230 None.

18231 **APPLICATION USAGE**

18232 None.

18233 **FUTURE DIRECTIONS**

18234 None.

18235 **SEE ALSO**

18236 *fork()*, *popen()*, *waitpid()*, <stdio.h>.

18237 CHANGE HISTORY

18238 First released in Issue 1.

18239 Derived from Issue 1 of the SVID.

18240 Issue 4

18241 The following changes are incorporated for alignment with the ISO POSIX-2 standard:

- 18242 • The interface is no longer marked as an extension.
- 18243 • The simple DESCRIPTION given in Issue 3 is replaced with a more complete description in
18244 this issue. In particular, interactions between this function and *wait()* and *waitpid()* are
18245 defined.

18246 **NAME**

18247 perror — write error messages to standard error

18248 **SYNOPSIS**

18249 #include <stdio.h>

18250 void perror(const char *s);

18251 **DESCRIPTION**

18252 The *perror()* function maps the error number accessed through the symbol *errno* to a language-
 18253 dependent error message, which is written to the standard error stream as follows: first (if *s* is
 18254 not a null pointer and the character pointed to by *s* is not the null byte), the string pointed to by *s*
 18255 followed by a colon and a space character; then an error message string followed by a newline
 18256 character. The contents of the error message strings are the same as those returned by *strerror()*
 18257 with argument *errno*.

18258 The *perror()* function will mark the file associated with the standard error stream as having been
 18259 written (*st_ctime*, *st_mtime* marked for update) at some time between its successful completion
 18260 and *exit()*, *abort()*, or the completion of *fflush()* or *fclose()* on *stderr*.

18261 The *perror()* function does not change the orientation of the standard error stream.

18262 **RETURN VALUE**18263 The *perror()* function returns no value.18264 **ERRORS**

18265 No errors are defined.

18266 **EXAMPLES**

18267 None.

18268 **APPLICATION USAGE**

18269 None.

18270 **FUTURE DIRECTIONS**

18271 None.

18272 **SEE ALSO**18273 *strerror()*, <stdio.h>.18274 **CHANGE HISTORY**

18275 First released in Issue 1.

18276 Derived from Issue 1 of the SVID.

18277 **Issue 4**

18278 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 18279 • A paragraph is added to the DESCRIPTION defining the effects of this function on the
 18280 *st_ctime* and *st_mtime* fields of the standard error stream.

18281 The following change is incorporated for alignment with the ISO C standard:

- 18282 • The type of argument *s* is changed from **char *** to **const char ***.

18283 Another change is incorporated as follows:

- 18284 • The language for error message strings was given as implementation-dependent in Issue 3.
 18285 In this issue, they are defined as language-dependent.

18286 **Issue 5**

18287 A paragraph is added to the DESCRIPTION indicating that *perror()* does not change the
18288 orientation of the standard error stream.

18289 **NAME**

18290 pipe — create an interprocess channel

18291 **SYNOPSIS**

18292 #include <unistd.h>

18293 int pipe(int *fildes*[2]);18294 **DESCRIPTION**

18295 The *pipe()* function will create a pipe and place two file descriptors, one each into the arguments
 18296 *fildes*[0] and *fildes*[1], that refer to the open file descriptions for the read and write ends of the
 18297 pipe. Their integer values will be the two lowest available at the time of the *pipe()* call. The
 18298 O_NONBLOCK and FD_CLOEXEC flags shall be clear on both file descriptors. (The *fcntl()*
 18299 function can be used to set both these flags.)

18300 Data can be written to the file descriptor *fildes*[1] and read from file descriptor *fildes*[0]. A read
 18301 on the file descriptor *fildes*[0] will access data written to file descriptor *fildes*[1] on a first-in-first-
 18302 EX out basis. It is unspecified whether *fildes*[0] is also open for writing and whether *fildes*[1] is also
 18303 open for reading.

18304 A process has the pipe open for reading (correspondingly writing) if it has a file descriptor open
 18305 that refers to the read end, *fildes*[0] (write end, *fildes*[1]).

18306 Upon successful completion, *pipe()* will mark for update the *st_atime*, *st_ctime* and *st_mtime*
 18307 fields of the pipe.

18308 **RETURN VALUE**

18309 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate
 18310 the error.

18311 **ERRORS**18312 The *pipe()* function will fail if:

18313 [EMFILE] More than {OPEN_MAX} minus two file descriptors are already in use by this
 18314 process.

18315 [ENFILE] The number of simultaneously open files in the system would exceed a
 18316 system-imposed limit.

18317 **EXAMPLES**

18318 None.

18319 **APPLICATION USAGE**

18320 None.

18321 **FUTURE DIRECTIONS**

18322 None.

18323 **SEE ALSO**18324 *fcntl()*, *read()*, *write()*, <fcntl.h>, <unistd.h>.18325 **CHANGE HISTORY**

18326 First released in Issue 1.

18327 Derived from Issue 1 of the SVID.

18328 **Issue 4**

18329 The following change is incorporated in this issue:

- 18330 • The <unistd.h> header is added to the SYNOPSIS section.

18331 **Issue 4, Version 2**

18332 The DESCRIPTION is updated for X/OPEN UNIX conformance to indicate that certain
18333 dispositions of *fildev*[0] and *fildev*[1] are unspecified.

18334 NAME

18335 poll — input/output multiplexing

18336 SYNOPSIS

18337 EX `#include <poll.h>`18338 `int poll(struct pollfd fds[], nfds_t nfds, int timeout);`

18339

18340 DESCRIPTION

18341 The `poll()` function provides applications with a mechanism for multiplexing input/output over
 18342 a set of file descriptors. For each member of the array pointed to by `fds`, `poll()` examines the
 18343 given file descriptor for the event(s) specified in `events`. The number of `pollfd` structures in the
 18344 `fds` array is specified by `nfds`. The `poll()` function identifies those file descriptors on which an
 18345 application can read or write data, or on which certain events have occurred.

18346 The `fds` argument specifies the file descriptors to be examined and the events of interest for each
 18347 file descriptor. It is a pointer to an array with one member for each open file descriptor of
 18348 interest. The array's members are `pollfd` structures within which `fd` specifies an open file
 18349 descriptor and `events` and `revents` are bitmasks constructed by OR-ing a combination of the
 18350 following event flags:

18351 POLLIN Data other than high-priority data may be read without blocking. For
 18352 STREAMS, this flag is set in `revents` even if the message is of zero length.

18353 POLLRDNORM Normal data (priority band equals 0) may be read without blocking. For
 18354 STREAMS, this flag is set in `revents` even if the message is of zero length.

18355 POLLRDBAND Data from a non-zero priority band may be read without blocking. For
 18356 STREAMS, this flag is set in `revents` even if the message is of zero length.

18357 POLLPRI High-priority data may be received without blocking. For STREAMS, this flag
 18358 is set in `revents` even if the message is of zero length.

18359 POLLOUT Normal data (priority band equals 0) may be written without blocking.

18360 POLLWRNORM Same as POLLOUT.

18361 POLLWRBAND Priority data (priority band > 0) may be written. This event only examines
 18362 bands that have been written to at least once.

18363 POLLERR An error has occurred on the device or stream. This flag is only valid in the
 18364 `revents` bitmask; it is ignored in the `events` member.

18365 POLLHUP The device has been disconnected. This event and POLLOUT are mutually
 18366 exclusive; a stream can never be writable if a hangup has occurred. However,
 18367 this event and POLLIN, POLLRDNORM, POLLRDBAND or POLLPRI are not
 18368 mutually exclusive. This flag is only valid in the `revents` bitmask; it is ignored
 18369 in the `events` member.

18370 POLLNVAL The specified `fd` value is invalid. This flag is only valid in the `revents`
 18371 member; it is ignored in the `events` member.

18372 If the value of `fd` is less than 0, `events` is ignored and `revents` is set to 0 in that entry on return
 18373 from `poll()`.

18374 In each `pollfd` structure, `poll()` clears the `revents` member except that where the application
 18375 requested a report on a condition by setting one of the bits of `events` listed above, `poll()` sets the
 18376 corresponding bit in `revents` if the requested condition is true. In addition, `poll()` sets the
 18377 POLLHUP, POLLERR and POLLNVAL flag in `revents` if the condition is true, even if the

18378 application did not set the corresponding bit in **events**.

18379 If none of the defined events have occurred on any selected file descriptor, *poll()* waits at least
 18380 *timeout* milliseconds for an event to occur on any of the selected file descriptors. If the value of
 18381 *timeout* is 0, *poll()* returns immediately. If the value of *timeout* is -1, *poll()* blocks until a
 18382 requested event occurs or until the call is interrupted.

18383 Implementations may place limitations on the granularity of timeout intervals. If the requested
 18384 timeout interval requires a finer granularity than the implementation supports, the actual
 18385 timeout interval will be rounded up to the next supported value.

18386 The *poll()* function is not affected by the O_NONBLOCK flag.

18387 The *poll()* function supports regular files, terminal and pseudo-terminal devices, STREAMS-
 18388 based files, FIFOs and pipes. The behaviour of *poll()* on elements of *fds* that refer to other types
 18389 of file is unspecified.

18390 Regular files always poll TRUE for reading and writing.

18391 **RETURN VALUE**

18392 Upon successful completion, *poll()* returns a non-negative value. A positive value indicates the
 18393 total number of file descriptors that have been selected (that is, file descriptors for which the
 18394 **revents** member is non-zero). A value of 0 indicates that the call timed out and no file
 18395 descriptors have been selected. Upon failure, *poll()* returns -1 and sets *errno* to indicate the
 18396 error.

18397 **ERRORS**

18398 The *poll()* function will fail if:

18399	[EAGAIN]	The allocation of internal data structures failed but a subsequent request may
18400		succeed.
18401	[EINTR]	A signal was caught during <i>poll()</i> .
18402	[EINVAL]	The <i>nfds</i> argument is greater than {OPEN_MAX}, or one of the fd members
18403		refers to a STREAM or multiplexer that is linked (directly or indirectly)
18404		downstream from a multiplexer.

18405 **EXAMPLES**

18406 None.

18407 **APPLICATION USAGE**

18408 None.

18409 **FUTURE DIRECTIONS**

18410 None.

18411 **SEE ALSO**

18412 *getmsg()*, *putmsg()*, *read()*, *select()*, *write()*, <poll.h>, <stropts.h>, Section 2.5 on page 34.

18413 **CHANGE HISTORY**

18414 First released in Issue 4, Version 2.

18415 **Issue 5**

18416 Moved from X/OPEN UNIX extension to BASE.

18417 The description of POLLWRBAND is updated.

18418 **NAME**

18419 popen — initiate pipe streams to or from a process

18420 **SYNOPSIS**

18421 #include <stdio.h>

18422 FILE *popen(const char *command, const char *mode);

18423 **DESCRIPTION**

18424 The *popen()* function executes the command specified by the string *command*. It creates a pipe
 18425 between the calling program and the executed command, and returns a pointer to a stream that
 18426 can be used to either read from or write to the pipe.

18427 If the implementation supports the referenced **XCU** specification, the environment of the
 18428 executed command will be as if a child process were created within the *popen()* call using *fork()*,
 18429 and the child invoked the *sh* utility using the call:

18430 `execl(shell_path, "sh", "-c", command, (char *)0);`18431 where *shell_path* is an unspecified pathname for the *sh* utility.

18432 The *popen()* function ensures that any streams from previous *popen()* calls that remain open in
 18433 the parent process are closed in the new child process.

18434 The *mode* argument to *popen()* is a string that specifies I/O mode:

- 18435 1. If *mode* is **r**, when the child process is started its file descriptor `STDOUT_FILENO` will be
 18436 the writable end of the pipe, and the file descriptor *fileno(stream)* in the calling process,
 18437 where *stream* is the stream pointer returned by *popen()*, will be the readable end of the
 18438 pipe.
- 18439 2. If *mode* is **w**, when the child process is started its file descriptor `STDIN_FILENO` will be the
 18440 readable end of the pipe, and the file descriptor *fileno(stream)* in the calling process, where
 18441 *stream* is the stream pointer returned by *popen()*, will be the writable end of the pipe.
- 18442 3. If *mode* is any other value, the result is undefined.

18443 After *popen()*, both the parent and the child process will be capable of executing independently
 18444 before either terminates.

18445 Pipe streams are byte oriented.

18446 **RETURN VALUE**

18447 On successful completion, *popen()* returns a pointer to an open stream that can be used to read
 18448 or write to the pipe. Otherwise, it returns a null pointer and may set *errno* to indicate the error.

18449 **ERRORS**18450 The *popen()* function may fail if:

18451 EX [EMFILE] {FOPEN_MAX} or {STREAM_MAX} streams are currently open in the calling
 18452 process.

18453 [EINVAL] The *mode* argument is invalid.18454 The *popen()* function may also set *errno* values as described by *fork()* or *pipe()*.

18455 **EXAMPLES**

18456 None.

18457 **APPLICATION USAGE**

18458 Because open files are shared, a mode **r** command can be used as an input filter and a mode **w**
 18459 command as an output filter.

18460 Buffered reading before opening an input filter may leave the standard input of that filter
 18461 mispositioned. Similar problems with an output filter may be prevented by careful buffer
 18462 flushing, for example, with *fflush()*.

18463 A stream opened by *popen()* should be closed by *pclose()*.

18464 The behaviour of *popen()* is specified for values of *mode* of **r** and **w**. Other modes such as **rb** and
 18465 **wb** might be supported by specific implementations, but these would not be portable features.
 18466 Note that historical implementations of *popen()* only check to see if the first character of *mode* is
 18467 **r**. Thus, a *mode* of **robert the robot** would be treated as *mode r*, and a *mode* of **anything else**
 18468 would be treated as *mode w*.

18469 If the application calls *waitpid()* or *waitid()* with a *pid* argument greater than 0, and it still has a
 18470 stream that was called with *popen()* open, it must ensure that *pid* does not refer to the process
 18471 started by *popen()*.

18472 To determine whether or not the **XCU** specification environment is present, use the function call:

18473 `sysconf(_SC_2_VERSION)`

18474 (See *sysconf()*).

18475 **FUTURE DIRECTIONS**

18476 None.

18477 **SEE ALSO**

18478 *sh*, *pclose()*, *pipe()*, *sysconf()*, *system()*, **<stdio.h>**.

18479 **CHANGE HISTORY**

18480 First released in Issue 1.

18481 Derived from Issue 1 of the SVID.

18482 **Issue 4**

18483 The following changes are incorporated for alignment with the ISO POSIX-2 standard:

- 18484 • The interface is no longer marked as an extension.
- 18485 • The type of arguments *command* and *mode* are changed from **char *** to **const char ***.
- 18486 • The DESCRIPTION is completely rewritten for alignment with the ISO POSIX-2 standard,
 18487 although it describes essentially the same functionality as Issue 3.
- 18488 • The **XCU** specification's *sh* utility is no longer required in all circumstances.
- 18489 • The ERRORS section is added.

18490 Another change is incorporated as follows:

- 18491 • The APPLICATION USAGE section is extended. Only notes about buffer flushing are
 18492 retained from Issue 3.

18493 **Issue 5**

18494 A statement is added to the DESCRIPTION indicating that pipe streams are byte oriented.

18495 **NAME**

18496 pow — power function

18497 **SYNOPSIS**

18498 #include <math.h>

18499 double pow(double x, double y);

18500 **DESCRIPTION**18501 The *pow()* function computes the value of *x* raised to the power *y*, x^y . If *x* is negative, *y* must be
18502 an integer value.18503 An application wishing to check for error situations should set *errno* to 0 before calling *pow()*. If
18504 *errno* is non-zero on return, or the return value is NaN, an error has occurred.18505 **RETURN VALUE**18506 Upon successful completion, *pow()* returns the value of *x* raised to the power *y*.18507 If *x* is 0 and *y* is 0, 1.0 is returned.18508 EX If *y* is NaN, or *y* is non-zero and *x* is NaN, NaN is returned and *errno* may be set to [EDOM]. If *y*
18509 is 0.0 and *x* is NaN, either 1.0 is returned, or NaN is returned and *errno* may be set to [EDOM].18510 EX If *x* is 0.0 and *y* is negative, -HUGE_VAL is returned and *errno* may be set to [EDOM] or
18511 [ERANGE].18512 If the correct value would cause overflow, ±HUGE_VAL is returned, and *errno* is set to
18513 [ERANGE].18514 If the correct value would cause underflow, 0 is returned and *errno* may be set to [ERANGE].18515 **ERRORS**18516 The *pow()* function will fail if:18517 [EDOM] The value of *x* is negative and *y* is non-integral.

18518 [ERANGE] The value to be returned would have caused overflow.

18519 The *pow()* function may fail if:18520 EX [EDOM] The value of *x* is 0.0 and *y* is negative, or *y* is NaN.

18521 [ERANGE] The correct value would cause underflow.

18522 EX No other errors will occur.

18523 **EXAMPLES**

18524 None.

18525 **APPLICATION USAGE**

18526 None.

18527 **FUTURE DIRECTIONS**

18528 None.

18529 **SEE ALSO**18530 *exp()*, *isnan()*, <math.h>.18531 **CHANGE HISTORY**

18532 First released in Issue 1.

18533 Derived from Issue 1 of the SVID.

18534 **Issue 4**

18535 The following changes are incorporated in this issue:

- 18536 • References to *matherr()* are removed.
- 18537 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
18538 the ISO C standard and to rationalise error handling in the mathematics functions.
- 18539 • The return value specified for [EDOM] is marked as an extension.

18540 **Issue 5**

18541 The DESCRIPTION is updated to indicate how an application should check for an error. This
18542 text was previously published in the APPLICATION USAGE section.

18543 **NAME**18544

```
pread
```

 — read from a file18545 **SYNOPSIS**18546 EX

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

18547

```
ssize_t pread(int fildev, void *buf, size_t nbyte, off_t offset);
```

18548

18549 **DESCRIPTION**18550 Refer to *read()*.18551 **CHANGE HISTORY**

18552 First released in Issue 5.

18553 **NAME**

18554 printf — print formatted output

18555 **SYNOPSIS**

18556 #include <stdio.h>

18557 int printf(const char **format*, ...);18558 **DESCRIPTION**18559 Refer to *fprintf()*.18560 **CHANGE HISTORY**

18561 First released in Issue 1.

18562 Derived from Issue 1 of the SVID.

18563 **Issue 4**

18564 The following change is incorporated for alignment with the ISO C standard:

- 18565
- The type of the argument *format* is changed from **char *** to **const char ***.

18566 Another change is incorporated as follows:

- 18567
- The detailed description, including the *printf()* CHANGE HISTORY section is located under *fprintf()*.
- 18568

18569 **NAME**

18570 pthread_atfork — register fork handlers

18571 **SYNOPSIS**

18572 #include <sys/types.h>

18573 #include <unistd.h>

```
18574 int pthread_atfork(void (*prepare)(void), void (*parent)(void),
18575 void (*child)(void));
```

18576 **DESCRIPTION**

18577 The *pthread_atfork()* function declares fork handlers to be called before and after *fork()*, in the
 18578 context of the thread that called *fork()*. The *prepare* fork handler is called before *fork()* processing
 18579 commences. The *parent* fork handle is called after *fork()* processing completes in the parent
 18580 process. The *child* fork handler is called after *fork()* processing completes in the child process. If
 18581 no handling is desired at one or more of these three points, the corresponding fork handler
 18582 address(es) may be set to NULL.

18583 The order of calls to *pthread_atfork()* is significant. The *parent* and *child* fork handlers are called
 18584 in the order in which they were established by calls to *pthread_atfork()*. The *prepare* fork handlers
 18585 are called in the opposite order.

18586 **RETURN VALUE**

18587 Upon successful completion, *pthread_atfork()* returns a value of zero. Otherwise, an error
 18588 number is returned to indicate the error.

18589 **ERRORS**18590 The *pthread_atfork()* function will fail if:

18591 [ENOMEM] Insufficient table space exists to record the fork handler addresses.

18592 The *pthread_atfork()* function will not return an error code of [EINTR].18593 **EXAMPLES**

18594 None.

18595 **APPLICATION USAGE**

18596 None.

18597 **FUTURE DIRECTIONS**

18598 None.

18599 **SEE ALSO**18600 *atexit()*, *fork()*, <sys/types.h>18601 **CHANGE HISTORY**

18602 First released in Issue 5.

18603 Derived from POSIX Threads Extension, including PASC 1003.1c-95 #4.

18604 NAME

18605 pthread_attr_getguardsize, pthread_attr_setguardsize — get or set the thread guardsize
 18606 attribute

18607 SYNOPSIS

18608 EX #include <pthread.h>

```
18609 int pthread_attr_getguardsize(const pthread_attr_t *attr,  
18610                             size_t *guardsize);  
18611 int pthread_attr_setguardsize(pthread_attr_t *attr,  
18612                             size_t guardsize);  
18613
```

18614 DESCRIPTION

18615 The *guardsize* attribute controls the size of the guard area for the created thread's stack. The
 18616 *guardsize* attribute provides protection against overflow of the stack pointer. If a thread's stack is
 18617 created with guard protection, the implementation allocates extra memory at the overflow end
 18618 of the stack as a buffer against stack overflow of the stack pointer. If an application overflows
 18619 into this buffer an error results (possibly in a SIGSEGV signal being delivered to the thread).

18620 The *guardsize* attribute is provided to the application for two reasons:

- 18621 1. Overflow protection can potentially result in wasted system resources. An application that
 18622 creates a large number of threads, and which knows its threads will never overflow their
 18623 stack, can save system resources by turning off guard areas.
- 18624 2. When threads allocate large data structures on the stack, large guard areas may be needed
 18625 to detect stack overflow.

18626 The *pthread_attr_getguardsize()* function gets the *guardsize* attribute in the *attr* object. This
 18627 attribute is returned in the *guardsize* parameter.

18628 The *pthread_attr_setguardsize()* function sets the *guardsize* attribute in the *attr* object. The new
 18629 value of this attribute is obtained from the *guardsize* parameter. If *guardsize* is zero, a guard area
 18630 will not be provided for threads created with *attr*. If *guardsize* is greater than zero, a guard area of
 18631 at least size *guardsize* bytes is provided for each thread created with *attr*.

18632 A conforming implementation is permitted to round up the value contained in *guardsize* to a
 18633 multiple of the configurable system variable PAGESIZE (see <sys/mman.h>). If an
 18634 implementation rounds up the value of *guardsize* to a multiple of PAGESIZE, a call to
 18635 *pthread_attr_getguardsize()* specifying *attr* will store in the *guardsize* parameter the guard size
 18636 specified by the previous *pthread_attr_setguardsize()* function call.

18637 The default value of the *guardsize* attribute is PAGESIZE bytes. The actual value of PAGESIZE is
 18638 implementation-dependent and may not be the same on all implementations.

18639 If the *stackaddr* attribute has been set (that is, the caller is allocating and managing its own thread
 18640 stacks), the *guardsize* attribute is ignored and no protection will be provided by the
 18641 implementation. It is the responsibility of the application to manage stack overflow along with
 18642 stack allocation and management in this case.

18643 RETURN VALUE

18644 If successful, the *pthread_attr_getguardsize()* and *pthread_attr_setguardsize()* functions return
 18645 zero. Otherwise, an error number is returned to indicate the error.

18646 **ERRORS**

18647 The *pthread_attr_getguardsize()* and *pthread_attr_setguardsize()* functions will fail if:

18648 [EINVAL] The attribute *attr* is invalid.

18649 [EINVAL] The parameter *guardsize* is invalid.

18650 [EINVAL] The parameter *guardsize* contains an invalid value.

18651 **EXAMPLES**

18652 None.

18653 **APPLICATION USAGE**

18654 None.

18655 **FUTURE DIRECTIONS**

18656 None.

18657 **SEE ALSO**

18658 <pthread.h>.

18659 **CHANGE HISTORY**

18660 First released in Issue 5.

18661 **NAME**

18662 pthread_attr_init, pthread_attr_destroy — initialise and destroy threads attribute object

18663 **SYNOPSIS**

18664 #include <pthread.h>

18665 int pthread_attr_init(pthread_attr_t *attr);

18666 int pthread_attr_destroy(pthread_attr_t *attr);

18667 **DESCRIPTION**

18668 The function *pthread_attr_init()* initialises a thread attributes object *attr* with the default value
18669 for all of the individual attributes used by a given implementation.

18670 The resulting attribute object (possibly modified by setting individual attribute values), when
18671 used by *pthread_create()*, defines the attributes of the thread created. A single attributes object
18672 can be used in multiple simultaneous calls to *pthread_create()*.

18673 The *pthread_attr_destroy()* function is used to destroy a thread attributes object. An
18674 implementation may cause *pthread_attr_destroy()* to set *attr* to an implementation-dependent
18675 invalid value. The behaviour of using the attribute after it has been destroyed is undefined.

18676 **RETURN VALUE**

18677 Upon successful completion, *pthread_attr_init()* and *pthread_attr_destroy()* return a value of 0.
18678 Otherwise, an error number is returned to indicate the error.

18679 **ERRORS**

18680 The *pthread_attr_init()* function will fail if:

18681 [ENOMEM] Insufficient memory exists to initialise the thread attributes object.

18682 These functions will not return an error code of [EINTR].

18683 **EXAMPLES**

18684 None.

18685 **APPLICATION USAGE**

18686 None.

18687 **FUTURE DIRECTIONS**

18688 None.

18689 **SEE ALSO**

18690 *pthread_attr_setstackaddr()*, *pthread_attr_setstacksize()*, *pthread_attr_setdetachstate()*,
18691 *pthread_create()*, <pthread.h>.

18692 **CHANGE HISTORY**

18693 First released in Issue 5.

18694 Included for alignment with the POSIX Threads Extension.

18695 NAME

18696 pthread_attr_setdetachstate, pthread_attr_getdetachstate — set and get detachstate attribute

18697 SYNOPSIS

18698 #include <pthread.h>

18699 int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

18700 int pthread_attr_getdetachstate(const pthread_attr_t *attr,

18701 int *detachstate);

18702 DESCRIPTION

18703 The *detachstate* attribute controls whether the thread is created in a detached state. If the thread
18704 is created detached, then use of the ID of the newly created thread by the *pthread_detach()* or
18705 *pthread_join()* function is an error.

18706 The *pthread_attr_setdetachstate()* and *pthread_attr_getdetachstate()*, respectively, set and get the
18707 *detachstate* attribute in the *attr* object.

18708 The *detachstate* can be set to either PTHREAD_CREATE_DETACHED or
18709 PTHREAD_CREATE_JOINABLE. A value of PTHREAD_CREATE_DETACHED causes all
18710 threads created with *attr* to be in the detached state, whereas using a value of
18711 PTHREAD_CREATE_JOINABLE causes all threads created with *attr* to be in the joinable state.
18712 The default value of the *detachstate* attribute is PTHREAD_CREATE_JOINABLE.

18713 RETURN VALUE

18714 Upon successful completion, *pthread_attr_setdetachstate()* and *pthread_attr_getdetachstate()* return
18715 a value of 0. Otherwise, an error number is returned to indicate the error.

18716 The *pthread_attr_getdetachstate()* function stores the value of the *detachstate* attribute in *detachstate*
18717 if successful.

18718 ERRORS

18719 The *pthread_attr_setdetachstate()* function will fail if:

18720 [EINVAL] The value of *detachstate* was not valid

18721 These functions will not return an error code of [EINTR].

18722 EXAMPLES

18723 None.

18724 APPLICATION USAGE

18725 None.

18726 FUTURE DIRECTIONS

18727 None.

18728 SEE ALSO

18729 *pthread_attr_init()*, *pthread_attr_setstackaddr()*, *pthread_attr_setstacksize()*, *pthread_create()*,
18730 <pthread.h>.

18731 CHANGE HISTORY

18732 First released in Issue 5.

18733 Included for alignment with the POSIX Threads Extension.

18734 NAME

18735 pthread_attr_setinheritsched, pthread_attr_getinheritsched — set and get inheritsched attribute
 18736 (REALTIME THREADS)

18737 SYNOPSIS

18738 RTT #include <pthread.h>

```
18739 int pthread_attr_setinheritsched(pthread_attr_t *attr,
18740     int inheritsched);
18741 int pthread_attr_getinheritsched(const pthread_attr_t *attr,
18742     int *inheritsched);
18743
```

18744 DESCRIPTION

18745 The functions *pthread_attr_setinheritsched()* and *pthread_attr_getinheritsched()*, respectively, set
 18746 and get the *inheritsched* attribute in the *attr* argument.

18747 When the attribute objects are used by *pthread_create()*, the *inheritsched* attribute determines how
 18748 the other scheduling attributes of the created thread are to be set:

18749 PTHREAD_INHERIT_SCHED

18750 Specifies that the scheduling policy and associated attributes are to be inherited from the
 18751 creating thread, and the scheduling attributes in this *attr* argument are to be ignored.

18752 PTHREAD_EXPLICIT_SCHED

18753 Specifies that the scheduling policy and associated attributes are to be set to the
 18754 corresponding values from this attribute object.

18755 The symbols PTHREAD_INHERIT_SCHED and PTHREAD_EXPLICIT_SCHED are defined in
 18756 the header <pthread.h>.

18757 RETURN VALUE

18758 If successful, the *pthread_attr_setinheritsched()* and *pthread_attr_getinheritsched()* functions return
 18759 zero. Otherwise, an error number is returned to indicate the error.

18760 ERRORS

18761 The *pthread_attr_setinheritsched()* and *pthread_attr_getinheritsched()* functions will fail if:

18762 [ENOSYS] The option _POSIX_THREAD_PRIORITY_SCHEDULING is not defined and
 18763 the implementation does not support the function.

18764 The *pthread_attr_setinheritsched()* function may fail if:

18765 [EINVAL] The value of the attribute being set is not valid.

18766 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

18767 EXAMPLES

18768 None.

18769 APPLICATION USAGE

18770 After these attributes have been set, a thread can be created with the specified attributes using
 18771 *pthread_create()*. Using these routines does not affect the current running thread.

18772 FUTURE DIRECTIONS

18773 None.

18774 SEE ALSO

18775 *pthread_attr_init()*, *pthread_attr_setscope()*, *pthread_attr_setschedpolicy()*,
 18776 *pthread_attr_setschedparam()*, *pthread_create()*, <pthread.h>, *pthread_setsched_param()*, <sched.h>.

18777 **CHANGE HISTORY**

18778 First released in Issue 5.

18779 Included for alignment with the POSIX Threads Extension.

18780 Marked as part of the Realtime Threads Feature Group.

18781 **NAME**

18782 pthread_attr_setschedparam, pthread_attr_getschedparam — set and get schedparam attribute

18783 **SYNOPSIS**

18784 #include <pthread.h>

18785 int pthread_attr_setschedparam(pthread_attr_t *attr,
18786 const struct sched_param *param);18787 int pthread_attr_getschedparam(const pthread_attr_t *attr,
18788 struct sched_param *param);18789 **DESCRIPTION**18790 The functions *pthread_attr_setschedparam()* and *pthread_attr_getschedparam()*, respectively, set
18791 and get the scheduling parameter attributes in the *attr* argument. The contents of the *param*
18792 structure are defined in <sched.h>. For the SCHED_FIFO and SCHED_RR policies, the only
18793 required member of *param* is *sched_priority*.18794 **RETURN VALUE**18795 If successful, the *pthread_attr_setschedparam()* and *pthread_attr_getschedparam()* functions return
18796 zero. Otherwise, an error number is returned to indicate the error.18797 **ERRORS**18798 The *pthread_attr_setschedparam()* function may fail if:

18799 [EINVAL] The value of the attribute being set is not valid.

18800 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

18801 The *pthread_attr_setschedparam()* and *pthread_attr_getschedparam()* functions will not return an
18802 error code of [EINTR].18803 **EXAMPLES**

18804 None.

18805 **APPLICATION USAGE**18806 After these attributes have been set, a thread can be created with the specified attributes using
18807 *pthread_create()*. Using these routines does not affect the current running thread.18808 **FUTURE DIRECTIONS**

18809 None.

18810 **SEE ALSO**18811 *pthread_attr_init()*, *pthread_attr_setscope()*, *pthread_attr_setinheritsched()*,
18812 *pthread_attr_setschedpolicy()*, *pthread_create()*, <pthread.h>, *pthread_setsched_param()*, <sched.h>.18813 **CHANGE HISTORY**

18814 First released in Issue 5.

18815 Included for alignment with the POSIX Threads Extension.

18816 **NAME**

18817 pthread_attr_setschedpolicy, pthread_attr_getschedpolicy — set and get schedpolicy attribute
 18818 (**REALTIME THREADS**)

18819 **SYNOPSIS**

18820 RTT `#include <pthread.h>`

```
18821 int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
18822 int pthread_attr_getschedpolicy(const pthread_attr_t *attr,
18823 int *policy);
18824
```

18825 **DESCRIPTION**

18826 The functions *pthread_attr_setschedpolicy()* and *pthread_attr_getschedpolicy()*, respectively, set and
 18827 get the *schedpolicy* attribute in the *attr* argument.

18828 The supported values of *policy* include SCHED_FIFO, SCHED_RR and SCHED_OTHER, which
 18829 are defined by the header <**sched.h**>. When threads executing with the scheduling policy
 18830 SCHED_FIFO or SCHED_RR are waiting on a mutex, they acquire the mutex in priority order
 18831 when the mutex is unlocked.

18832 **RETURN VALUE**

18833 If successful, the *pthread_attr_setschedpolicy()* and *pthread_attr_getschedpolicy()* functions return
 18834 zero. Otherwise, an error number is returned to indicate the error.

18835 **ERRORS**

18836 The *pthread_attr_setschedpolicy()* and *pthread_attr_getschedpolicy()* functions will fail if:

18837 [ENOSYS] The option _POSIX_THREAD_PRIORITY_SCHEDULING is not defined and
 18838 the implementation does not support the function.

18839 The *pthread_attr_setschedpolicy()* function may fail if:

18840 [EINVAL] The value of the attribute being set is not valid.

18841 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

18842 **EXAMPLES**

18843 None.

18844 **APPLICATION USAGE**

18845 After these attributes have been set, a thread can be created with the specified attributes using
 18846 *pthread_create()*. Using these routines does not affect the current running thread.

18847 **FUTURE DIRECTIONS**

18848 None.

18849 **SEE ALSO**

18850 *pthread_attr_init()*, *pthread_attr_setscope()*, *pthread_attr_setinheritsched()*,
 18851 *pthread_attr_setschedparam()*, *pthread_create()*, <**pthread.h**>, *pthread_setsched_param()*, <**sched.h**>.

18852 **CHANGE HISTORY**

18853 First released in Issue 5.

18854 Included for alignment with the POSIX Threads Extension.

18855 Marked as part of the Realtime Threads Feature Group.

18856 **NAME**

18857 pthread_attr_setscope, pthread_attr_getscope — set and get contentionscope attribute
 18858 (REALTIME THREADS)

18859 **SYNOPSIS**

18860 RTT `#include <pthread.h>`

```
18861 int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
18862 int pthread_attr_getscope(const pthread_attr_t *attr,
18863 int *contentionscope);
18864
```

18865 **DESCRIPTION**

18866 The *pthread_attr_setscope()* and *pthread_attr_getscope()* functions are used to set and get the
 18867 *contentionscope* attribute in the *attr* object.

18868 The *contentionscope* attribute may have the values PTHREAD_SCOPE_SYSTEM, signifying
 18869 system scheduling contention scope, or PTHREAD_SCOPE_PROCESS, signifying process
 18870 scheduling contention scope. The symbols PTHREAD_SCOPE_SYSTEM and
 18871 PTHREAD_SCOPE_PROCESS are defined by the header **<pthread.h>**.

18872 **RETURN VALUE**

18873 If successful, the *pthread_attr_setscope()* and *pthread_attr_getscope()* functions return zero.
 18874 Otherwise, an error number is returned to indicate the error.

18875 **ERRORS**

18876 The *pthread_attr_setscope()* and *pthread_attr_getscope()* functions will fail if:

18877 [ENOSYS] The option _POSIX_THREAD_PRIORITY_SCHEDULING is not defined and
 18878 the implementation does not support the function.

18879 The *pthread_attr_setscope()*, function may fail if:

18880 [EINVAL] The value of the attribute being set is not valid.

18881 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

18882 **EXAMPLES**

18883 None.

18884 **APPLICATION USAGE**

18885 After these attributes have been set, a thread can be created with the specified attributes using
 18886 *pthread_create()*. Using these routines does not affect the current running thread.

18887 **FUTURE DIRECTIONS**

18888 None.

18889 **SEE ALSO**

18890 *pthread_attr_init()*, *pthread_attr_setinheritsched()*, *pthread_attr_setschedpolicy()*,
 18891 *pthread_attr_setschedparam()*, *pthread_create()*, **<pthread.h>**, *pthread_setsched_param()*, **<sched.h>**.

18892 **CHANGE HISTORY**

18893 First released in Issue 5.

18894 Included for alignment with the POSIX Threads Extension.

18895 Marked as part of the Realtime Threads Feature Group.

18896 **NAME**

18897 pthread_attr_setstackaddr, pthread_attr_getstackaddr — set and get stackaddr attribute

18898 **SYNOPSIS**

18899 #include <pthread.h>

18900 int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);

18901 int pthread_attr_getstackaddr(const pthread_attr_t *attr,

18902 void **stackaddr);

18903 **DESCRIPTION**18904 The functions *pthread_attr_setstackaddr()* and *pthread_attr_getstackaddr()*, respectively, set and get
18905 the thread creation *stackaddr* attribute in the *attr* object.18906 The *stackaddr* attribute specifies the location of storage to be used for the created thread's stack.

18907 The size of the storage is at least PTHREAD_STACK_MIN.

18908 **RETURN VALUE**18909 Upon successful completion, *pthread_attr_setstackaddr()* and *pthread_attr_getstackaddr()* return a
18910 value of 0. Otherwise, an error number is returned to indicate the error.18911 The *pthread_attr_getstackaddr()* function stores the *stackaddr* attribute value in *stackaddr* if
18912 successful.18913 **ERRORS**

18914 No errors are defined.

18915 These functions will not return an error code of [EINTR].

18916 **EXAMPLES**

18917 None.

18918 **APPLICATION USAGE**

18919 None.

18920 **FUTURE DIRECTIONS**

18921 None.

18922 **SEE ALSO**18923 *pthread_attr_init()*, *pthread_attr_setdetachstate()*, *pthread_attr_setstacksize()*, *pthread_create()*,
18924 <limits.h>, <pthread.h>.18925 **CHANGE HISTORY**

18926 First released in Issue 5.

18927 Included for alignment with the POSIX Threads Extension.

18928 **NAME**

18929 pthread_attr_setstacksize, pthread_attr_getstacksize — set and get stacksize attribute

18930 **SYNOPSIS**

18931 #include <pthread.h>

18932 int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);

18933 int pthread_attr_getstacksize(const pthread_attr_t *attr,

18934 size_t *stacksize);

18935 **DESCRIPTION**

18936 The functions *pthread_attr_setstacksize()* and *pthread_attr_getstacksize()*, respectively, set and get
18937 the thread creation *stacksize* attribute in the *attr* object.

18938 The *stacksize* attribute defines the minimum stack size (in bytes) allocated for the created threads
18939 stack.

18940 **RETURN VALUE**

18941 Upon successful completion, *pthread_attr_setstacksize()* and *pthread_attr_getstacksize()* return a
18942 value of 0. Otherwise, an error number is returned to indicate the error. The
18943 *pthread_attr_getstacksize()* function stores the *stacksize* attribute value in *stacksize* if successful.

18944 **ERRORS**

18945 The *pthread_attr_setstacksize()* function will fail if:

18946 [EINVAL] The value of *stacksize* is less than PTHREAD_STACK_MIN or exceeds a
18947 system-imposed limit.

18948 These functions will not return an error code of [EINTR].

18949 **EXAMPLES**

18950 None.

18951 **APPLICATION USAGE**

18952 None.

18953 **FUTURE DIRECTIONS**

18954 None.

18955 **SEE ALSO**

18956 *pthread_attr_init()*, *pthread_attr_setstackaddr()*, *pthread_attr_setdetachstate()*, *pthread_create()*,
18957 <limits.h>, <pthread.h>.

18958 **CHANGE HISTORY**

18959 First released in Issue 5.

18960 Included for alignment with the POSIX Threads Extension.

18961 **NAME**

18962 pthread_cancel — cancel execution of a thread

18963 **SYNOPSIS**

18964 #include <pthread.h>

18965 int pthread_cancel(pthread_t thread);

18966 **DESCRIPTION**

18967 The *pthread_cancel()* function requests that *thread* be canceled. The target threads cancelability state and type determines when the cancellation takes effect. When the cancellation is acted on, 18968 the cancellation cleanup handlers for *thread* are called. When the last cancellation cleanup 18969 handler returns, the thread-specific data destructor functions are called for *thread*. When the last 18970 destructor function returns, *thread* is terminated. 18971

18972 The cancellation processing in the target thread runs asynchronously with respect to the calling 18973 thread returning from *pthread_cancel()*.

18974 **RETURN VALUE**

18975 If successful, the *pthread_cancel()* function returns zero. Otherwise, an error number is returned 18976 to indicate the error.

18977 **ERRORS**18978 The *pthread_cancel()* function may fail if:

18979 [ESRCH] No thread could be found corresponding to that specified by the given thread 18980 ID.

18981 The *pthread_cancel()* function will not return an error code of [EINTR].18982 **EXAMPLES**

18983 None.

18984 **APPLICATION USAGE**

18985 None.

18986 **FUTURE DIRECTIONS**

18987 None.

18988 **SEE ALSO**

18989 *pthread_exit()*, *pthread_join()*, *pthread_setcancelstate()*, *pthread_cond_wait()*, 18990 *pthread_cond_timedwait()*, <pthread.h>.

18991 **CHANGE HISTORY**

18992 First released in Issue 5.

18993 Included for alignment with the POSIX Threads Extension.

18994 **NAME**

18995 pthread_cleanup_push, pthread_cleanup_pop — establish cancellation handlers

18996 **SYNOPSIS**

18997 #include <pthread.h>

18998 void pthread_cleanup_push(void (*routine)(void*), void *arg);

18999 void pthread_cleanup_pop(int execute);

19000 **DESCRIPTION**

19001 The *pthread_cleanup_push()* function pushes the specified cancellation cleanup handler *routine*
19002 onto the calling thread's cancellation cleanup stack. The cancellation cleanup handler is popped
19003 from the cancellation cleanup stack and invoked with the argument *arg* when: (a) the thread
19004 exits (that is, calls *pthread_exit()*), (b) the thread acts upon a cancellation request, or (c) the thread
19005 calls *pthread_cleanup_pop()* with a non-zero *execute* argument.

19006 The *pthread_cleanup_pop()* function removes the routine at the top of the calling thread's
19007 cancellation cleanup stack and optionally invokes it (if *execute* is non-zero).

19008 These functions may be implemented as macros and will appear as statements and in pairs
19009 within the same lexical scope (that is, the *pthread_cleanup_push()* macro may be thought to
19010 expand to a token list whose first token is '{' with *pthread_cleanup_pop()* expanding to a token list
19011 whose last token is the corresponding '}').

19012 The effect of calling *longjmp()* or *siglongjmp()* is undefined if there have been any calls to
19013 *pthread_cleanup_push()* or *pthread_cleanup_pop()* made without the matching call since the jump
19014 buffer was filled. The effect of calling *longjmp()* or *siglongjmp()* from inside a cancellation
19015 cleanup handler is also undefined unless the jump buffer was also filled in the cancellation
19016 cleanup handler.

19017 **RETURN VALUE**

19018 The *pthread_cleanup_push()* and *pthread_cleanup_pop()* functions return no value.

19019 **ERRORS**

19020 No errors are defined.

19021 These functions will not return an error code of [EINTR].

19022 **EXAMPLES**

19023 None.

19024 **APPLICATION USAGE**

19025 None.

19026 **FUTURE DIRECTIONS**

19027 None.

19028 **SEE ALSO**

19029 *pthread_cancel()*, *pthread_setcancelstate()*, <pthread.h>.

19030 **CHANGE HISTORY**

19031 First released in Issue 5.

19032 Included for alignment with the POSIX Threads Extension.

19033 **NAME**

19034 pthread_cond_init, pthread_cond_destroy — initialise and destroy condition variables

19035 **SYNOPSIS**

19036 #include <pthread.h>

```
19037 int pthread_cond_init(pthread_cond_t *cond,
19038     const pthread_condattr_t *attr);
19039 int pthread_cond_destroy(pthread_cond_t *cond);
19040 pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

19041 **DESCRIPTION**

19042 The function *pthread_cond_init()* initialises the condition variable referenced by *cond* with
 19043 attributes referenced by *attr*. If *attr* is NULL, the default condition variable attributes are used;
 19044 the effect is the same as passing the address of a default condition variable attributes object.
 19045 Upon successful initialisation, the state of the condition variable becomes initialised.

19046 Attempting to initialise an already initialised condition variable results in undefined behaviour.

19047 The function *pthread_cond_destroy()* destroys the given condition variable specified by *cond*; the
 19048 object becomes, in effect, uninitialised. An implementation may cause *pthread_cond_destroy()* to
 19049 set the object referenced by *cond* to an invalid value. A destroyed condition variable object can
 19050 be re-initialised using *pthread_cond_init()*; the results of otherwise referencing the object after it
 19051 has been destroyed are undefined.

19052 It is safe to destroy an initialised condition variable upon which no threads are currently
 19053 blocked. Attempting to destroy a condition variable upon which other threads are currently
 19054 blocked results in undefined behaviour.

19055 In cases where default condition variable attributes are appropriate, the macro
 19056 PTHREAD_COND_INITIALIZER can be used to initialise condition variables that are statically
 19057 allocated. The effect is equivalent to dynamic initialisation by a call to *pthread_cond_init()* with
 19058 parameter *attr* specified as NULL, except that no error checks are performed.

19059 **RETURN VALUE**

19060 If successful, the *pthread_cond_init()* and *pthread_cond_destroy()* functions return zero.
 19061 Otherwise, an error number is returned to indicate the error. The [EBUSY] and [EINVAL] error
 19062 checks, if implemented, act as if they were performed immediately at the beginning of
 19063 processing for the function and caused an error return prior to modifying the state of the
 19064 condition variable specified by *cond*.

19065 **ERRORS**

19066 The *pthread_cond_init()* function will fail if:

19067 [EAGAIN] The system lacked the necessary resources (other than memory) to initialise
 19068 another condition variable.

19069 [ENOMEM] Insufficient memory exists to initialise the condition variable.

19070 The *pthread_cond_init()* function may fail if:

19071 [EBUSY] The implementation has detected an attempt to re-initialise the object
 19072 referenced by *cond*, a previously initialised, but not yet destroyed, condition
 19073 variable.

19074 [EINVAL] The value specified by *attr* is invalid.

19075 The *pthread_cond_destroy()* function may fail if:

19076 [EBUSY] The implementation has detected an attempt to destroy the object referenced
19077 by *cond* while it is referenced (for example, while being used in a
19078 *pthread_cond_wait()* or *pthread_cond_timedwait()*) by another thread.

19079 [EINVAL] The value specified by *cond* is invalid.

19080 These functions will not return an error code of [EINTR].

19081 **EXAMPLES**

19082 None.

19083 **APPLICATION USAGE**

19084 None.

19085 **FUTURE DIRECTIONS**

19086 None.

19087 **SEE ALSO**

19088 *pthread_cond_signal()*, *pthread_cond_broadcast()*, *pthread_cond_wait()*, *pthread_cond_timedwait()*,
19089 <pthread.h>.

19090 **CHANGE HISTORY**

19091 First released in Issue 5.

19092 Included for alignment with the POSIX Threads Extension.

19093 NAME

19094 pthread_cond_signal, pthread_cond_broadcast — signal or broadcast a condition

19095 SYNOPSIS

19096 #include <pthread.h>

19097 int pthread_cond_signal(pthread_cond_t *cond);

19098 int pthread_cond_broadcast(pthread_cond_t *cond);

19099 DESCRIPTION

19100 These two functions are used to unblock threads blocked on a condition variable.

19101 The *pthread_cond_signal()* call unblocks at least one of the threads that are blocked on the
19102 specified condition variable *cond* (if any threads are blocked on *cond*).

19103 The *pthread_cond_broadcast()* call unblocks all threads currently blocked on the specified
19104 condition variable *cond*.

19105 If more than one thread is blocked on a condition variable, the scheduling policy determines the
19106 order in which threads are unblocked. When each thread unblocked as a result of a
19107 *pthread_cond_signal()* or *pthread_cond_broadcast()* returns from its call to *pthread_cond_wait()* or
19108 *pthread_cond_timedwait()*, the thread owns the mutex with which it called *pthread_cond_wait()* or
19109 *pthread_cond_timedwait()*. The thread(s) that are unblocked contend for the mutex according to
19110 the scheduling policy (if applicable), and as if each had called *pthread_mutex_lock()*.

19111 The *pthread_cond_signal()* or *pthread_cond_broadcast()* functions may be called by a thread
19112 whether or not it currently owns the mutex that threads calling *pthread_cond_wait()* or
19113 *pthread_cond_timedwait()* have associated with the condition variable during their waits;
19114 however, if predictable scheduling behaviour is required, then that mutex is locked by the
19115 thread calling *pthread_cond_signal()* or *pthread_cond_broadcast()*.

19116 The *pthread_cond_signal()* and *pthread_cond_broadcast()* functions have no effect if there are no
19117 threads currently blocked on *cond*.

19118 RETURN VALUE

19119 If successful, the *pthread_cond_signal()* and *pthread_cond_broadcast()* functions return zero.
19120 Otherwise, an error number is returned to indicate the error.

19121 ERRORS

19122 The *pthread_cond_signal()* and *pthread_cond_broadcast()* function may fail if:

19123 [EINVAL] The value *cond* does not refer to an initialised condition variable.

19124 These functions will not return an error code of [EINTR].

19125 EXAMPLES

19126 None.

19127 APPLICATION USAGE

19128 None.

19129 FUTURE DIRECTIONS

19130 None.

19131 SEE ALSO

19132 *pthread_cond_init()*, *pthread_cond_wait()*, *pthread_cond_timedwait()*, <pthread.h>.

19133 **CHANGE HISTORY**

19134 First released in Issue 5.

19135 Included for alignment with the POSIX Threads Extension.

19136 **NAME**

19137 pthread_cond_wait, pthread_cond_timedwait — wait on a condition

19138 **SYNOPSIS**

19139 #include <pthread.h>

19140 int pthread_cond_wait(pthread_cond_t *cond);

19141 int pthread_cond_timedwait(pthread_cond_t *cond,

19142 pthread_mutex_t *mutex, const struct timespec *abstime);

19143 **DESCRIPTION**

19144 The *pthread_cond_wait()* and *pthread_cond_timedwait()* functions are used to block on a condition
 19145 variable. They are called with *mutex* locked by the calling thread or undefined behaviour will
 19146 result.

19147 These functions atomically release *mutex* and cause the calling thread to block on the condition
 19148 variable *cond*; atomically here means “atomically with respect to access by another thread to the
 19149 mutex and then the condition variable”. That is, if another thread is able to acquire the mutex
 19150 after the about-to-block thread has released it, then a subsequent call to *pthread_cond_signal()* or
 19151 *pthread_cond_broadcast()* in that thread behaves as if it were issued after the about-to-block
 19152 thread has blocked.

19153 Upon successful return, the mutex has been locked and is owned by the calling thread.

19154 When using condition variables there is always a boolean predicate involving shared variables
 19155 associated with each condition wait that is true if the thread should proceed. Spurious wakeups
 19156 from the *pthread_cond_wait()* or *pthread_cond_timedwait()* functions may occur. Since the return
 19157 from *pthread_cond_wait()* or *pthread_cond_timedwait()* does not imply anything about the value
 19158 of this predicate, the predicate should be re-evaluated upon such return.

19159 The effect of using more than one mutex for concurrent *pthread_cond_wait()* or
 19160 *pthread_cond_timedwait()* operations on the same condition variable is undefined; that is, a
 19161 condition variable becomes bound to a unique mutex when a thread waits on the condition
 19162 variable, and this (dynamic) binding ends when the wait returns.

19163 A condition wait (whether timed or not) is a cancellation point. When the cancelability enable
 19164 state of a thread is set to PTHREAD_CANCEL_DEFERRED, a side effect of acting upon a
 19165 cancellation request while in a condition wait is that the mutex is (in effect) re-acquired before
 19166 calling the first cancellation cleanup handler. The effect is as if the thread were unblocked,
 19167 allowed to execute up to the point of returning from the call to *pthread_cond_wait()* or
 19168 *pthread_cond_timedwait()*, but at that point notices the cancellation request and instead of
 19169 returning to the caller of *pthread_cond_wait()* or *pthread_cond_timedwait()*, starts the thread
 19170 cancellation activities, which includes calling cancellation cleanup handlers.

19171 A thread that has been unblocked because it has been canceled while blocked in a call to
 19172 *pthread_cond_wait()* or *pthread_cond_timedwait()* does not consume any condition signal that
 19173 may be directed concurrently at the condition variable if there are other threads blocked on the
 19174 condition variable.

19175 The *pthread_cond_timedwait()* function is the same as *pthread_cond_wait()* except that an error is
 19176 returned if the absolute time specified by *abstime* passes (that is, system time equals or exceeds
 19177 *abstime*) before the condition *cond* is signaled or broadcasted, or if the absolute time specified by
 19178 *abstime* has already been passed at the time of the call. When such time-outs occur,
 19179 *pthread_cond_timedwait()* will nonetheless release and reacquire the mutex referenced by *mutex*.
 19180 The function *pthread_cond_timedwait()* is also a cancellation point.

19181 If a signal is delivered to a thread waiting for a condition variable, upon return from the signal
 19182 handler the thread resumes waiting for the condition variable as if it was not interrupted, or it

19183 returns zero due to spurious wakeup.

19184 RETURN VALUE

19185 Except in the case of [ETIMEDOUT], all these error checks act as if they were performed
 19186 immediately at the beginning of processing for the function and cause an error return, in effect,
 19187 prior to modifying the state of the mutex specified by *mutex* or the condition variable specified
 19188 by *cond*.

19189 Upon successful completion, a value of zero is returned. Otherwise, an error number is returned
 19190 to indicate the error.

19191 ERRORS

19192 The *pthread_cond_timedwait()* function will fail if:

19193 [ETIMEDOUT] The time specified by *abstime* to *pthread_cond_timedwait()* has passed.

19194 The *pthread_cond_wait()* and *pthread_cond_timedwait()* functions may fail if:

19195 [EINVAL] The value specified by *cond*, *mutex*, or *abstime* is invalid.

19196 [EINVAL] Different mutexes were supplied for concurrent *pthread_cond_wait()* or
 19197 *pthread_cond_timedwait()* operations on the same condition variable.

19198 [EINVAL] The mutex was not owned by the current thread at the time of the call.

19199 These functions will not return an error code of [EINTR].

19200 EXAMPLES

19201 None.

19202 APPLICATION USAGE

19203 None.

19204 FUTURE DIRECTIONS

19205 None.

19206 SEE ALSO

19207 *pthread_cond_signal()*, *pthread_cond_broadcast()*, <pthread.h>.

19208 CHANGE HISTORY

19209 First released in Issue 5.

19210 Included for alignment with the POSIX Threads Extension.

19211 **NAME**

19212 pthread_condattr_getpshared, pthread_condattr_setpshared — get and set the process-shared
 19213 condition variable attributes

19214 **SYNOPSIS**

```
19215 #include <pthread.h>

19216 int pthread_condattr_getpshared(const pthread_condattr_t *attr,
19217                               int *pshared);
19218 int pthread_condattr_setpshared(pthread_condattr_t *attr,
19219                               int pshared);
```

19220 **DESCRIPTION**

19221 The *pthread_condattr_getpshared()* function obtains the value of the *process-shared* attribute from
 19222 the attributes object referenced by *attr*. The *pthread_condattr_setpshared()* function is used to set
 19223 the *process-shared* attribute in an initialised attributes object referenced by *attr*.

19224 The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a condition
 19225 variable to be operated upon by any thread that has access to the memory where the condition
 19226 variable is allocated, even if the condition variable is allocated in memory that is shared by
 19227 multiple processes. If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the
 19228 condition variable will only be operated upon by threads created within the same process as the
 19229 thread that initialised the condition variable; if threads of differing processes attempt to operate
 19230 on such a condition variable, the behaviour is undefined. The default value of the attribute is
 19231 PTHREAD_PROCESS_PRIVATE.

19232 Additional attributes, their default values, and the names of the associated functions to get and
 19233 set those attribute values are implementation-dependent.

19234 **RETURN VALUE**

19235 If successful, the *pthread_condattr_setpshared()* function returns zero. Otherwise, an error
 19236 number is returned to indicate the error.

19237 If successful, the *pthread_condattr_getpshared()* function returns zero and stores the value of the
 19238 *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise, an
 19239 error number is returned to indicate the error.

19240 **ERRORS**

19241 The *pthread_condattr_getpshared()* and *pthread_condattr_setpshared()* functions may fail if:

19242 [EINVAL] The value specified by *attr* is invalid.

19243 The *pthread_condattr_setpshared()* function may fail if:

19244 [EINVAL] The new value specified for the attribute is outside the range of legal values
 19245 for that attribute.

19246 These functions will not return an error code of [EINTR].

19247 **EXAMPLES**

19248 None.

19249 **APPLICATION USAGE**

19250 None.

19251 **FUTURE DIRECTIONS**

19252 None.

19253 **SEE ALSO**

19254 *pthread_condattr_init()*, *pthread_create()*, *pthread_mutex_init()*, *pthread_cond_init()*, *<pthread.h>*.

19255 **CHANGE HISTORY**

19256 First released in Issue 5.

19257 Included for alignment with the POSIX Threads Extension.

19258 **NAME**

19259 pthread_condattr_init, pthread_condattr_destroy — initialise and destroy condition variable
 19260 attributes object

19261 **SYNOPSIS**

19262 #include <pthread.h>

19263 int pthread_condattr_init(pthread_condattr_t *attr);

19264 int pthread_condattr_destroy(pthread_condattr_t *attr);

19265 **DESCRIPTION**

19266 The function *pthread_condattr_init()* initialises a condition variable attributes object *attr* with the
 19267 default value for all of the attributes defined by the implementation.

19268 Attempting to initialise an already initialised condition variable attributes object results in
 19269 undefined behaviour.

19270 After a condition variable attributes object has been used to initialise one or more condition
 19271 variables, any function affecting the attributes object (including destruction) does not affect any
 19272 previously initialised condition variables.

19273 The *pthread_condattr_destroy()* function destroys a condition variable attributes object; the object
 19274 becomes, in effect, uninitialised. An implementation may cause *pthread_condattr_destroy()* to set
 19275 the object referenced by *attr* to an invalid value. A destroyed condition variable attributes object
 19276 can be re-initialised using *pthread_condattr_init()*; the results of otherwise referencing the object
 19277 after it has been destroyed are undefined.

19278 Additional attributes, their default values, and the names of the associated functions to get and
 19279 set those attribute values are implementation-dependent.

19280 **RETURN VALUE**

19281 If successful, the *pthread_condattr_init()* and *pthread_condattr_destroy()* functions return zero.
 19282 Otherwise, an error number is returned to indicate the error.

19283 **ERRORS**

19284 The *pthread_condattr_init()* function will fail if:

19285 [ENOMEM] Insufficient memory exists to initialise the condition variable attributes object.

19286 The *pthread_condattr_destroy()* function may fail if:

19287 [EINVAL] The value specified by *attr* is invalid.

19288 These functions will not return an error code of [EINTR].

19289 **EXAMPLES**

19290 None.

19291 **APPLICATION USAGE**

19292 None.

19293 **FUTURE DIRECTIONS**

19294 None.

19295 **SEE ALSO**

19296 *pthread_condattr_getshared()*, *pthread_create()*, *pthread_mutex_init()*, *pthread_cond_init()*,
 19297 <pthread.h>.

19298 CHANGE HISTORY

19299 First released in Issue 5.

19300 Included for alignment with the POSIX Threads Extension.

19301 **NAME**

19302 pthread_create — thread creation

19303 **SYNOPSIS**

19304 #include <pthread.h>

```
19305 int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
19306 void *(*start_routine)(void*), void *arg);
```

19307 **DESCRIPTION**

19308 The *pthread_create()* function is used to create a new thread, with attributes specified by *attr*,
 19309 within a process. If *attr* is NULL, the default attributes are used. If the attributes specified by
 19310 *attr* are modified later, the thread's attributes are not affected. Upon successful completion,
 19311 *pthread_create()* stores the ID of the created thread in the location referenced by *thread*.

19312 The thread is created executing *start_routine* with *arg* as its sole argument. If the *start_routine*
 19313 returns, the effect is as if there was an implicit call to *pthread_exit()* using the return value of
 19314 *start_routine* as the exit status. Note that the thread in which *main()* was originally invoked
 19315 differs from this. When it returns from *main()*, the effect is as if there was an implicit call to
 19316 *exit()* using the return value of *main()* as the exit status.

19317 The signal state of the new thread is initialised as follows:

- 19318 • The signal mask is inherited from the creating thread.
- 19319 • The set of signals pending for the new thread is empty.

19320 If *pthread_create()* fails, no new thread is created and the contents of the location referenced by
 19321 *thread* are undefined.

19322 **RETURN VALUE**

19323 If successful, the *pthread_create()* function returns zero. Otherwise, an error number is returned
 19324 to indicate the error.

19325 **ERRORS**

19326 The *pthread_create()* function will fail if:

19327 [EAGAIN] The system lacked the necessary resources to create another thread, or the
 19328 system-imposed limit on the total number of threads in a process
 19329 PTHREAD_THREADS_MAX would be exceeded.

19330 [EINVAL] The value specified by *attr* is invalid.

19331 EX [EPERM] The caller does not have appropriate permission to set the required
 19332 scheduling parameters or scheduling policy.

19333 The *pthread_create()* function will not return an error code of [EINTR].

19334 **EXAMPLES**

19335 None.

19336 **APPLICATION USAGE**

19337 None.

19338 **FUTURE DIRECTIONS**

19339 None.

19340 **SEE ALSO**

19341 *pthread_exit()*, *pthread_join()*, *fork()*, <pthread.h>.

19342 **CHANGE HISTORY**

19343 First released in Issue 5.

19344 Included for alignment with the POSIX Threads Extension.

19345 **NAME**

19346 pthread_detach — detach a thread

19347 **SYNOPSIS**

19348 #include <pthread.h>

19349 int pthread_detach(pthread_t *thread*);19350 **DESCRIPTION**

19351 The *pthread_detach()* function is used to indicate to the implementation that storage for the
 19352 thread *thread* can be reclaimed when that thread terminates. If *thread* has not terminated,
 19353 *pthread_detach()* will not cause it to terminate. The effect of multiple *pthread_detach()* calls on the
 19354 same target thread is unspecified.

19355 **RETURN VALUE**

19356 If the call succeeds, *pthread_detach()* returns 0. Otherwise, an error number is returned to
 19357 indicate the error.

19358 **ERRORS**19359 The *pthread_detach()* function will fail if:

19360 [EINVAL] The implementation has detected that the value specified by *thread* does not
 19361 refer to a joinable thread.

19362 [ESRCH] No thread could be found corresponding to that specified by the given thread
 19363 ID.

19364 The *pthread_detach()* function will not return an error code of [EINTR].19365 **EXAMPLES**

19366 None.

19367 **APPLICATION USAGE**

19368 None.

19369 **FUTURE DIRECTIONS**

19370 None.

19371 **SEE ALSO**19372 *pthread_join()*, <pthread.h>.19373 **CHANGE HISTORY**

19374 First released in Issue 5.

19375 Included for alignment with the POSIX Threads Extension.

19376 NAME

19377 pthread_equal — compare thread IDs

19378 SYNOPSIS

19379 #include <pthread.h>

19380 int pthread_equal(pthread_t *t1*, pthread_t *t2*);

19381 DESCRIPTION

19382 This function compares the thread IDs *t1* and *t2*.

19383 RETURN VALUE

19384 The *pthread_equal()* function returns a non-zero value if *t1* and *t2* are equal; otherwise, zero is
19385 returned.

19386 If either *t1* or *t2* are not valid thread IDs, the behaviour is undefined.

19387 ERRORS

19388 No errors are defined.

19389 The *pthread_equal()* function will not return an error code of [EINTR].

19390 EXAMPLES

19391 None.

19392 APPLICATION USAGE

19393 None.

19394 FUTURE DIRECTIONS

19395 None.

19396 SEE ALSO

19397 *pthread_create()*, *pthread_self()*, <pthread.h>.

19398 CHANGE HISTORY

19399 First released in Issue 5.

19400 Included for alignment with the POSIX Threads Extension.

19401 **NAME**

19402 pthread_exit — thread termination

19403 **SYNOPSIS**

19404 #include <pthread.h>

19405 void pthread_exit(void *value_ptr);

19406 **DESCRIPTION**

19407 The *pthread_exit()* function terminates the calling thread and makes the value *value_ptr* available
 19408 to any successful join with the terminating thread. Any cancellation cleanup handlers that have
 19409 been pushed and not yet popped are popped in the reverse order that they were pushed and
 19410 then executed. After all cancellation cleanup handlers have been executed, if the thread has any
 19411 thread-specific data, appropriate destructor functions will be called in an unspecified order.
 19412 Thread termination does not release any application visible process resources, including, but not
 19413 limited to, mutexes and file descriptors, nor does it perform any process level cleanup actions,
 19414 including, but not limited to, calling any *atexit()* routines that may exist.

19415 An implicit call to *pthread_exit()* is made when a thread other than the thread in which *main()*
 19416 was first invoked returns from the start routine that was used to create it. The function's return
 19417 value serves as the thread's exit status.

19418 The behaviour of *pthread_exit()* is undefined if called from a cancellation cleanup handler or
 19419 destructor function that was invoked as a result of either an implicit or explicit call to
 19420 *pthread_exit()*.

19421 After a thread has terminated, the result of access to local (auto) variables of the thread is
 19422 undefined. Thus, references to local variables of the exiting thread should not be used for the
 19423 *pthread_exit()* *value_ptr* parameter value.

19424 The process exits with an exit status of 0 after the last thread has been terminated. The
 19425 behaviour is as if the implementation called *exit()* with a zero argument at thread termination
 19426 time.

19427 **RETURN VALUE**19428 The *pthread_exit()* function cannot return to its caller.19429 **ERRORS**

19430 No errors are defined.

19431 The *pthread_exit()* function will not return an error code of [EINTR].19432 **EXAMPLES**

19433 None.

19434 **APPLICATION USAGE**

19435 None.

19436 **FUTURE DIRECTIONS**

19437 None.

19438 **SEE ALSO**19439 *pthread_create()*, *pthread_join()*, *exit()*, *_exit()*, <pthread.h>.19440 **CHANGE HISTORY**

19441 First released in Issue 5.

19442 Included for alignment with the POSIX Threads Extension.

19443 **NAME**

19444 pthread_getconcurrency, pthread_setconcurrency — get or set level of concurrency

19445 **SYNOPSIS**

19446 EX #include <pthread.h>

19447 int pthread_getconcurrency(void);

19448 int pthread_setconcurrency(int new_level);

19449

19450 **DESCRIPTION**

19451 Unbound threads in a process may or may not be required to be simultaneously active. By
 19452 default, the threads implementation ensures that a sufficient number of threads are active so that
 19453 the process can continue to make progress. While this conserves system resources, it may not
 19454 produce the most effective level of concurrency.

19455 The *pthread_setconcurrency()* function allows an application to inform the threads
 19456 implementation of its desired concurrency level, *new_level*. The actual level of concurrency
 19457 provided by the implementation as a result of this function call is unspecified.

19458 If *new_level* is zero, it causes the implementation to maintain the concurrency level at its
 19459 discretion as if *pthread_setconcurrency()* was never called.

19460 The *pthread_getconcurrency()* function returns the value set by a previous call to the
 19461 *pthread_setconcurrency()* function. If the *pthread_setconcurrency()* function was not previously
 19462 called, this function returns zero to indicate that the implementation is maintaining the
 19463 concurrency level.

19464 When an application calls *pthread_setconcurrency()* it is informing the implementation of its
 19465 desired concurrency level. The implementation uses this as a hint, not a requirement.

19466 If an implementation does not support multiplexing of user threads on top of several kernel
 19467 scheduled entities, the *pthread_setconcurrency()* and *pthread_getconcurrency()* functions will be
 19468 provided for source code compatibility but they will have no effect when called. To maintain the
 19469 function semantics, the *new_level* parameter will be saved when *pthread_setconcurrency()* is called
 19470 so that a subsequent call to *pthread_getconcurrency()* returns the same value.

19471 **RETURN VALUE**

19472 If successful, the *pthread_setconcurrency()* function returns zero. Otherwise, an error number is
 19473 returned to indicate the error.

19474 The *pthread_getconcurrency()* function always returns the concurrency level set by a previous call
 19475 to *pthread_setconcurrency()*. If the *pthread_setconcurrency()* function has never been called,
 19476 *pthread_getconcurrency()* returns zero.

19477 **ERRORS**

19478 The *pthread_setconcurrency()* function will fail if:

19479 [EINVAL] The value specified by *new_level* is negative.

19480 [EAGAIN] The value specific by *new_level* would cause a system resource to be exceeded.

19481 **EXAMPLES**

19482 None.

19483 **APPLICATION USAGE**

19484 Use of these functions changes the state of the underlying concurrency upon which the
 19485 application depends. Library developers are advised to not use the *pthread_getconcurrency()* and
 19486 *pthread_setconcurrency()* functions since their use may conflict with an applications use of these
 19487 functions.

19488 **FUTURE DIRECTIONS**

19489 None.

19490 **SEE ALSO**

19491 <pthread.h>.

19492 **CHANGE HISTORY**

19493 First released in Issue 5.

19494 NAME

19495 pthread_getschedparam, pthread_setschedparam — dynamic thread scheduling parameters
 19496 access (**REALTIME THREADS**)

19497 SYNOPSIS

19498 RTT `#include <pthread.h>`

```
19499 int pthread_getschedparam(pthread_t thread, int *policy,
19500     struct sched_param *param);
19501 int pthread_setschedparam(pthread_t thread, int *policy,
19502     const struct sched_param *param);
19503
```

19504 DESCRIPTION

19505 The *pthread_getschedparam()* and *pthread_setschedparam()* allow the scheduling policy and
 19506 scheduling parameters of individual threads within a multi-threaded process to be retrieved and
 19507 set. For SCHED_FIFO and SCHED_RR, the only required member of the **sched_param** structure
 19508 is the priority *sched_priority*. For SCHED_OTHER, the affected scheduling parameters are
 19509 implementation-dependent.

19510 The *pthread_getschedparam()* function retrieves the scheduling policy and scheduling parameters
 19511 for the thread whose thread ID is given by *thread* and stores those values in *policy* and *param*,
 19512 respectively. The priority value returned from *pthread_getschedparam()* is the value specified by
 19513 the most recent *pthread_setschedparam()* or *pthread_create()* call affecting the target thread, and
 19514 reflects any temporary adjustments to its priority as a result of any priority inheritance or ceiling
 19515 functions. The *pthread_setschedparam()* function sets the scheduling policy and associated
 19516 scheduling parameters for the thread whose thread ID is given by *thread* to the policy and
 19517 associated parameters provided in *policy* and *param*, respectively.

19518 The *policy* parameter may have the value SCHED_OTHER, that has implementation-dependent
 19519 scheduling parameters, SCHED_FIFO or SCHED_RR, that have the single scheduling parameter,
 19520 *priority*.

19521 If the *pthread_setschedparam()* function fails, no scheduling parameters will be changed for the
 19522 target thread.

19523 RETURN VALUE

19524 If successful, the *pthread_getschedparam()* and *pthread_setschedparam()* functions return zero.
 19525 Otherwise, an error number is returned to indicate the error.

19526 ERRORS

19527 The *pthread_getschedparam()* and *pthread_setschedparam()* functions will fail if:

19528 [ENOSYS] The option _POSIX_THREAD_PRIORITY_SCHEDULING is not defined and
 19529 the implementation does not support the function.

19530 The *pthread_getschedparam()* function may fail if:

19531 [ESRCH] The value specified by *thread* does not refer to a existing thread.

19532 The *pthread_setschedparam()* function may fail if:

19533 [EINVAL] The value specified by *policy* or one of the scheduling parameters associated
 19534 with the scheduling policy *policy* is invalid.

19535 [ENOTSUP] An attempt was made to set the policy or scheduling parameters to an
 19536 unsupported value.

19537 [EPERM] The caller does not have the appropriate permission to set either the
 19538 scheduling parameters or the scheduling policy of the specified thread.

19539	[EPERM]	The implementation does not allow the application to modify one of the	
19540		parameters to the value specified.	
19541	[ESRCH]	The value specified by <i>thread</i> does not refer to a existing thread.	
19542	EXAMPLES		
19543		None.	
19544	APPLICATION USAGE		
19545		None.	
19546	FUTURE DIRECTIONS		
19547		None.	
19548	SEE ALSO		
19549		<i>sched_setparam()</i> , <i>sched_getparam()</i> , <i>sched_setscheduler()</i> , <i>sched_getscheduler()</i> , <pthread.h> ,	
19550		<sched.h> .	
19551	CHANGE HISTORY		
19552		First released in Issue 5.	
19553		Included for alignment with the POSIX Threads Extension.	

19554 **NAME**

19555 pthread_join — wait for thread termination

19556 **SYNOPSIS**

19557 #include <pthread.h>

19558 int pthread_join(pthread_t *thread*, void ***value_ptr*);19559 **DESCRIPTION**

19560 The *pthread_join()* function suspends execution of the calling thread until the target *thread*
 19561 terminates, unless the target *thread* has already terminated. On return from a successful
 19562 *pthread_join()* call with a non-NULL *value_ptr* argument, the value passed to *pthread_exit()* by
 19563 the terminating thread is made available in the location referenced by *value_ptr*. When a
 19564 *pthread_join()* returns successfully, the target thread has been terminated. The results of
 19565 multiple simultaneous calls to *pthread_join()* specifying the same target thread are undefined. If
 19566 the thread calling *pthread_join()* is canceled, then the target thread will not be detached.

19567 It is unspecified whether a thread that has exited but remains unjoined counts against
 19568 _POSIX_THREAD_THREADS_MAX.

19569 **RETURN VALUE**

19570 If successful, the *pthread_join()* function returns zero. Otherwise, an error number is returned to
 19571 indicate the error.

19572 **ERRORS**19573 The *pthread_join()* function will fail if:

19574 [EINVAL] The implementation has detected that the value specified by *thread* does not
 19575 refer to a joinable thread.

19576 [ESRCH] No thread could be found corresponding to that specified by the given thread
 19577 ID.

19578 The *pthread_join()* function may fail if:

19579 [EDEADLK] A deadlock was detected or the value of *thread* specifies the calling thread.

19580 The *pthread_join()* function will not return an error code of [EINTR].19581 **EXAMPLES**

19582 None.

19583 **APPLICATION USAGE**

19584 None.

19585 **FUTURE DIRECTIONS**

19586 None.

19587 **SEE ALSO**19588 *pthread_create()*, *wait()*, <pthread.h>.19589 **CHANGE HISTORY**

19590 First released in Issue 5.

19591 Included for alignment with the POSIX Threads Extension.

19592 **NAME**

19593 pthread_key_create — thread-specific data key creation

19594 **SYNOPSIS**

19595 #include <pthread.h>

19596 int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));

19597 **DESCRIPTION**

19598 This function creates a thread-specific data key visible to all threads in the process. Key values
 19599 provided by *pthread_key_create()* are opaque objects used to locate thread-specific data.
 19600 Although the same key value may be used by different threads, the values bound to the key by
 19601 *pthread_setspecific()* are maintained on a per-thread basis and persist for the life of the calling
 19602 thread.

19603 Upon key creation, the value NULL is associated with the new key in all active threads. Upon
 19604 thread creation, the value NULL is associated with all defined keys in the new thread.

19605 An optional destructor function may be associated with each key value. At thread exit, if a key
 19606 value has a non-NULL destructor pointer, and the thread has a non-NULL value associated with
 19607 that key, the function pointed to is called with the current associated value as its sole argument.
 19608 The order of destructor calls is unspecified if more than one destructor exists for a thread when
 19609 it exits.

19610 If, after all the destructors have been called for all non-NULL values with associated destructors,
 19611 there are still some non-NULL values with associated destructors, then the process will be
 19612 repeated. If, after at least PTHREAD_DESTRUCTOR_ITERATIONS iterations of destructor calls
 19613 for outstanding non-NULL values, there are still some non-NULL values with associated
 19614 destructors, implementations may stop calling destructors, or they may continue calling
 19615 destructors until no non-NULL values with associated destructors exist, even though this might
 19616 result in an infinite loop.

19617 **RETURN VALUE**

19618 If successful, the *pthread_key_create()* function stores the newly created key value at *key and
 19619 returns zero. Otherwise, an error number is returned to indicate the error.

19620 **ERRORS**19621 The *pthread_key_create()* function will fail if:

19622 [EAGAIN] The system lacked the necessary resources to create another thread-specific
 19623 data key, or the system-imposed limit on the total number of keys per process
 19624 PTHREAD_KEYS_MAX has been exceeded.

19625 [ENOMEM] Insufficient memory exists to create the key.

19626 The *pthread_key_create()* function will not return an error code of [EINTR].

19627 **EXAMPLES**

19628 None.

19629 **APPLICATION USAGE**

19630 None.

19631 **FUTURE DIRECTIONS**

19632 None.

19633 **SEE ALSO**19634 *pthread_getspecific()*, *pthread_setspecific()*, *pthread_key_delete()*, <pthread.h>.

19635 **CHANGE HISTORY**

19636 First released in Issue 5.

19637 Included for alignment with the POSIX Threads Extension.

19638 **NAME**

19639 pthread_key_delete — thread-specific data key deletion

19640 **SYNOPSIS**

19641 #include <pthread.h>

19642 int pthread_key_delete(pthread_key_t key);

19643 **DESCRIPTION**

19644 This function deletes a thread-specific data key previously returned by *pthread_key_create()*. The
 19645 thread-specific data values associated with *key* need not be NULL at the time
 19646 *pthread_key_delete()* is called. It is the responsibility of the application to free any application
 19647 storage or perform any cleanup actions for data structures related to the deleted key or
 19648 associated thread-specific data in any threads; this cleanup can be done either before or after
 19649 *pthread_key_delete()* is called. Any attempt to use *key* following the call to *pthread_key_delete()*
 19650 results in undefined behaviour.

19651 The *pthread_key_delete()* function is callable from within destructor functions. No destructor
 19652 functions will be invoked by *pthread_key_delete()*. Any destructor function that may have been
 19653 associated with *key* will no longer be called upon thread exit.

19654 **RETURN VALUE**

19655 If successful, the *pthread_key_delete()* function returns zero. Otherwise, an error number is
 19656 returned to indicate the error.

19657 **ERRORS**19658 The *pthread_key_delete()* function may fail if:19659 [EINVAL] The *key* value is invalid.19660 The *pthread_key_delete()* function will not return an error code of [EINTR].19661 **EXAMPLES**

19662 None.

19663 **APPLICATION USAGE**

19664 None.

19665 **FUTURE DIRECTIONS**

19666 None.

19667 **SEE ALSO**19668 *pthread_key_create()*, <pthread.h>.19669 **CHANGE HISTORY**

19670 First released in Issue 5.

19671 Included for alignment with the POSIX Threads Extension.

19672 NAME

19673 pthread_kill — send a signal to a thread

19674 SYNOPSIS

19675 #include <signal.h>

19676 int pthread_kill(pthread_t thread, int sig);

19677 DESCRIPTION

19678 The *pthread_kill()* function is used to request that a signal be delivered to the specified thread.

19679 As in *kill()*, if *sig* is zero, error checking is performed but no signal is actually sent.

19680 RETURN VALUE

19681 Upon successful completion, the function returns a value of zero. Otherwise the function
19682 returns an error number. If the *pthread_kill()* function fails, no signal is sent.

19683 ERRORS

19684 The *pthread_kill()* function will fail if:

19685 [ESRCH] No thread could be found corresponding to that specified by the given thread
19686 ID.

19687 [EINVAL] The value of the *sig* argument is an invalid or unsupported signal number.

19688 The *pthread_kill()* function will not return an error code of [EINTR].

19689 EXAMPLES

19690 None.

19691 APPLICATION USAGE

19692 None.

19693 FUTURE DIRECTIONS

19694 None.

19695 SEE ALSO

19696 *kill()*, *pthread_self()*, *raise()*, <signal.h>.

19697 CHANGE HISTORY

19698 First released in Issue 5.

19699 Included for alignment with the POSIX Threads Extension.

19700 **NAME**

19701 pthread_mutex_init, pthread_mutex_destroy — initialise or destroy a mutex

19702 **SYNOPSIS**

19703 #include <pthread.h>

```
19704 int pthread_mutex_init(pthread_mutex_t *mutex,
19705     const pthread_mutexattr_t *attr);
19706 int pthread_mutex_destroy(pthread_mutex_t *mutex);
19707 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

19708 **DESCRIPTION**

19709 The *pthread_mutex_init()* function initialises the mutex referenced by *mutex* with attributes
 19710 specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect is the same as
 19711 passing the address of a default mutex attributes object. Upon successful initialisation, the state
 19712 of the mutex becomes initialised and unlocked.

19713 Attempting to initialise an already initialised mutex results in undefined behaviour.

19714 The *pthread_mutex_destroy()* function destroys the mutex object referenced by *mutex*; the mutex
 19715 object becomes, in effect, uninitialised. An implementation may cause *pthread_mutex_destroy()*
 19716 to set the object referenced by *mutex* to an invalid value. A destroyed mutex object can be re-
 19717 initialised using *pthread_mutex_init()*; the results of otherwise referencing the object after it has
 19718 been destroyed are undefined.

19719 It is safe to destroy an initialised mutex that is unlocked. Attempting to destroy a locked mutex
 19720 results in undefined behaviour.

19721 In cases where default mutex attributes are appropriate, the macro
 19722 PTHREAD_MUTEX_INITIALIZER can be used to initialise mutexes that are statically allocated.
 19723 The effect is equivalent to dynamic initialisation by a call to *pthread_mutex_init()* with parameter
 19724 *attr* specified as NULL, except that no error checks are performed.

19725 **RETURN VALUE**

19726 If successful, the *pthread_mutex_init()* and *pthread_mutex_destroy()* functions return zero.
 19727 Otherwise, an error number is returned to indicate the error. The [EBUSY] and [EINVAL] error
 19728 checks, if implemented, act as if they were performed immediately at the beginning of
 19729 processing for the function and cause an error return prior to modifying the state of the mutex
 19730 specified by *mutex*.

19731 **ERRORS**

19732 The *pthread_mutex_init()* function will fail if:

19733 [EAGAIN] The system lacked the necessary resources (other than memory) to initialise
 19734 another mutex.

19735 [ENOMEM] Insufficient memory exists to initialise the mutex.

19736 [EPERM] The caller does not have the privilege to perform the operation.

19737 The *pthread_mutex_init()* function may fail if:

19738 [EBUSY] The implementation has detected an attempt to re-initialise the object
 19739 referenced by *mutex*, a previously initialised, but not yet destroyed, mutex.

19740 [EINVAL] The value specified by *attr* is invalid.

19741 The *pthread_mutex_destroy()* function may fail if:

19742 [EBUSY] The implementation has detected an attempt to destroy the object referenced
 19743 by *mutex* while it is locked or referenced (for example, while being used in a

19744		<i>pthread_cond_wait()</i> or <i>pthread_cond_timedwait()</i> by another thread.	
19745	[EINVAL]	The value specified by <i>mutex</i> is invalid.	
19746		These functions will not return an error code of [EINTR].	
19747	EXAMPLES		
19748		None.	
19749	APPLICATION USAGE		
19750		None.	
19751	FUTURE DIRECTIONS		
19752		None.	
19753	SEE ALSO		
19754		<i>pthread_mutex_getprioceiling()</i> , <i>pthread_mutex_lock()</i> , <i>pthread_mutex_unlock()</i> ,	
19755		<i>pthread_mutex_setprioceiling()</i> , <i>pthread_mutex_trylock()</i> , <i>pthread_mutexattr_getpshared()</i> ,	
19756		<i>pthread_mutexattr_setpshared()</i> , <pthread.h>.	
19757	CHANGE HISTORY		
19758		First released in Issue 5.	
19759		Included for alignment with the POSIX Threads Extension.	

19760 **NAME**

19761 pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a
 19762 mutex

19763 **SYNOPSIS**

19764 #include <pthread.h>

19765 int pthread_mutex_lock(pthread_mutex_t *mutex);

19766 int pthread_mutex_trylock(pthread_mutex_t *mutex);

19767 int pthread_mutex_unlock(pthread_mutex_t *mutex);

19768 **DESCRIPTION**

19769 The mutex object referenced by *mutex* is locked by calling *pthread_mutex_lock()*. If the mutex is
 19770 already locked, the calling thread blocks until the mutex becomes available. This operation
 19771 returns with the mutex object referenced by *mutex* in the locked state with the calling thread as
 19772 its owner.

19773 EX If the mutex type is PTHREAD_MUTEX_NORMAL, deadlock detection is not provided.
 19774 Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it
 19775 has not locked or a mutex which is unlocked, undefined behaviour results.

19776 If the mutex type is PTHREAD_MUTEX_ERRORCHECK, then error checking is provided. If a
 19777 thread attempts to relock a mutex that it has already locked, an error will be returned. If a thread
 19778 attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be
 19779 returned.

19780 If the mutex type is PTHREAD_MUTEX_RECURSIVE, then the mutex maintains the concept of
 19781 a lock count. When a thread successfully acquires a mutex for the first time, the lock count is set
 19782 to one. Every time a thread relocks this mutex, the lock count is incremented by one. Each time
 19783 the thread unlocks the mutex, the lock count is decremented by one. When the lock count
 19784 reaches zero, the mutex becomes available for other threads to acquire. If a thread attempts to
 19785 unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.

19786 If the mutex type is PTHREAD_MUTEX_DEFAULT, attempting to recursively lock the mutex
 19787 results in undefined behaviour. Attempting to unlock the mutex if it was not locked by the
 19788 calling thread results in undefined behaviour. Attempting to unlock the mutex if it is not locked
 19789 results in undefined behaviour.

19790 The function *pthread_mutex_trylock()* is identical to *pthread_mutex_lock()* except that if the mutex
 19791 object referenced by *mutex* is currently locked (by any thread, including the current thread), the
 19792 call returns immediately.

19793 EX The *pthread_mutex_unlock()* function releases the mutex object referenced by *mutex*. The manner
 19794 in which a mutex is released is dependent upon the mutex's type attribute. If there are threads
 19795 blocked on the mutex object referenced by *mutex* when *pthread_mutex_unlock()* is called,
 19796 resulting in the mutex becoming available, the scheduling policy is used to determine which
 19797 thread shall acquire the mutex. (In the case of PTHREAD_MUTEX_RECURSIVE mutexes, the
 19798 mutex becomes available when the count reaches zero and the calling thread no longer has any
 19799 locks on this mutex).

19800 If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the
 19801 thread resumes waiting for the mutex as if it was not interrupted.

19802 **RETURN VALUE**

19803 If successful, the *pthread_mutex_lock()* and *pthread_mutex_unlock()* functions return zero.
 19804 Otherwise, an error number is returned to indicate the error.

19805 The function *pthread_mutex_trylock()* returns zero if a lock on the mutex object referenced by
 19806 *mutex* is acquired. Otherwise, an error number is returned to indicate the error.

19807 ERRORS

19808 The *pthread_mutex_lock()* and *pthread_mutex_trylock()* functions will fail if:

19809 [EINVAL] The *mutex* was created with the protocol attribute having the value
 19810 PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than
 19811 the mutex's current priority ceiling.

19812 The *pthread_mutex_trylock()* function will fail if:

19813 [EBUSY] The *mutex* could not be acquired because it was already locked.

19814 The *pthread_mutex_lock()*, *pthread_mutex_trylock()* and *pthread_mutex_unlock()* functions may fail
 19815 if:

19816 [EINVAL] The value specified by *mutex* does not refer to an initialised mutex object.

19817 EX [EAGAIN] The mutex could not be acquired because the maximum number of recursive
 19818 locks for *mutex* has been exceeded.

19819 The *pthread_mutex_lock()* function may fail if:

19820 [EDEADLK] The current thread already owns the mutex.

19821 The *pthread_mutex_unlock()* function may fail if:

19822 [EPERM] The current thread does not own the mutex.

19823 These functions will not return an error code of [EINTR].

19824 EXAMPLES

19825 None.

19826 APPLICATION USAGE

19827 None.

19828 FUTURE DIRECTIONS

19829 None.

19830 SEE ALSO

19831 *pthread_mutex_init()*, *pthread_mutex_destroy()*, <pthread.h>.

19832 CHANGE HISTORY

19833 First released in Issue 5.

19834 Included for alignment with the POSIX Threads Extension.

19835 **NAME**

19836 pthread_mutex_setprioceiling, pthread_mutex_getprioceiling — change the priority ceiling of a
 19837 mutex (**REALTIME THREADS**)

19838 **SYNOPSIS**

19839 RTT `#include <pthread.h>`

```
19840 int pthread_mutex_setprioceiling(pthread_mutex_t *mutex,
19841     int prioceiling, int *old_ceiling);
19842 int pthread_mutex_getprioceiling(const pthread_mutex_t *mutex,
19843     int *prioceiling);
19844
```

19845 **DESCRIPTION**

19846 The *pthread_mutex_getprioceiling()* function returns the current priority ceiling of the mutex.

19847 The *pthread_mutex_setprioceiling()* function either locks the mutex if it is unlocked, or blocks until
 19848 it can successfully lock the mutex, then it changes the mutex's priority ceiling and releases the
 19849 mutex. When the change is successful, the previous value of the priority ceiling is returned in
 19850 *old_ceiling*. The process of locking the mutex need not adhere to the priority protect protocol.

19851 If the *pthread_mutex_setprioceiling()* function fails, the mutex priority ceiling is not changed.

19852 **RETURN VALUE**

19853 If successful, the *pthread_mutex_setprioceiling()* and *pthread_mutex_getprioceiling()* functions
 19854 return zero. Otherwise, an error number is returned to indicate the error.

19855 **ERRORS**

19856 The *pthread_mutex_getprioceiling()* and *pthread_mutex_setprioceiling()* functions will fail if:

19857 [ENOSYS] The option `_POSIX_THREAD_PRIO_PROTECT` is not defined and the
 19858 implementation does not support the function.

19859 The *pthread_mutex_setprioceiling()* and *pthread_mutex_getprioceiling()* functions may fail if:

19860 [EINVAL] The priority requested by *prioceiling* is out of range.

19861 [EINVAL] The value specified by *mutex* does not refer to a currently existing mutex.

19862 [ENOSYS] The implementation does not support the priority ceiling protocol for
 19863 mutexes.

19864 [EPERM] The caller does not have the privilege to perform the operation.

19865 **EXAMPLES**

19866 None.

19867 **APPLICATION USAGE**

19868 None.

19869 **FUTURE DIRECTIONS**

19870 None.

19871 **SEE ALSO**

19872 *pthread_mutex_init()*, *pthread_mutex_lock()*, *pthread_mutex_unlock()*, *pthread_mutex_trylock()*,
 19873 `<pthread.h>`.

19874 CHANGE HISTORY

19875 First released in Issue 5.

19876 Included for alignment with the POSIX Threads Extension.

19877 Marked as part of the Realtime Threads Feature Group.

19878 **NAME**

19879 pthread_mutexattr_getpshared, pthread_mutexattr_setpshared — set and get process-shared
 19880 attribute

19881 **SYNOPSIS**

19882 #include <pthread.h>

19883 int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr,
 19884 int *pshared);
 19885 int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
 19886 int pshared);

19887 **DESCRIPTION**

19888 The *pthread_mutexattr_getpshared()* function obtains the value of the *process-shared* attribute from
 19889 the attributes object referenced by *attr*. The *pthread_mutexattr_setpshared()* function is used to set
 19890 the *process-shared* attribute in an initialised attributes object referenced by *attr*.

19891 The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a mutex to be
 19892 operated upon by any thread that has access to the memory where the mutex is allocated, even if
 19893 the mutex is allocated in memory that is shared by multiple processes. If the *process-shared*
 19894 attribute is PTHREAD_PROCESS_PRIVATE, the mutex will only be operated upon by threads
 19895 created within the same process as the thread that initialised the mutex; if threads of differing
 19896 processes attempt to operate on such a mutex, the behaviour is undefined. The default value of
 19897 the attribute is PTHREAD_PROCESS_PRIVATE.

19898 **RETURN VALUE**

19899 Upon successful completion, *pthread_mutexattr_setpshared()* returns zero. Otherwise, an error
 19900 number is returned to indicate the error.

19901 Upon successful completion, *pthread_mutexattr_getpshared()* returns zero and stores the value of
 19902 the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise,
 19903 an error number is returned to indicate the error.

19904 **ERRORS**

19905 The *pthread_mutexattr_getpshared()* and *pthread_mutexattr_setpshared()* functions may fail if:

19906 [EINVAL] The value specified by *attr* is invalid.

19907 The *pthread_mutexattr_setpshared()* function may fail if:

19908 [EINVAL] The new value specified for the attribute is outside the range of legal values
 19909 for that attribute.

19910 These functions will not return an error code of [EINTR].

19911 **EXAMPLES**

19912 None.

19913 **APPLICATION USAGE**

19914 None.

19915 **FUTURE DIRECTIONS**

19916 None.

19917 **SEE ALSO**

19918 *pthread_create()*, *pthread_mutex_init()*, *pthread_mutexattr_init()*, *pthread_cond_init()*, <pthread.h>.

19919 CHANGE HISTORY

19920 First released in Issue 5.

19921 Included for alignment with the POSIX Threads Extension.

19922 **NAME**

19923 pthread_mutexattr_init, pthread_mutexattr_destroy — initialise and destroy mutex attributes
 19924 object

19925 **SYNOPSIS**

19926 #include <pthread.h>

19927 int pthread_mutexattr_init(pthread_mutexattr_t *attr);

19928 int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

19929 **DESCRIPTION**

19930 The function *pthread_mutexattr_init()* initialises a mutex attributes object *attr* with the default
 19931 value for all of the attributes defined by the implementation.

19932 The effect of initialising an already initialised mutex attributes object is undefined.

19933 After a mutex attributes object has been used to initialise one or more mutexes, any function
 19934 affecting the attributes object (including destruction) does not affect any previously initialised
 19935 mutexes.

19936 The *pthread_mutexattr_destroy()* function destroys a mutex attributes object; the object becomes,
 19937 in effect, uninitialised. An implementation may cause *pthread_mutexattr_destroy()* to set the
 19938 object referenced by *attr* to an invalid value. A destroyed mutex attributes object can be re-
 19939 initialised using *pthread_mutexattr_init()*; the results of otherwise referencing the object after it
 19940 has been destroyed are undefined.

19941 **RETURN VALUE**

19942 Upon successful completion, *pthread_mutexattr_init()* and *pthread_mutexattr_destroy()* return
 19943 zero. Otherwise, an error number is returned to indicate the error.

19944 **ERRORS**

19945 The *pthread_mutexattr_init()* function may fail if:

19946 [ENOMEM] Insufficient memory exists to initialise the mutex attributes object.

19947 The *pthread_mutexattr_destroy()* function may fail if:

19948 [EINVAL] The value specified by *attr* is invalid.

19949 These functions will not return an error code of [EINTR].

19950 **EXAMPLES**

19951 None.

19952 **APPLICATION USAGE**

19953 None.

19954 **FUTURE DIRECTIONS**

19955 None.

19956 **SEE ALSO**

19957 *pthread_create()*, *pthread_mutex_init()*, *pthread_mutexattr_init()*, *pthread_cond_init()*, <pthread.h>.

19958 **CHANGE HISTORY**

19959 First released in Issue 5.

19960 Included for alignment with the POSIX Threads Extension.

19961 **NAME**

19962 pthread_mutexattr_setprioceiling, pthread_mutexattr_getprioceiling — set and get prioceiling
 19963 attribute of mutex attribute object (**REALTIME THREADS**)

19964 **SYNOPSIS**

19965 RTT `#include <pthread.h>`

```
19966 int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,  
19967     int prioceiling);  
19968 int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *attr,  
19969     int *prioceiling);  
19970
```

19971 **DESCRIPTION**

19972 The *pthread_mutexattr_setprioceiling()* and *pthread_mutexattr_getprioceiling()* functions,
 19973 respectively, set and get the priority ceiling attribute of a mutex attribute object pointed to by
 19974 *attr* which was previously created by the function *pthread_mutexattr_init()*.

19975 The *prioceiling* attribute contains the priority ceiling of initialised mutexes. The values of
 19976 *prioceiling* will be within the maximum range of priorities defined by SCHED_FIFO.

19977 The *prioceiling* attribute defines the priority ceiling of initialised mutexes, which is the minimum
 19978 priority level at which the critical section guarded by the mutex is executed. In order to avoid
 19979 priority inversion, the priority ceiling of the mutex will be set to a priority higher than or equal
 19980 to the highest priority of all the threads that may lock that mutex. The values of *prioceiling* will
 19981 be within the maximum range of priorities defined under the SCHED_FIFO scheduling policy.

19982 **RETURN VALUE**

19983 Upon successful completion, the *pthread_mutexattr_setprioceiling()* and
 19984 *pthread_mutexattr_getprioceiling()* functions return zero. Otherwise, an error number is returned
 19985 to indicate the error.

19986 **ERRORS**

19987 The *pthread_mutexattr_setprioceiling()* and *pthread_mutexattr_getprioceiling()* functions will fail if:

19988 [ENOSYS] The option _POSIX_THREAD_PRIO_PROTECT is not defined and the
 19989 implementation does not support the function.

19990 The *pthread_mutexattr_setprioceiling()* and *pthread_mutexattr_getprioceiling()* functions may fail if:

19991 [EINVAL] The value specified by *attr* or *prioceiling* is invalid.

19992 [EPERM] The caller does not have the privilege to perform the operation.

19993 **EXAMPLES**

19994 None.

19995 **APPLICATION USAGE**

19996 None.

19997 **FUTURE DIRECTIONS**

19998 None.

19999 **SEE ALSO**

20000 *pthread_create()*, *pthread_mutex_init()*, *pthread_cond_init()*, **<pthread.h>**.

20001 **CHANGE HISTORY**

20002 First released in Issue 5.

20003 Included for alignment with the POSIX Threads Extension.

20004 Marked as part of the Realtime Threads Feature Group.

20005 NAME

20006 pthread_mutexattr_setprotocol, pthread_mutexattr_getprotocol — set and get protocol attribute
 20007 of mutex attribute object (**REALTIME THREADS**)

20008 SYNOPSIS

20009 RTT `#include <pthread.h>`

```
20010 int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,  
20011     int protocol);  
20012 int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr,  
20013     int *protocol);  
20014
```

20015 DESCRIPTION

20016 The *pthread_mutexattr_setprotocol()* and *pthread_mutexattr_getprotocol()* functions, respectively,
 20017 set and get the protocol attribute of a mutex attribute object pointed to by *attr* which was
 20018 previously created by the function *pthread_mutexattr_init()*.

20019 The *protocol* attribute defines the protocol to be followed in utilising mutexes. The value of
 20020 *protocol* may be one of PTHREAD_PRIO_NONE, PTHREAD_PRIO_INHERIT or
 20021 PTHREAD_PRIO_PROTECT, which are defined by the header **<pthread.h>**.

20022 When a thread owns a mutex with the PTHREAD_PRIO_NONE protocol attribute, its priority
 20023 and scheduling are not affected by its mutex ownership.

20024 When a thread is blocking higher priority threads because of owning one or more mutexes with
 20025 the PTHREAD_PRIO_INHERIT protocol attribute, it executes at the higher of its priority or the
 20026 priority of the highest priority thread waiting on any of the mutexes owned by this thread and
 20027 initialised with this protocol.

20028 When a thread owns one or more mutexes initialised with the PTHREAD_PRIO_PROTECT
 20029 protocol, it executes at the higher of its priority or the highest of the priority ceilings of all the
 20030 mutexes owned by this thread and initialised with this attribute, regardless of whether other
 20031 threads are blocked on any of these mutexes or not.

20032 While a thread is holding a mutex which has been initialised with the PRIO_INHERIT or
 20033 PRIO_PROTECT protocol attributes, it will not be subject to being moved to the tail of the
 20034 scheduling queue at its priority in the event that its original priority is changed, such as by a call
 20035 to *sched_setparam()*. Likewise, when a thread unlocks a mutex that has been initialised with the
 20036 PRIO_INHERIT or PRIO_PROTECT protocol attributes, it will not be subject to being moved to
 20037 the tail of the scheduling queue at its priority in the event that its original priority is changed.

20038 If a thread simultaneously owns several mutexes initialised with different protocols, it will
 20039 execute at the highest of the priorities that it would have obtained by each of these protocols.

20040 When a thread makes a call to *pthread_mutex_lock()*, if the symbol
 20041 **_POSIX_THREAD_PRIO_INHERIT** is defined and the mutex was initialised with the protocol
 20042 attribute having the value PTHREAD_PRIO_INHERIT, when the calling thread is blocked
 20043 because the mutex is owned by another thread, that owner thread will inherit the priority level
 20044 of the calling thread as long as it continues to own the mutex. The implementation updates its
 20045 execution priority to the maximum of its assigned priority and all its inherited priorities.
 20046 Furthermore, if this owner thread itself becomes blocked on another mutex, the same priority
 20047 inheritance effect will be propagated to this other owner thread, in a recursive manner.

20048 RETURN VALUE

20049 Upon successful completion, the *pthread_mutexattr_setprotocol()* and
 20050 *pthread_mutexattr_getprotocol()* functions return zero. Otherwise, an error number is returned to
 20051 indicate the error.

20052 **ERRORS**

20053 The *pthread_mutexattr_setprotocol()* and *pthread_mutexattr_getprotocol()* functions will fail if:

20054 [ENOSYS] Neither one of the options `_POSIX_THREAD_PRIO_PROTECT` and
 20055 `_POSIX_THREAD_PRIO_INHERIT` is defined and the implementation does
 20056 not support the function.

20057 [ENOTSUP] The value specified by *protocol* is an unsupported value.

20058 The *pthread_mutexattr_setprotocol()* and *pthread_mutexattr_getprotocol()* functions may fail if:

20059 [EINVAL] The value specified by *attr* ro *protocol* is invalid.

20060 [EPERM] The caller does not have the privilege to perform the operation.

20061 **EXAMPLES**

20062 None.

20063 **APPLICATION USAGE**

20064 None.

20065 **FUTURE DIRECTIONS**

20066 None.

20067 **SEE ALSO**

20068 *pthread_create()*, *pthread_mutex_init()*, *pthread_cond_init()*, `<pthread.h>`.

20069 **CHANGE HISTORY**

20070 First released in Issue 5.

20071 Included for alignment with the POSIX Threads Extension.

20072 Marked as part of the Realtime Threads Feature Group.

20073 **NAME**

20074 pthread_mutexattr_gettype, pthread_mutexattr_settype — get or set a mutex type

20075 **SYNOPSIS**20076 EX

```
#include <pthread.h>
```

20077

```
int pthread_mutexattr_gettype(pthread_mutexattr_t *attr, int *type);
```

20078

```
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

20079

20080 **DESCRIPTION**

20081 The *pthread_mutexattr_gettype()* and *pthread_mutexattr_settype()* functions respectively get and
 20082 set the mutex *type* attribute. This attribute is set in the *type* parameter to these functions. The
 20083 default value of the *type* attribute is PTHREAD_MUTEX_DEFAULT.

20084 The type of mutex is contained in the *type* attribute of the mutex attributes. Valid mutex types
 20085 include:

20086 PTHREAD_MUTEX_NORMAL

20087 This type of mutex does not detect deadlock. A thread attempting to relock this mutex
 20088 without first unlocking it will deadlock. Attempting to unlock a mutex locked by a different
 20089 thread results in undefined behaviour. Attempting to unlock an unlocked mutex results in
 20090 undefined behaviour.

20091 PTHREAD_MUTEX_ERRORCHECK

20092 This type of mutex provides error checking. A thread attempting to relock this mutex
 20093 without first unlocking it will return with an error. A thread attempting to unlock a mutex
 20094 which another thread has locked will return with an error. A thread attempting to unlock
 20095 an unlocked mutex will return with an error.

20096 PTHREAD_MUTEX_RECURSIVE

20097 A thread attempting to relock this mutex without first unlocking it will succeed in locking
 20098 the mutex. The relocking deadlock which can occur with mutexes of type
 20099 PTHREAD_MUTEX_NORMAL cannot occur with this type of mutex. Multiple locks of this
 20100 mutex require the same number of unlocks to release the mutex before another thread can
 20101 acquire the mutex. A thread attempting to unlock a mutex which another thread has locked
 20102 will return with an error. A thread attempting to unlock an unlocked mutex will return with
 20103 an error.

20104 PTHREAD_MUTEX_DEFAULT

20105 Attempting to recursively lock a mutex of this type results in undefined behaviour.
 20106 Attempting to unlock a mutex of this type which was not locked by the calling thread
 20107 results in undefined behaviour. Attempting to unlock a mutex of this type which is not
 20108 locked results in undefined behaviour. An implementation is allowed to map this mutex to
 20109 one of the other mutex types.

20110 **RETURN VALUE**

20111 If successful, the *pthread_mutexattr_settype()* function returns zero. Otherwise, an error number
 20112 is returned to indicate the error.

20113 Upon successful completion, the *pthread_mutexattr_gettype()* function returns zero and stores the
 20114 value of the *type* attribute of *attr* into the object referenced by the *type* parameter. Otherwise an
 20115 error is returned to indicate the error.

20116 **ERRORS**

20117 The *pthread_mutexattr_gettype()* and *pthread_mutexattr_settype()* functions will fail if:

20118 [EINVAL] The value *type* is invalid.

20119 The *pthread_mutexattr_gettype()* and *pthread_mutexattr_settype()* functions may fail if:

20120 [EINVAL] The value specified by *attr* is invalid.

20121 **EXAMPLES**

20122 None.

20123 **APPLICATION USAGE**

20124 It is advised that an application should not use a PTHREAD_MUTEX_RECURSIVE mutex with
20125 condition variables because the implicit unlock performed for a *pthread_cond_wait()* or
20126 *pthread_cond_timedwait()* may not actually release the mutex (if it had been locked multiple
20127 times). If this happens, no other thread can satisfy the condition of the predicate.

20128 **FUTURE DIRECTIONS**

20129 None.

20130 **SEE ALSO**

20131 *pthread_cond_wait()*, *pthread_cond_timedwait()*, <pthread.h>.

20132 **CHANGE HISTORY**

20133 First released in Issue 5.

20134 **NAME**

20135 pthread_once — dynamic package initialisation

20136 **SYNOPSIS**

20137 #include <pthread.h>

```
20138 int pthread_once(pthread_once_t *once_control,  
20139 void (*init_routine)(void));  
20140 pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

20141 **DESCRIPTION**

20142 The first call to *pthread_once()* by any thread in a process, with a given *once_control*, will call the
20143 *init_routine()* with no arguments. Subsequent calls of *pthread_once()* with the same *once_control*
20144 will not call the *init_routine()*. On return from *pthread_once()*, it is guaranteed that *init_routine()*
20145 has completed. The *once_control* parameter is used to determine whether the associated
20146 initialisation routine has been called.

20147 The function *pthread_once()* is not a cancellation point. However, if *init_routine()* is a
20148 cancellation point and is canceled, the effect on *once_control* is as if *pthread_once()* was never
20149 called.

20150 The constant PTHREAD_ONCE_INIT is defined by the header <pthread.h>.

20151 The behaviour of *pthread_once()* is undefined if *once_control* has automatic storage duration or is
20152 not initialised by PTHREAD_ONCE_INIT.

20153 **RETURN VALUE**

20154 Upon successful completion, *pthread_once()* returns zero. Otherwise, an error number is
20155 returned to indicate the error.

20156 **ERRORS**

20157 No errors are defined.

20158 The *pthread_once()* function will not return an error code of [EINTR].

20159 **EXAMPLES**

20160 None.

20161 **APPLICATION USAGE**

20162 None.

20163 **FUTURE DIRECTIONS**

20164 None.

20165 **SEE ALSO**

20166 <pthread.h>.

20167 **CHANGE HISTORY**

20168 First released in Issue 5.

20169 Included for alignment with the POSIX Threads Extension.

20170 **NAME**

20171 pthread_rwlock_init, pthread_rwlock_destroy — initialise or destroy a read-write lock object

20172 **SYNOPSIS**

20173 EX #include <pthread.h>

```
20174 int pthread_rwlock_init(pthread_rwlock_t *rwlock,
20175     const pthread_rwlockattr_t *attr);
20176 int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
20177 pthread_rwlock_t rwlock=PTHREAD_RWLOCK_INITIALIZER;
20178
```

20179 **DESCRIPTION**

20180 The *pthread_rwlock_init()* function initialises the read-write lock referenced by *rwlock* with the
 20181 attributes referenced by *attr*. If *attr* is NULL, the default read-write lock attributes are used; the
 20182 effect is the same as passing the address of a default read-write lock attributes object. Once
 20183 initialised, the lock can be used any number of times without being re-initialised. Upon
 20184 successful initialisation, the state of the read-write lock becomes initialised and unlocked.
 20185 Results are undefined if *pthread_rwlock_init()* is called specifying an already initialised read-
 20186 write lock. Results are undefined if a read-write lock is used without first being initialised.

20187 If the *pthread_rwlock_init()* function fails, *rwlock* is not initialised and the contents of *rwlock* are
 20188 undefined.

20189 The *pthread_rwlock_destroy()* function destroys the read-write lock object referenced by *rwlock*
 20190 and releases any resources used by the lock. The effect of subsequent use of the lock is undefined
 20191 until the lock is re-initialised by another call to *pthread_rwlock_init()*. An implementation may
 20192 cause *pthread_rwlock_destroy()* to set the object referenced by *rwlock* to an invalid value. Results
 20193 are undefined if *pthread_rwlock_destroy()* is called when any thread holds *rwlock*. Attempting to
 20194 destroy an uninitialised read-write lock results in undefined behaviour. A destroyed read-write
 20195 lock object can be re-initialised using *pthread_rwlock_init()*; the results of otherwise referencing
 20196 the read-write lock object after it has been destroyed are undefined.

20197 In cases where default read-write lock attributes are appropriate, the macro
 20198 PTHREAD_RWLOCK_INITIALIZER can be used to initialise read-write locks that are statically
 20199 allocated. The effect is equivalent to dynamic initialisation by a call to *pthread_rwlock_init()* with
 20200 the parameter *attr* specified as NULL, except that no error checks are performed.

20201 **RETURN VALUE**

20202 If successful, the *pthread_rwlock_init()* and *pthread_rwlock_destroy()* functions return zero.
 20203 Otherwise, an error number is returned to indicate the error. The [EBUSY] and [EINVAL] error
 20204 checks, if implemented, will act as if they were performed immediately at the beginning of
 20205 processing for the function and caused an error return prior to modifying the state of the read-
 20206 write lock specified by *rwlock*.

20207 **ERRORS**20208 The *pthread_rwlock_init()* function will fail if:

20209 [EAGAIN] The system lacked the necessary resources (other than memory) to initialise
 20210 another read-write lock.

20211 [ENOMEM] Insufficient memory exists to initialise the read-write lock.

20212 [EPERM] The caller does not have the privilege to perform the operation.

20213 The *pthread_rwlock_init()* function may fail if:

20214 [EBUSY] The implementation has detected an attempt to re-initialise the object
 20215 referenced by *rwlock*, a previously initialised but not yet destroyed read-write

20216		lock.	
20217	[EINVAL]	The value specified by <i>attr</i> is invalid.	
20218		The <i>pthread_rwlock_destroy()</i> function may fail if:	
20219	[EBUSY]	The implementation has detected an attempt to destroy the object referenced	
20220		by <i>rwlock</i> while it is locked.	
20221	[EINVAL]	The value specified by <i>attr</i> is invalid.	
20222	EXAMPLES		
20223		None.	
20224	APPLICATION USAGE		
20225		Similar functions are being developed by IEEE PASC. In keeping with its objective of ensuring	
20226		that CAE Specifications are fully aligned with formal standards, The Open Group intends to add	
20227		any new interfaces adopted by an official IEEE standard in this area.	
20228	FUTURE DIRECTIONS		
20229		None.	
20230	SEE ALSO		
20231		<pthread.h>, <i>pthread_rwlock_rdlock()</i> , <i>pthread_rwlock_wrlock()</i> , <i>pthread_rwlockattr_init()</i> ,	
20232		<i>pthread_rwlock_unlock()</i> .	
20233	CHANGE HISTORY		
20234		First released in Issue 5.	

20235 **NAME**

20236 pthread_rwlock_rdlock, pthread_rwlock_tryrdlock — lock a read-write lock object for reading

20237 **SYNOPSIS**

20238 EX #include <pthread.h>

20239 int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);

20240 int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

20241

20242 **DESCRIPTION**

20243 The *pthread_rwlock_rdlock()* function applies a read lock to the read-write lock referenced by
 20244 *rwlock*. The calling thread acquires the read lock if a writer does not hold the lock and there are
 20245 no writers blocked on the lock. It is unspecified whether the calling thread acquires the lock
 20246 when a writer does not hold the lock and there are writers waiting for the lock. If a writer holds
 20247 the lock, the calling thread will not acquire the read lock. If the read lock is not acquired, the
 20248 calling thread blocks (that is, it does not return from the *pthread_rwlock_rdlock()* call) until it can
 20249 acquire the lock. Results are undefined if the calling thread holds a write lock on *rwlock* at the
 20250 time the call is made.

20251 Implementations are allowed to favour writers over readers to avoid writer starvation.

20252 A thread may hold multiple concurrent read locks on *rwlock* (that is, successfully call the
 20253 *pthread_rwlock_rdlock()* function *n* times). If so, the thread must perform matching unlocks (that
 20254 is, it must call the *pthread_rwlock_unlock()* function *n* times).

20255 The function *pthread_rwlock_tryrdlock()* applies a read lock as in the *pthread_rwlock_rdlock()*
 20256 function with the exception that the function fails if any thread holds a write lock on *rwlock* or
 20257 there are writers blocked on *rwlock*.

20258 Results are undefined if any of these functions are called with an uninitialised read-write lock.

20259 If a signal is delivered to a thread waiting for a read-write lock for reading, upon return from the
 20260 signal handler the thread resumes waiting for the read-write lock for reading as if it was not
 20261 interrupted.

20262 **RETURN VALUE**

20263 If successful, the *pthread_rwlock_rdlock()* function returns zero. Otherwise, an error number is
 20264 returned to indicate the error.

20265 The function *pthread_rwlock_tryrdlock()* returns zero if the lock for reading on the read-write lock
 20266 object referenced by *rwlock* is acquired. Otherwise an error number is returned to indicate the
 20267 error.

20268 **ERRORS**20269 The *pthread_rwlock_tryrdlock()* function will fail if:

20270 [EBUSY] The read-write lock could not be acquired for reading because a writer holds
 20271 the lock or was blocked on it.

20272 The *pthread_rwlock_rdlock()* and *pthread_rwlock_tryrdlock()* functions may fail if:

20273 [EINVAL] The value specified by *rwlock* does not refer to an initialised read-write lock
 20274 object.

20275 [EDEADLK] The current thread already owns the read-write lock for writing.

20276 [EAGAIN] The read lock could not be acquired because the maximum number of read
 20277 locks for *rwlock* has been exceeded.

20278 EXAMPLES

20279 None.

20280 APPLICATION USAGE

20281 Similar functions are being developed by IEEE PASC. In keeping with its objective of ensuring
20282 that CAE Specifications are fully aligned with formal standards, The Open Group intends to add
20283 any new interfaces adopted by an official IEEE standard in this area.

20284 Realtime applications may encounter priority inversion when using read-write locks. The
20285 problem occurs when a high priority thread “locks” a read-write lock that is about to be
20286 “unlocked” by a low priority thread, but the low priority thread is preempted by a medium
20287 priority thread. This scenario leads to priority inversion; a high priority thread is blocked by
20288 lower priority threads for an unlimited period of time. During system design, realtime
20289 programmers must take into account the possibility of this kind of priority inversion. They can
20290 deal with it in a number of ways, such as by having critical sections that are guarded by read-
20291 write locks execute at a high priority, so that a thread cannot be preempted while executing in its
20292 critical section.

20293 FUTURE DIRECTIONS

20294 None.

20295 SEE ALSO

20296 <pthread.h>, pthread_rwlock_init(), pthread_rwlock_wrlock(), pthread_rwlockattr_init(),
20297 pthread_rwlock_unlock().

20298 CHANGE HISTORY

20299 First released in Issue 5.

20300 **NAME**

20301 pthread_rwlock_unlock — unlock a read-write lock object

20302 **SYNOPSIS**20303 EX `#include <pthread.h>`20304 `int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);`

20305

20306 **DESCRIPTION**

20307 The *pthread_rwlock_unlock()* function is called to release a lock held on the read-write lock object
 20308 referenced by *rwlock*. Results are undefined if the read-write lock *rwlock* is not held by the calling
 20309 thread.

20310 If this function is called to release a read lock from the read-write lock object and there are other
 20311 read locks currently held on this read-write lock object, the read-write lock object remains in the
 20312 read locked state. If this function releases the calling thread's last read lock on this read-write
 20313 lock object, then the calling thread is no longer one of the owners of the object. If this function
 20314 releases the last read lock for this read-write lock object, the read-write lock object will be put in
 20315 the unlocked state with no owners.

20316 If this function is called to release a write lock for this read-write lock object, the read-write lock
 20317 object will be put in the unlocked state with no owners.

20318 If the call to the *pthread_rwlock_unlock()* function results in the read-write lock object becoming
 20319 unlocked and there are multiple threads waiting to acquire the read-write lock object for writing,
 20320 the scheduling policy is used to determine which thread acquires the read-write lock object for
 20321 writing. If there are multiple threads waiting to acquire the read-write lock object for reading,
 20322 the scheduling policy is used to determine the order in which the waiting threads acquire the
 20323 read-write lock object for reading. If there are multiple threads blocked on *rwlock* for both read
 20324 locks and write locks, it is unspecified whether the readers acquire the lock first or whether a
 20325 writer acquires the lock first.

20326 Results are undefined if any of these functions are called with an uninitialised read-write lock.

20327 **RETURN VALUE**

20328 If successful, the *pthread_rwlock_unlock()* function returns zero. Otherwise, an error number is
 20329 returned to indicate the error.

20330 **ERRORS**

20331 The *pthread_rwlock_unlock()* function may fail if:

20332 [EINVAL] The value specified by *rwlock* does not refer to an initialised read-write lock
 20333 object.

20334 [EPERM] The current thread does not own the read-write lock.

20335 **EXAMPLES**

20336 None.

20337 **APPLICATION USAGE**

20338 Similar functions are being developed by IEEE PASC. In keeping with its objective of ensuring
 20339 that CAE Specifications are fully aligned with formal standards, The Open Group intends to add
 20340 any new interfaces adopted by an official IEEE standard in this area.

20341 **FUTURE DIRECTIONS**

20342 None.

20343 **SEE ALSO**

20344 <pthread.h>, pthread_rwlock_init(), pthread_rwlock_wrlock(), pthread_rwlockattr_init(),
20345 pthread_rwlock_rdlock().

20346 **CHANGE HISTORY**

20347 First released in Issue 5.

20348 **NAME**

20349 pthread_rwlock_wrlock, pthread_rwlock_trywrlock — lock a read-write lock object for writing

20350 **SYNOPSIS**

20351 EX #include <pthread.h>

20352 int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

20353 int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);

20354

20355 **DESCRIPTION**

20356 The *pthread_rwlock_wrlock()* function applies a write lock to the read-write lock referenced by
 20357 *rwlock*. The calling thread acquires the write lock if no other thread (reader or writer) holds the
 20358 read-write lock *rwlock*. Otherwise, the thread blocks (that is, does not return from the
 20359 *pthread_rwlock_wrlock()* call) until it can acquire the lock. Results are undefined if the calling
 20360 thread holds the read-write lock (whether a read or write lock) at the time the call is made.

20361 Implementations are allowed to favour writers over readers to avoid writer starvation.

20362 The function *pthread_rwlock_trywrlock()* applies a write lock like the *pthread_rwlock_wrlock()*
 20363 function, with the exception that the function fails if any thread currently holds *rwlock* (for
 20364 reading or writing).

20365 Results are undefined if any of these functions are called with an uninitialised read-write lock.

20366 If a signal is delivered to a thread waiting for a read-write lock for writing, upon return from the
 20367 signal handler the thread resumes waiting for the read-write lock for writing as if it was not
 20368 interrupted.

20369 **RETURN VALUE**

20370 If successful, the *pthread_rwlock_wrlock()* function returns zero. Otherwise, an error number is
 20371 returned to indicate the error.

20372 The function *pthread_rwlock_trywrlock()* returns zero if the lock for writing on the read-write lock
 20373 object referenced by *rwlock* is acquired. Otherwise an error number is returned to indicate the
 20374 error.

20375 **ERRORS**20376 The *pthread_rwlock_trywrlock()* function will fail if:

20377 [EBUSY] The read-write lock could not be acquired for writing because it was already
 20378 locked for reading or writing.

20379 The *pthread_rwlock_wrlock()* and *pthread_rwlock_trywrlock()* functions may fail if:

20380 [EINVAL] The value specified by *rwlock* does not refer to an initialised read-write lock
 20381 object.

20382 [EDEADLK] The current thread already owns the read-write lock for writing or reading.

20383 **EXAMPLES**

20384 None.

20385 **APPLICATION USAGE**

20386 Similar functions are being developed by IEEE PASC. In keeping with its objective of ensuring
 20387 that CAE Specifications are fully aligned with formal standards, The Open Group intends to add
 20388 any new interfaces adopted by an official IEEE standard in this area.

20389 Realtime applications may encounter priority inversion when using read-write locks. The
 20390 problem occurs when a high priority thread “locks” a read-write lock that is about to be
 20391 “unlocked” by a low priority thread, but the low priority thread is preempted by a medium

20392 priority thread. This scenario leads to priority inversion; a high priority thread is blocked by
20393 lower priority threads for an unlimited period of time. During system design, realtime
20394 programmers must take into account the possibility of this kind of priority inversion. They can
20395 deal with it in a number of ways, such as by having critical sections that are guarded by read-
20396 write locks execute at a high priority, so that a thread cannot be preempted while executing in its
20397 critical section.

20398 FUTURE DIRECTIONS

20399 None.

20400 SEE ALSO

20401 <pthread.h>, *pthread_rwlock_init()*, *pthread_rwlock_unlock()*, *pthread_rwlockattr_init()*,
20402 *pthread_rwlock_rdlock()*.

20403 CHANGE HISTORY

20404 First released in Issue 5.

20405 **NAME**

20406 pthread_rwlockattr_getpshared, pthread_rwlockattr_setpshared — get and set process-shared
 20407 attribute of read-write lock attributes object

20408 **SYNOPSIS**

```
20409 EX      #include <pthread.h>

20410      int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr,
20411      int *pshared);
20412      int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
20413      int pshared);
20414
```

20415 **DESCRIPTION**

20416 The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a read-write lock
 20417 to be operated upon by any thread that has access to the memory where the read-write lock is
 20418 allocated, even if the read-write lock is allocated in memory that is shared by multiple processes.
 20419 If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the read-write lock will only be
 20420 operated upon by threads created within the same process as the thread that initialised the
 20421 read-write lock; if threads of differing processes attempt to operate on such a read-write lock,
 20422 the behaviour is undefined. The default value of the *process-shared* attribute is
 20423 PTHREAD_PROCESS_PRIVATE.

20424 The *pthread_rwlockattr_getpshared()* function obtains the value of the *process-shared* attribute from
 20425 the initialised attributes object referenced by *attr*. The *pthread_rwlockattr_setpshared()* function is
 20426 used to set the *process-shared* attribute in an initialised attributes object referenced by *attr*.

20427 **RETURN VALUE**

20428 If successful, the *pthread_rwlockattr_setpshared()* function returns zero. Otherwise, an error
 20429 number is returned to indicate the error.

20430 Upon successful completion, the *pthread_rwlockattr_getpshared()* returns zero and stores the
 20431 value of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter.
 20432 Otherwise an error number is returned to indicate the error.

20433 **ERRORS**

20434 The *pthread_rwlockattr_getpshared()* and *pthread_rwlockattr_setpshared()* functions may fail if:

20435 [EINVAL] The value specified by *attr* is invalid.

20436 The *pthread_rwlockattr_setpshared()* function may fail if:

20437 [EINVAL] The new value specified for the attribute is outside the range of legal values
 20438 for that attribute.

20439 **EXAMPLES**

20440 None.

20441 **APPLICATION USAGE**

20442 Similar functions are being developed by IEEE PASC. In keeping with its objective of ensuring
 20443 that CAE Specifications are fully aligned with formal standards, The Open Group intends to add
 20444 any new interfaces adopted by an official IEEE standard in this area.

20445 **FUTURE DIRECTIONS**

20446 None.

20447 **SEE ALSO**

20448 <pthread.h>, pthread_rwlock_init(), pthread_rwlock_unlock(), pthread_rwlock_wrlock(),
 20449 pthread_rwlock_rdlock(), pthread_rwlockattr_init().

20450 **CHANGE HISTORY**

20451 First released in Issue 5.

20452 **NAME**

20453 pthread_rwlockattr_init, pthread_rwlockattr_destroy — initialise and destroy read-write lock
 20454 attributes object

20455 **SYNOPSIS**

20456 EX `#include <pthread.h>`

```
20457 int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
20458 int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
20459
```

20460 **DESCRIPTION**

20461 The function *pthread_rwlockattr_init()* initialises a read-write lock attributes object *attr* with the
 20462 default value for all of the attributes defined by the implementation.

20463 Results are undefined if *pthread_rwlockattr_init()* is called specifying an already initialised read-
 20464 write lock attributes object.

20465 After a read-write lock attributes object has been used to initialise one or more read-write locks,
 20466 any function affecting the attributes object (including destruction) does not affect any previously
 20467 initialised read-write locks.

20468 The *pthread_rwlockattr_destroy()* function destroys a read-write lock attributes object. The effect
 20469 of subsequent use of the object is undefined until the object is re-initialised by another call to
 20470 *pthread_rwlockattr_init()*. An implementation may cause *pthread_rwlockattr_destroy()* to set the
 20471 object referenced by *attr* to an invalid value.

20472 **RETURN VALUE**

20473 If successful, the *pthread_rwlockattr_init()* and *pthread_rwlockattr_destroy()* functions return zero.
 20474 Otherwise, an error number is returned to indicate the error.

20475 **ERRORS**

20476 The *pthread_rwlockattr_init()* function will fail if:

20477 [ENOMEM] Insufficient memory exists to initialise the read-write lock attributes object.

20478 The *pthread_rwlockattr_destroy()* function may fail if:

20479 [EINVAL] The value specified by *attr* is invalid.

20480 **EXAMPLES**

20481 None.

20482 **APPLICATION USAGE**

20483 Similar functions are being developed by IEEE PASC. In keeping with its objective of ensuring
 20484 that CAE Specifications are fully aligned with formal standards, The Open Group intends to add
 20485 any new interfaces adopted by an official IEEE standard in this area.

20486 **FUTURE DIRECTIONS**

20487 None.

20488 **SEE ALSO**

20489 `<pthread.h>`, *pthread_rwlock_init()*, *pthread_rwlock_unlock()*, *pthread_rwlock_wrlock()*,
 20490 *pthread_rwlock_rdlock()*, *pthread_rwlockattr_getpshared()*.

20491 **CHANGE HISTORY**

20492 First released in Issue 5.

20493 **NAME**

20494 pthread_self — get calling thread's ID

20495 **SYNOPSIS**

20496 #include <pthread.h>

20497 pthread_t pthread_self(void);

20498 **DESCRIPTION**20499 The *pthread_self()* function returns the thread ID of the calling thread.20500 **RETURN VALUE**

20501 See DESCRIPTION above.

20502 **ERRORS**

20503 No errors are defined.

20504 The *pthread_self()* function will not return an error code of [EINTR].20505 **EXAMPLES**

20506 None.

20507 **APPLICATION USAGE**

20508 None.

20509 **FUTURE DIRECTIONS**

20510 None.

20511 **SEE ALSO**20512 *pthread_create()*, *pthread_equal()*, <pthread.h>.20513 **CHANGE HISTORY**

20514 First released in Issue 5.

20515 Included for alignment with the POSIX Threads Extension.

20516 **NAME**

20517 pthread_setcancelstate, pthread_setcanceltype, pthread_testcancel — set cancelability state

20518 **SYNOPSIS**

20519 #include <pthread.h>

20520 int pthread_setcancelstate(int *state*, int **oldstate*);

20521 int pthread_setcanceltype(int *type*, int **oldtype*);

20522 void pthread_testcancel(void);

20523 **DESCRIPTION**

20524 The *pthread_setcancelstate()* function atomically both sets the calling thread's cancelability state
 20525 to the indicated *state* and returns the previous cancelability state at the location referenced by
 20526 *oldstate*. Legal values for *state* are PTHREAD_CANCEL_ENABLE and
 20527 PTHREAD_CANCEL_DISABLE.

20528 The *pthread_setcanceltype()* function atomically both sets the calling thread's cancelability type to
 20529 the indicated *type* and returns the previous cancelability type at the location referenced by
 20530 *oldtype*. Legal values for *type* are PTHREAD_CANCEL_DEFERRED and
 20531 PTHREAD_CANCEL_ASYNCHRONOUS.

20532 The cancelability state and type of any newly created threads, including the thread in which
 20533 *main()* was first invoked, are PTHREAD_CANCEL_ENABLE and
 20534 PTHREAD_CANCEL_DEFERRED respectively.

20535 The *pthread_testcancel()* function creates a cancellation point in the calling thread. The
 20536 *pthread_testcancel()* function has no effect if cancelability is disabled.

20537 **RETURN VALUE**

20538 If successful, the *pthread_setcancelstate()* and *pthread_setcanceltype()* functions return zero.
 20539 Otherwise, an error number is returned to indicate the error.

20540 **ERRORS**

20541 The *pthread_setcancelstate()* function may fail if:

20542 [EINVAL] The specified state is not PTHREAD_CANCEL_ENABLE or
 20543 PTHREAD_CANCEL_DISABLE.

20544 The *pthread_setcanceltype()* function may fail if:

20545 [EINVAL] The specified type is not PTHREAD_CANCEL_DEFERRED or
 20546 PTHREAD_CANCEL_ASYNCHRONOUS.

20547 These functions will not return an error code of [EINTR].

20548 **EXAMPLES**

20549 None.

20550 **APPLICATION USAGE**

20551 None.

20552 **FUTURE DIRECTIONS**

20553 None.

20554 **SEE ALSO**

20555 *pthread_cancel()*, <pthread.h>.

20556 **CHANGE HISTORY**

20557 First released in Issue 5.

20558 Included for alignment with the POSIX Threads Extension.

20559 **NAME**

20560 pthread_setconcurrency — get or set level of concurrency

20561 **SYNOPSIS**20562 EX `#include <pthread.h>`20563 `int pthread_setconcurrency(int new_level);`

20564

20565 **DESCRIPTION**20566 Refer to *pthread_getconcurrency()*.20567 **CHANGE HISTORY**

20568 First released in Issue 5.

20569 **NAME**

20570 pthread_setspecific, pthread_getspecific — thread-specific data management

20571 **SYNOPSIS**

20572 #include <pthread.h>

20573 int pthread_setspecific(pthread_key_t key, const void *value);

20574 void *pthread_getspecific(pthread_key_t key);

20575 **DESCRIPTION**

20576 The *pthread_setspecific()* function associates a thread-specific *value* with a *key* obtained via a
20577 previous call to *pthread_key_create()*. Different threads may bind different values to the same
20578 key. These values are typically pointers to blocks of dynamically allocated memory that have
20579 been reserved for use by the calling thread.

20580 The *pthread_getspecific()* function returns the value currently bound to the specified *key* on behalf
20581 of the calling thread.

20582 The effect of calling *pthread_setspecific()* or *pthread_getspecific()* with a *key* value not obtained
20583 from *pthread_key_create()* or after *key* has been deleted with *pthread_key_delete()* is undefined.

20584 Both *pthread_setspecific()* and *pthread_getspecific()* may be called from a thread-specific data
20585 destructor function. However, calling *pthread_setspecific()* from a destructor may result in lost
20586 storage or infinite loops.

20587 Both functions may be implemented as macros.

20588 **RETURN VALUE**

20589 The function *pthread_getspecific()* returns the thread-specific data value associated with the given
20590 *key*. If no thread-specific data value is associated with *key*, then the value NULL is returned.

20591 If successful, the *pthread_setspecific()* function returns zero. Otherwise, an error number is
20592 returned to indicate the error.

20593 **ERRORS**

20594 The *pthread_setspecific()* function will fail if:

20595 [ENOMEM] Insufficient memory exists to associate the value with the key.

20596 The *pthread_setspecific()* function may fail if:

20597 [EINVAL] The key value is invalid.

20598 No errors are returned from *pthread_getspecific()*.

20599 These functions will not return an error code of [EINTR].

20600 **EXAMPLES**

20601 None.

20602 **APPLICATION USAGE**

20603 None.

20604 **FUTURE DIRECTIONS**

20605 None.

20606 **SEE ALSO**

20607 *pthread_key_create()*, <pthread.h>.

20608 **CHANGE HISTORY**

20609 First released in Issue 5.

20610 Included for alignment with the POSIX Threads Extension.

20611 **NAME**

20612 pthread_sigmask — examine and change blocked signals

20613 **SYNOPSIS**

20614 #include <signal.h>

20615 int pthread_sigmask(int *how*, const sigset_t **set*, sigset_t **oset* *FP*);

20616 **DESCRIPTION**

20617 Refer to *sigprocmask*().

20618 **CHANGE HISTORY**

20619 First released in Issue 5.

20620 Included for alignment with the POSIX Threads Extension.

20621 **NAME**

20622 ptsname — get name of the slave pseudo-terminal device

20623 **SYNOPSIS**20624 EX `#include <stdlib.h>`20625 `char *ptsname(int fildev);`

20626

20627 **DESCRIPTION**

20628 The *ptsname()* function returns the name of the slave pseudo-terminal device associated with a
 20629 master pseudo-terminal device. The *fildev* argument is a file descriptor that refers to the master
 20630 device. The *ptsname()* function returns a pointer to a string containing the pathname of the
 20631 corresponding slave device.

20632 This interface need not be reentrant.

20633 **RETURN VALUE**

20634 Upon successful completion, *ptsname()* returns a pointer to a string which is the name of the
 20635 pseudo-terminal slave device. Upon failure, *ptsname()* returns a null pointer. This could occur if
 20636 *fildev* is an invalid file descriptor or if the slave device name does not exist in the file system.

20637 **ERRORS**

20638 No errors are defined.

20639 **EXAMPLES**

20640 None.

20641 **APPLICATION USAGE**20642 The value returned may point to a static data area that is overwritten by each call to *ptsname()*.20643 **FUTURE DIRECTIONS**

20644 None.

20645 **SEE ALSO**20646 *grantpt()*, *open()*, *ttyname()*, *unlockpt()*, *<stdlib.h>*.20647 **CHANGE HISTORY**

20648 First released in Issue 4, Version 2.

20649 **Issue 5**

20650 Moved from X/OPEN UNIX extension to BASE.

20651 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

20652 **NAME**

20653 putc — put byte on a stream

20654 **SYNOPSIS**

20655 #include <stdio.h>

20656 int putc(int *c*, FILE **stream*);

20657 **DESCRIPTION**

20658 The *putc()* function is equivalent to *fputc()*, except that if it is implemented as a macro it may
20659 evaluate *stream* more than once, so the argument should never be an expression with side-
20660 effects.

20661 **RETURN VALUE**

20662 Refer to *fputc()*.

20663 **ERRORS**

20664 Refer to *fputc()*.

20665 **EXAMPLES**

20666 None.

20667 **APPLICATION USAGE**

20668 Because it may be implemented as a macro, *putc()* may treat a *stream* argument with side-effects
20669 incorrectly. In particular, *putc(c, *f++)* will not necessarily work correctly. Therefore, use of this
20670 interface is not recommended in such situations; *fputc()* should be used instead.

20671 **FUTURE DIRECTIONS**

20672 None.

20673 **SEE ALSO**

20674 *fputc()*, <stdio.h>.

20675 **CHANGE HISTORY**

20676 First released in Issue 1.

20677 Derived from Issue 1 of the SVID.

20678 **Issue 4**

20679 The following change is incorporated for alignment with the ISO C standard:

- 20680 • The *c* argument is not allowed to be evaluated more than once.

20681 Another change is incorporated as follows:

- 20682 • The APPLICATION USAGE section now states that the use of this function is not
20683 recommended with a *stream* argument with side effects.

20684 **NAME**20685 putchar — put byte on *stdout* stream20686 **SYNOPSIS**

20687 #include <stdio.h>

20688 int putchar(int *c*);20689 **DESCRIPTION**20690 The function call *putchar(c)* is equivalent to *putc(c, stdout)*.20691 **RETURN VALUE**20692 Refer to *fputc()*.20693 **ERRORS**20694 Refer to *fputc()*.20695 **EXAMPLES**

20696 None.

20697 **APPLICATION USAGE**

20698 None.

20699 **FUTURE DIRECTIONS**

20700 None.

20701 **SEE ALSO**20702 *putc()*, <stdio.h>.20703 **CHANGE HISTORY**

20704 First released in Issue 1.

20705 Derived from Issue 1 of the SVID.

20706 NAME

20707 putc_unlocked — stdio with explicit client locking

20708 SYNOPSIS

20709 #include <stdio.h>

20710 int putc_unlocked(int *c*, FILE **stream*);

20711 DESCRIPTION

20712 Refer to *getc_unlocked()*.

20713 CHANGE HISTORY

20714 First released in Issue 5.

20715 Included for alignment with the POSIX Threads Extension.

20716 **NAME**

20717 putchar_unlocked — stdio with explicit client locking

20718 **SYNOPSIS**

20719 #include <stdio.h>

20720 int putchar_unlocked(int *c*);20721 **DESCRIPTION**20722 Refer to *getc_unlocked()*.20723 **CHANGE HISTORY**

20724 First released in Issue 5.

20725 Included for alignment with the POSIX Threads Extension.

20726 **NAME**

20727 putenv — change or add a value to environment

20728 **SYNOPSIS**20729 EX `#include <stdlib.h>`20730 `int putenv(char *string);`

20731

20732 **DESCRIPTION**

20733 The *putenv()* function uses the *string* argument to set environment variable values. The *string*
 20734 argument should point to a string of the form "*name=value*". The *putenv()* function makes the
 20735 value of the environment variable *name* equal to *value* by altering an existing variable or creating
 20736 a new one. In either case, the string pointed to by *string* becomes part of the environment, so
 20737 altering the string will change the environment. The space used by *string* is no longer used once
 20738 a new string-defining *name* is passed to *putenv()*.

20739 This interface need not be reentrant.

20740 **RETURN VALUE**

20741 Upon successful completion, *putenv()* returns 0. Otherwise, it returns a non-zero value and sets
 20742 *errno* to indicate the error.

20743 **ERRORS**20744 The *putenv()* function may fail if:

20745 [ENOMEM] Insufficient memory was available.

20746 **EXAMPLES**

20747 None.

20748 **APPLICATION USAGE**

20749 The *putenv()* function manipulates the environment pointed to by *environ*, and can be used in
 20750 conjunction with *getenv()*.

20751 This routine may use *malloc()* to enlarge the environment.

20752 A potential error is to call *putenv()* with an automatic variable as the argument, then return from
 20753 the calling function while *string* is still part of the environment.

20754 **FUTURE DIRECTIONS**

20755 None.

20756 **SEE ALSO**20757 *exec*, *getenv()*, *malloc()*, <stdlib.h>.20758 **CHANGE HISTORY**

20759 First released in Issue 1.

20760 Derived from Issue 1 of the SVID.

20761 **Issue 4**

20762 The following changes are incorporated in this issue:

- 20763 • The <stdlib.h> header is added to the SYNOPSIS section.
- 20764 • The type of argument *string* is changed from **char *** to **const char ***.

20765 **Issue 5**

20766 The type of the argument to this function is changed from **const char*** to **char***. This was
20767 indicated as a FUTURE DIRECTION in previous issues.

20768 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

20769 NAME

20770 putmsg, putpmsg — send a message on a STREAM

20771 SYNOPSIS

20772 EX

```
#include <stropts.h>
```

```
20773 int putmsg(int fildes, const struct strbuf *ctlptr,
20774           const struct strbuf *dataptr, int flags);
20775 int putpmsg(int fildes, const struct strbuf *ctlptr,
20776            const struct strbuf *dataptr, int band, int flags);
20777
```

20778 DESCRIPTION

20779 The *putmsg()* function creates a message from a process buffer(s) and sends the message to a
 20780 STREAMS file. The message may contain either a data part, a control part, or both. The data
 20781 and control parts are distinguished by placement in separate buffers, as described below. The
 20782 semantics of each part is defined by the STREAMS module that receives the message.

20783 The *putpmsg()* function does the same thing as *putmsg()*, but the process can send messages in
 20784 different priority bands. Except where noted, all requirements on *putmsg()* also pertain to
 20785 *putpmsg()*.

20786 The *fildes* argument specifies a file descriptor referencing an open STREAM. The *ctlptr* and
 20787 *dataptr* arguments each point to a **strbuf** structure.

20788 The *ctlptr* argument points to the structure describing the control part, if any, to be included in
 20789 the message. The *buf* member in the **strbuf** structure points to the buffer where the control
 20790 information resides, and the *len* member indicates the number of bytes to be sent. The *maxlen*
 20791 member is not used by *putmsg()*. In a similar manner, the argument *dataptr* specifies the data, if
 20792 any, to be included in the message. The *flags* argument indicates what type of message should
 20793 be sent and is described further below.

20794 To send the data part of a message, *dataptr* must not be a null pointer and the *len* member of
 20795 *dataptr* must be 0 or greater. To send the control part of a message, the corresponding values
 20796 must be set for *ctlptr*. No data (control) part will be sent if either *dataptr* (*ctlptr*) is a null pointer
 20797 or the *len* member of *dataptr* (*ctlptr*) is set to -1.

20798 For *putmsg()*, if a control part is specified and *flags* is set to RS_HIPRI, a high priority message is
 20799 sent. If no control part is specified, and *flags* is set to RS_HIPRI, *putmsg()* fails and sets *errno* to
 20800 [EINVAL]. If *flags* is set to 0, a normal message (priority band equal to 0) is sent. If a control part
 20801 and data part are not specified and *flags* is set to 0, no message is sent and 0 is returned.

20802 For *putpmsg()*, the flags are different. The *flags* argument is a bitmask with the following
 20803 mutually-exclusive flags defined: MSG_HIPRI and MSG_BAND. If *flags* is set to 0, *putpmsg()*
 20804 fails and sets *errno* to [EINVAL]. If a control part is specified and *flags* is set to MSG_HIPRI and
 20805 *band* is set to 0, a high-priority message is sent. If *flags* is set to MSG_HIPRI and either no control
 20806 part is specified or *band* is set to a non-zero value, *putpmsg()* fails and sets *errno* to [EINVAL]. If
 20807 *flags* is set to MSG_BAND, then a message is sent in the priority band specified by *band*. If a
 20808 control part and data part are not specified and *flags* is set to MSG_BAND, no message is sent
 20809 and 0 is returned.

20810 The *putmsg()* function blocks if the STREAM write queue is full due to internal flow control
 20811 conditions, with the following exceptions:

- 20812 • For high-priority messages, *putmsg()* does not block on this condition and continues
 20813 processing the message.

20814 • For other messages, *putmsg()* does not block but fails when the write queue is full and
 20815 O_NONBLOCK is set.

20816 The *putmsg()* function also blocks, unless prevented by lack of internal resources, while waiting
 20817 for the availability of message blocks in the STREAM, regardless of priority or whether
 20818 O_NONBLOCK has been specified. No partial message is sent.

20819 RETURN VALUE

20820 Upon successful completion, *putmsg()* and *putpmsg()* return 0. Otherwise, they return -1 and
 20821 set *errno* to indicate the error.

20822 ERRORS

20823 The *putmsg()* and *putpmsg()* functions will fail if:

20824 [EAGAIN] A non-priority message was specified, the O_NONBLOCK flag is set, and the
 20825 STREAM write queue is full due to internal flow control conditions; or buffers
 20826 could not be allocated for the message that was to be created.

20827 [EBADF] *fildev* is not a valid file descriptor open for writing.

20828 [EINTR] A signal was caught during *putmsg()*.

20829 [EINVAL] An undefined value is specified in *flags*, or *flags* is set to RS_HIPRI or
 20830 MSG_HIPRI and no control part is supplied, or the STREAM or multiplexer
 20831 referenced by *fildev* is linked (directly or indirectly) downstream from a
 20832 multiplexer, or *flags* is set to MSG_HIPRI and *band* is non-zero (for *putpmsg()*
 20833 only).

20834 [ENOSR] Buffers could not be allocated for the message that was to be created due to
 20835 insufficient STREAMS memory resources.

20836 [ENOSTR] A STREAM is not associated with *fildev*.

20837 [ENXIO] A hangup condition was generated downstream for the specified STREAM.

20838 [EPIPE] or [EIO] The *fildev* argument refers to a STREAMS-based pipe and the other end of the
 20839 pipe is closed. A SIGPIPE signal is generated for the calling thread.

20840 [ERANGE] The size of the data part of the message does not fall within the range
 20841 specified by the maximum and minimum packet sizes of the topmost
 20842 STREAM module. This value is also returned if the control part of the
 20843 message is larger than the maximum configured size of the control part of a
 20844 message, or if the data part of a message is larger than the maximum
 20845 configured size of the data part of a message.

20846 In addition, *putmsg()* and *putpmsg()* will fail if the STREAM head had processed an
 20847 asynchronous error before the call. In this case, the value of *errno* does not reflect the result of
 20848 *putmsg()* or *putpmsg()* but reflects the prior error.

20849 EXAMPLES

20850 None.

20851 APPLICATION USAGE

20852 None.

20853 FUTURE DIRECTIONS

20854 None.

20855 SEE ALSO

20856 *getmsg()*, *poll()*, *read()*, *write()*, <stropts.h>, Section 2.5 on page 34.

20857 **CHANGE HISTORY**

20858 First released in Issue 4, Version 2.

20859 **Issue 5**

20860 Moved from X/OPEN UNIX extension to BASE.

20861 The following line of text is removed from the DESCRIPTION: "The STREAM head guarantees
20862 that the control part of a message generated by *putmsg()* is at least 64 bytes in length".

20863 **NAME**

20864 puts — put a string on standard output

20865 **SYNOPSIS**

20866 #include <stdio.h>

20867 int puts(const char *s);

20868 **DESCRIPTION**20869 The *puts()* function writes the string pointed to by *s*, followed by a newline character, to the
20870 standard output stream *stdout*. The terminating null byte is not written.20871 The *st_ctime* and *st_mtime* fields of the file will be marked for update between the successful
20872 execution of *puts()* and the next successful completion of a call to *fflush()* or *fclose()* on the same
20873 stream or a call to *exit()* or *abort()*.20874 **RETURN VALUE**20875 Upon successful completion, *puts()* returns a non-negative number. Otherwise it returns EOF,
20876 sets an error indicator for the stream and *errno* is set to indicate the error.20877 **ERRORS**20878 Refer to *fputc()*.20879 **EXAMPLES**

20880 None.

20881 **APPLICATION USAGE**20882 The *puts()* function appends a newline character, while *fputs()* does not.20883 **FUTURE DIRECTIONS**

20884 None.

20885 **SEE ALSO**20886 *fputs()*, *fopen()*, *putc()*, *stdio()*, <stdio.h>.20887 **CHANGE HISTORY**

20888 First released in Issue 1.

20889 Derived from Issue 1 of the SVID.

20890 **Issue 4**

20891 The following change is incorporated for alignment with the ISO C standard:

- 20892 • The type of argument
- s*
- is changed from
- char ***
- to
- const char ***
- .

20893 Another change is incorporated as follows:

- 20894 • In the DESCRIPTION, the words “null character” are replaced by “null byte”.

20895 **NAME**

20896 pututxline — put an entry into user accounting database

20897 **SYNOPSIS**

20898 EX #include <utmpx.h>

20899 struct utmpx *pututxline(const struct utmpx *utmpx);

20900

20901 **DESCRIPTION**20902 Refer to *endutxent()*.20903 **CHANGE HISTORY**

20904 First released in Issue 4, Version 2.

20905 **Issue 5**

20906 Moved from X/OPEN UNIX extension to BASE.

20907 **NAME**20908 putw — put a word on a stream (**LEGACY**)20909 **SYNOPSIS**20910 EX `#include <stdio.h>`20911 `int putw(int w, FILE *stream);`

20912

20913 **DESCRIPTION**

20914 The *putw()* function writes the word (that is, type **int**) *w* to the output *stream* (at the position at
 20915 which the file offset, if defined, is pointing). The size of a word is the size of a type **int** and
 20916 varies from machine to machine. The *putw()* function neither assumes nor causes special
 20917 alignment in the file.

20918 The *st_ctime* and *st_mtime* fields of the file will be marked for update between the successful
 20919 execution of *putw()* and the next successful completion of a call to *fflush()* or *fclose()* on the same
 20920 stream or a call to *exit()* or *abort()*.

20921 This interface need not be reentrant.

20922 **RETURN VALUE**

20923 Upon successful completion, *putw()* returns 0. Otherwise, a non-zero value is returned, the error
 20924 indicators for the stream are set, and *errno* is set to indicate the error.

20925 **ERRORS**20926 Refer to *fputc()*.20927 **EXAMPLES**

20928 None.

20929 **APPLICATION USAGE**

20930 Because of possible differences in word length and byte ordering, files written using *putw()* are
 20931 implementation-dependent, and possibly cannot be read using *getw()* by a different application
 20932 or by the same application on a different processor.

20933 The *putw()* function is inherently byte stream oriented and is not tenable in the context of either
 20934 multibyte character streams or wide-character streams. Application programmers are
 20935 recommended to use one of the character based output functions instead.

20936 **FUTURE DIRECTIONS**

20937 None.

20938 **SEE ALSO**20939 *fopen()*, *fwrite()*, *getw()*, *<stdio.h>*.20940 **CHANGE HISTORY**

20941 First released in Issue 1.

20942 Derived from Issue 1 of the SVID.

20943 **Issue 5**

20944 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

20945 Marked LEGACY.

20946 **NAME**

20947 putwc — put a wide-character on a stream

20948 **SYNOPSIS**

20949 #include <stdio.h>

20950 #include <wchar.h>

20951 wint_t putwc(wchar_t *wc*, FILE **stream*);20952 **DESCRIPTION**

20953 The *putwc()* function is equivalent to *fputwc()*, except that if it is implemented as a macro it may
20954 evaluate *stream* more than once, so the argument should never be an expression with side-
20955 effects.

20956 **RETURN VALUE**20957 Refer to *fputwc()*.20958 **ERRORS**20959 Refer to *fputwc()*.20960 **EXAMPLES**

20961 None.

20962 **APPLICATION USAGE**

20963 Because it may be implemented as a macro, *putwc()* may treat a *stream* argument with side-
20964 effects incorrectly. In particular, *putwc(wc, *f++)* need not work correctly. Therefore, use of this
20965 interface is not recommended; *fputwc()* should be used instead.

20966 **FUTURE DIRECTIONS**

20967 None.

20968 **SEE ALSO**20969 *fputwc()*, <stdio.h>, <wchar.h>.20970 **CHANGE HISTORY**

20971 First released as a World-wide Portability Interface in Issue 4.

20972 **Issue 5**

20973 Aligned with ISO/IEC 9899:1990/Amendment 1:1994 (E). Specifically, the type of argument *wc*
20974 is changed from **wint_t** to **wchar_t**.

20975 The Optional Header (OH) marking is removed from <stdio.h>.

20976 **NAME**

20977 putwchar — put a wide-character on stdout stream

20978 **SYNOPSIS**

20979 #include <wchar.h>

20980 wint_t putwchar(wchar_t wc);

20981 **DESCRIPTION**

20982 The function call *putwchar(wc)* is equivalent to *putwc(wc, stdout)*.

20983 **RETURN VALUE**

20984 Refer to *fputwc()*.

20985 **ERRORS**

20986 Refer to *fputwc()*.

20987 **EXAMPLES**

20988 None.

20989 **APPLICATION USAGE**

20990 None.

20991 **FUTURE DIRECTIONS**

20992 None.

20993 **SEE ALSO**

20994 *fputwc()*, *putwc()*, <wchar.h>.

20995 **CHANGE HISTORY**

20996 First released in Issue 4.

20997 **Issue 5**

20998 Aligned with ISO/IEC 9899:1990/Amendment 1:1994 (E). Specifically, the type of argument *wc*
20999 is changed from **wint_t** to **wchar_t**.

21000 **NAME**

21001 prwrite — write on a file

21002 **SYNOPSIS**

21003 EX #include <unistd.h>

```
21004       ssize_t prwrite(int fildev, const void *buf, size_t nbyte,  
21005                       off_t offset);
```

21006

21007 **DESCRIPTION**21008 Refer to *write()*.21009 **CHANGE HISTORY**

21010 First released in Issue 5.

21011 **NAME**

21012 qsort — sort a table of data

21013 **SYNOPSIS**

21014 #include <stdlib.h>

```
21015       void qsort(void *base, size_t nel, size_t width
21016                   int (*compar)(const void *, const void *));
```

21017 **DESCRIPTION**

21018 The *qsort()* function sorts an array of *nel* objects, the initial element of which is pointed to by
 21019 *base*. The size of each object, in bytes, is specified by the *width* argument.

21020 The contents of the array are sorted in ascending order according to a comparison function. The
 21021 *compar* argument is a pointer to the comparison function, which is called with two arguments
 21022 that point to the elements being compared. The function must return an integer less than, equal
 21023 to, or greater than 0, if the first argument is considered respectively less than, equal to, or greater
 21024 than the second. If two members compare as equal, their order in the sorted array is unspecified.

21025 **RETURN VALUE**21026 The *qsort()* function returns no value.21027 **ERRORS**

21028 No errors are defined.

21029 **EXAMPLES**

21030 None.

21031 **APPLICATION USAGE**

21032 The comparison function need not compare every byte, so arbitrary data may be contained in
 21033 the elements in addition to the values being compared.

21034 **FUTURE DIRECTIONS**

21035 None.

21036 **SEE ALSO**

21037 <stdlib.h>.

21038 **CHANGE HISTORY**

21039 First released in Issue 1.

21040 Derived from Issue 1 of the SVID.

21041 **Issue 4**

21042 The following change is incorporated for alignment with the ISO C standard:

- 21043 • The arguments to *compar()* are formally defined in the SYNOPSIS section.

21044 **NAME**

21045 raise — send a signal to the executing process

21046 **SYNOPSIS**

21047 #include <signal.h>

21048 int raise(int *sig*);

21049 **DESCRIPTION**

21050 The *raise()* function sends the signal *sig* to the executing thread.

21051 The effect of the *raise()* function is equivalent to calling:

21052 pthread_kill(pthread_self(), *sig*);

21053 **RETURN VALUE**

21054 EX Upon successful completion, 0 is returned. Otherwise, a non-zero value is returned and *errno* is
21055 set to indicate the error.

21056 **ERRORS**

21057 The *raise()* function will fail if:

21058 EX [EINVAL] The value of the *sig* argument is an invalid signal number.

21059 **EXAMPLES**

21060 None.

21061 **APPLICATION USAGE**

21062 None.

21063 **FUTURE DIRECTIONS**

21064 None.

21065 **SEE ALSO**

21066 *kill()*, *sigaction()*, <signal.h>, <sys/types.h>.

21067 **CHANGE HISTORY**

21068 First released in Issue 4.

21069 Derived from the ANSI C standard.

21070 **Issue 5**

21071 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

21072 **NAME**

21073 rand, rand_r — pseudo-random number generator

21074 **SYNOPSIS**

21075 #include <stdlib.h>

21076 int rand (void);

21077 void srand(unsigned int *seed*);21078 int rand_r(unsigned int **seed*);21079 **DESCRIPTION**

21080 The *rand()* function computes a sequence of pseudo-random integers in the range 0 to
 21081 EX {RAND_MAX} with a period of at least 2^{32} .

21082 The *srand()* function uses the argument as a seed for a new sequence of pseudo-random
 21083 numbers to be returned by subsequent calls to *rand()*. If *srand()* is then called with the same
 21084 seed value, the sequence of pseudo-random numbers will be repeated. If *rand()* is called before
 21085 any calls to *srand()* are made, the same sequence will be generated as when *srand()* is first called
 21086 with a seed value of 1.

21087 The implementation will behave as if no function defined in this document calls *rand()* or *srand*.

21088 The *rand()* interface need not be reentrant.

21089 The *rand_r()* function computes a sequence of pseudo-random integers in the range 0 to
 21090 {RAND_MAX}. (The value of the {RAND_MAX} macro will be at least 32767.)

21091 If *rand_r()* is called with the same initial value for the object pointed to by *seed* and that object is
 21092 not modified between successive returns and calls to *rand_r()*, the same sequence shall be
 21093 generated.

21094 **RETURN VALUE**

21095 The *rand()* function returns the next pseudo-random number in the sequence. The *srand()*
 21096 function returns no value.

21097 The *rand_r()* function returns a pseudo-random integer.

21098 **ERRORS**

21099 No errors are defined.

21100 **EXAMPLES**

21101 None.

21102 **APPLICATION USAGE**

21103 The *drand48()* function provides a much more elaborate random number generator.

21104 The following code defines a pair of functions which could be incorporated into applications
 21105 wishing to ensure that the same sequence of numbers is generated across different machines:

```

21106     static unsigned long int next = 1;
21107     int myrand(void)      /* RAND_MAX assumed to be 32767 */
21108     {
21109         next = next * 1103515245 + 12345;
21110         return((unsigned int)(next/65536) % 32768);
21111     }
21112     void mysrand(unsigned int seed)
21113     {
21114         next = seed;
21115     }

```

21116 FUTURE DIRECTIONS

21117 None.

21118 SEE ALSO

21119 *drand48()*, *srand()*, <stdlib.h>.

21120 CHANGE HISTORY

21121 First released in Issue 1.

21122 Derived from Issue 1 of the SVID.

21123 Issue 4

21124 The following changes are incorporated for alignment with the ISO C standard:

- 21125 • The argument list of *rand()* is explicitly defined as **void**.
- 21126 • The argument *seed* is explicitly defined as **unsigned int**.

21127 Other changes are incorporated as follows:

- 21128 • The definition of *srand()* is added to the SYNOPSIS section.
- 21129 • In the DESCRIPTION, the text referring to the period of pseudo-random numbers is marked
- 21130 as an extension.
- 21131 • The example in the APPLICATION USAGE section is updated (a) to use ISO C syntax, and
- 21132 (b) to avoid name clashes with standard functions.

21133 Issue 5

21134 The *rand_r()* function is included for alignment with the POSIX Threads Extension.

21135 A note indicating that the *rand()* interface need not be reentrant is added to the DESCRIPTION.

21136 **NAME**

21137 random — generate pseudorandom number

21138 **SYNOPSIS**

21139 EX #include <stdlib.h>

21140 long random(void);

21141

21142 **DESCRIPTION**21143 Refer to *initstate()*.21144 **CHANGE HISTORY**

21145 First released in Issue 4, Version 2.

21146 **Issue 5**

21147 Moved from X/OPEN UNIX extension to BASE.

21148 **NAME**

21149 read, readv, pread — read from a file

21150 **SYNOPSIS**

21151 #include <unistd.h>

21152 ssize_t read(int *fildev*, void **buf*, size_t *nbyte*);21153 EX ssize_t pread(int *fildev*, void **buf*, size_t *nbyte*, off_t *offset*);

21154 #include <sys/uio.h>

21155 ssize_t readv(int *fildev*, const struct iovec **iov*, int *iovcnt*);

21156

21157 **DESCRIPTION**21158 The *read()* function attempts to read *nbyte* bytes from the file associated with the open file
21159 descriptor, *fildev*, into the buffer pointed to by *buf*.21160 If *nbyte* is 0, *read()* will return 0 and have no other results.21161 On files that support seeking (for example, a regular file), the *read()* starts at a position in the file
21162 given by the file offset associated with *fildev*. The file offset is incremented by the number of
21163 bytes actually read.21164 Files that do not support seeking, for example, terminals, always read from the current position.
21165 The value of a file offset associated with such a file is undefined.21166 No data transfer will occur past the current end-of-file. If the starting position is at or after the
21167 end-of-file, 0 will be returned. If the file refers to a device special file, the result of subsequent
21168 *read()* requests is implementation-dependent.21169 If the value of *nbyte* is greater than {SSIZE_MAX}, the result is implementation-dependent.

21170 When attempting to read from an empty pipe or FIFO:

- 21171 • If no process has the pipe open for writing, *read()* will return 0 to indicate end-of-file.
- 21172 • If some process has the pipe open for writing and O_NONBLOCK is set, *read()* will return -1
21173 and set *errno* to [EAGAIN].
- 21174 • If some process has the pipe open for writing and O_NONBLOCK is clear, *read()* will block
21175 the calling thread until some data is written or the pipe is closed by all processes that had the
21176 pipe open for writing.

21177 When attempting to read a file (other than a pipe or FIFO) that supports non-blocking reads and
21178 has no data currently available:

- 21179 • If O_NONBLOCK is set, *read()* will return a -1 and set *errno* to [EAGAIN].
- 21180 • If O_NONBLOCK is clear, *read()* will block the calling thread until some data becomes
21181 available.
- 21182 • The use of the O_NONBLOCK flag has no effect if there is some data available.

21183 The *read()* function reads data previously written to a file. If any portion of a regular file prior to
21184 the end-of-file has not been written, *read()* returns bytes with value 0. For example, *lseek()*
21185 allows the file offset to be set beyond the end of existing data in the file. If data is later written at
21186 this point, subsequent reads in the gap between the previous end of data and the newly written
21187 data will return bytes with value 0 until data is written into the gap.

21188	Upon successful completion, where <i>nbyte</i> is greater than 0, <i>read()</i> will mark for update the <i>st_atime</i> field of the file, and return the number of bytes read. This number will never be greater
21189	than <i>nbyte</i> . The value returned may be less than <i>nbyte</i> if the number of bytes left in the file is less
21190	than <i>nbyte</i> , if the <i>read()</i> request was interrupted by a signal, or if the file is a pipe or FIFO or
21191	special file and has fewer than <i>nbyte</i> bytes immediately available for reading. For example, a
21192	<i>read()</i> from a file associated with a terminal may return one typed line of data.
21193	
21194	If a <i>read()</i> is interrupted by a signal before it reads any data, it will return <code>-1</code> with <i>errno</i> set to
21195	<code>[EINTR]</code> .
21196 FIPS	If a <i>read()</i> is interrupted by a signal after it has successfully read some data, it will return the
21197	number of bytes read.
21198 EX	A <i>read()</i> from a STREAMS file can read data in three different modes: byte-stream mode,
21199	message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This
21200	can be changed using the <code>I_SRDOPT ioctl()</code> request, and can be tested with the <code>I_GRDOPT</code>
21201	<code>ioctl()</code> . In byte-stream mode, <i>read()</i> retrieves data from the STREAM until as many bytes as
21202	were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode
21203	ignores message boundaries.
21204	In STREAMS message-nondiscard mode, <i>read()</i> retrieves data until as many bytes as were
21205	requested are transferred, or until a message boundary is reached. If <i>read()</i> does not retrieve all
21206	the data in a message, the remaining data is left on the STREAM, and can be retrieved by the
21207	next <i>read()</i> call. Message-discard mode also retrieves data until as many bytes as were
21208	requested are transferred, or a message boundary is reached. However, unread data remaining
21209	in a message after the <i>read()</i> returns is discarded, and is not available for a subsequent <i>read()</i> ,
21210	<i>readv()</i> or <i>getmsg()</i> call.
21211	How <i>read()</i> handles zero-byte STREAMS messages is determined by the current read mode
21212	setting. In byte-stream mode, <i>read()</i> accepts data until it has read <i>nbyte</i> bytes, or until there is no
21213	more data to read, or until a zero-byte message block is encountered. The <i>read()</i> function then
21214	returns the number of bytes read, and places the zero-byte message back on the STREAM to be
21215	retrieved by the next <i>read()</i> , <i>readv()</i> or <i>getmsg()</i> . In message-nondiscard mode or message-
21216	discard mode, a zero-byte message returns 0 and the message is removed from the STREAM.
21217	When a zero-byte message is read as the first message on a STREAM, the message is removed
21218	from the STREAM and 0 is returned, regardless of the read mode.
21219	A <i>read()</i> from a STREAMS file returns the data in the message at the front of the STREAM head
21220	read queue, regardless of the priority band of the message.
21221	By default, STREAMs are in control-normal mode, in which a <i>read()</i> from a STREAMS file can
21222	only process messages that contain a data part but do not contain a control part. The <i>read()</i> fails
21223	if a message containing a control part is encountered at the STREAM head. This default action
21224	can be changed by placing the STREAM in either control-data mode or control-discard mode
21225	with the <code>I_SRDOPT ioctl()</code> command. In control-data mode, <i>read()</i> converts any control part to
21226	data and passes it to the application before passing any data part originally present in the same
21227	message. In control-discard mode, <i>read()</i> discards message control parts but returns to the
21228	process any data part in the message.
21229	In addition, <i>read()</i> and <i>readv()</i> will fail if the STREAM head had processed an asynchronous
21230	error before the call. In this case, the value of <i>errno</i> does not reflect the result of <i>read()</i> or <i>readv()</i>
21231	but reflects the prior error. If a hangup occurs on the STREAM being read, <i>read()</i> continues to
21232	operate normally until the STREAM head read queue is empty. Thereafter, it returns 0.
21233 EX	The <i>readv()</i> function is equivalent to <i>read()</i> , but places the input data into the <i>iovcnt</i> buffers
21234	specified by the members of the <i>iov</i> array: <i>iov</i> [0], <i>iov</i> [1], ..., <i>iov</i> [<i>iovcnt</i> -1]. The <i>iovcnt</i> argument is
21235	valid if greater than 0 and less than or equal to <code>{IOV_MAX}</code> .

21236		Each <i>iovec</i> entry specifies the base address and length of an area in memory where data should be placed. The <i>readv()</i> function always fills an area completely before proceeding to the next.
21237		
21238		Upon successful completion, <i>readv()</i> marks for update the <i>st_atime</i> field of the file.
21239	RT	If the Synchronized Input and Output option is supported:
21240		If the <i>O_DSYNC</i> and <i>O_RSYNC</i> bits have been set, read I/O operations on the file descriptor complete as defined by synchronised I/O data integrity completion. If the <i>O_SYNC</i> and <i>O_RSYNC</i> bits have been set, read I/O operations on the file descriptor complete as defined by synchronised I/O file integrity completion.
21241		
21242		
21243		
21244		If the Shared Memory Objects option is supported:
21245		If <i>fildes</i> refers to a shared memory object, the result of the <i>read()</i> function is unspecified.
21246	EX	For regular files, no data transfer will occur past the offset maximum established in the open file description associated with <i>fildes</i> .
21247		
21248		The <i>pread()</i> function performs the same action as <i>read()</i> , except that it reads from a given position in the file without changing the file pointer. The first three arguments to <i>pread()</i> are the same as <i>read()</i> with the addition of a fourth argument offset for the desired position inside the file. An attempt to perform a <i>pread()</i> on a file that is incapable of seeking results in an error.
21249		
21250		
21251		
21252		RETURN VALUE
21253	EX	Upon successful completion, <i>read()</i> , <i>pread()</i> and <i>readv()</i> return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions return -1 and set <i>errno</i> to indicate the error.
21254		
21255		
21256		ERRORS
21257	EX	The <i>read()</i> , <i>pread()</i> and <i>readv()</i> functions will fail if:
21258		[EAGAIN] The <i>O_NONBLOCK</i> flag is set for the file descriptor and the process would be delayed.
21259		
21260		[EBADF] The <i>fildes</i> argument is not a valid file descriptor open for reading.
21261	EX	[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.
21262		
21263		[EINTR] The read operation was terminated due to the receipt of a signal, and no data was transferred.
21264		
21265	EX	[EINVAL] The STREAM or multiplexer referenced by <i>fildes</i> is linked (directly or indirectly) downstream from a multiplexer.
21266		
21267	EX	[EIO] A physical I/O error has occurred.
21268		[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-dependent reasons.
21269		
21270		
21271		
21272	EX	[EISDIR] The <i>fildes</i> argument refers to a directory and the implementation does not allow the directory to be read using <i>read()</i> , <i>pread()</i> or <i>readv()</i> . The <i>readdir()</i> function should be used instead.
21273		
21274		
21275		[EOVERFLOW] The file is a regular file, <i>nbyte</i> 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 <i>fildes</i> .
21276		
21277		

21278	The <i>readv()</i> function will fail if:	
21279	[EINVAL]	The sum of the <i>iov_len</i> values in the <i>iov</i> array overflowed an <i>ssize_t</i> .
21280 EX	The <i>read()</i> , <i>pread()</i> and <i>readv()</i> functions may fail if:	
21281 EX	[ENXIO]	A request was made of a non-existent device, or the request was outside the capabilities of the device.
21282		
21283	The <i>readv()</i> function may fail if:	
21284	[EINVAL]	The <i>iovcnt</i> argument was less than or equal to 0, or greater than {IOV_MAX}.
21285	The <i>pread()</i> function will fail, and the file pointer remains unchanged, if:	
21286	[EINVAL]	The <i>offset</i> argument is invalid. The value is negative.
21287	[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.
21288		
21289	[ENXIO]	A request was outside the capabilities of the device.
21290	[ESPIPE]	<i>fildev</i> is associated with a pipe or FIFO.
21291		
21292	EXAMPLES	
21293	None.	
21294	APPLICATION USAGE	
21295	None.	
21296	FUTURE DIRECTIONS	
21297	None.	
21298	SEE ALSO	
21299	<i>fcntl()</i> , <i>ioctl()</i> , <i>lseek()</i> , <i>open()</i> , <i>pipe()</i> , <stropts.h>, <sys/uio.h>, <unistd.h>, XBD specification,	
21300	Chapter 9, General Terminal Interface.	
21301	CHANGE HISTORY	
21302	First released in Issue 1.	
21303	Derived from Issue 1 of the SVID.	
21304	Issue 4	
21305	The following changes are incorporated for alignment with the ISO POSIX-1 standard:	
21306	<ul style="list-style-type: none"> The type of the argument <i>buf</i> is changed from char * to void*, and the type of the argument <i>nbyte</i> is changed from unsigned to size_t. 	
21307		
21308	<ul style="list-style-type: none"> The DESCRIPTION now states that the result is implementation-dependent if <i>nbyte</i> is greater than {SSIZE_MAX}. This limit was defined by the constant {INT_MAX} in Issue 3. 	
21309		
21310	The following change is incorporated for alignment with the FIPS requirements:	
21311	<ul style="list-style-type: none"> The last paragraph of the DESCRIPTION now states that if <i>read()</i> is interrupted by a signal after it has successfully read some data, it will return the number of bytes read. In Issue 3 it was optional whether <i>read()</i> returned the number of bytes read, or whether it returned -1 with <i>errno</i> set to [EINTR]. 	
21312		
21313		
21314		

21315 Other changes are incorporated as follows:

- 21316 • The `<unistd.h>` header is added to the SYNOPSIS section.
- 21317 • The DESCRIPTION is rearranged for clarity and to align more closely with the ISO POSIX-1
- 21318 standard. No functional changes are made other than as noted elsewhere in this CHANGE
- 21319 HISTORY section.
- 21320 • In the ERRORS section in previous issues, generation of the [EIO] error depended on whether
- 21321 or not an implementation supported Job Control. This functionality is now defined as
- 21322 mandatory.
- 21323 • The [ENXIO] error is marked as an extension.
- 21324 • The APPLICATION USAGE section is removed.
- 21325 • The description of [EINTR] is amended.

21326 **Issue 4, Version 2**

21327 The following changes are incorporated for X/OPEN UNIX conformance:

- 21328 • The `readv()` function is added to the SYNOPSIS.
- 21329 • The DESCRIPTION is updated to describe the reading of data from STREAMS files. An
- 21330 operational description of the `readv()` function is also added.
- 21331 • References to the `readv()` function are added to the RETURN VALUE and ERRORS sections
- 21332 in appropriate places.
- 21333 • The ERRORS section has been restructured to describe errors that apply generally (that is, to
- 21334 both `read()` and `readv()`), and to describe those that apply to `readv()` specifically. The
- 21335 [EBADMSG], [EINVAL] and [EISDIR] errors are also added.

21336 **Issue 5**

21337 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX

21338 Threads Extension.

21339 Large File Summit extensions added.

21340 The `pread()` function is added.

21341 NAME

21342 readdir, readdir_r — read directory

21343 SYNOPSIS

21344 OH #include <sys/types.h>

21345 #include <dirent.h>

21346 struct dirent *readdir(DIR *dirp);

21347 int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);

21348 DESCRIPTION

21349 The type **DIR**, which is defined in the header **<dirent.h>**, represents a *directory stream*, which is
 21350 an ordered sequence of all the directory entries in a particular directory. Directory entries
 21351 represent files; files may be removed from a directory or added to a directory asynchronously to
 21352 the operation of *readdir()*.

21353 The *readdir()* function returns a pointer to a structure representing the directory entry at the
 21354 current position in the directory stream specified by the argument *dirp*, and positions the
 21355 directory stream at the next entry. It returns a null pointer upon reaching the end of the
 21356 directory stream. The structure *dirent* defined by the **<dirent.h>** header describes a directory
 21357 entry.

21358 EX If entries for dot or dot-dot exist, one entry will be returned for dot and one entry will be
 21359 returned for dot-dot; otherwise they will not be returned.

21360 The pointer returned by *readdir()* points to data which may be overwritten by another call to
 21361 *readdir()* on the same directory stream. This data is not overwritten by another call to *readdir()*
 21362 on a different directory stream.

21363 If a file is removed from or added to the directory after the most recent call to *opendir()* or
 21364 *rewinddir()*, whether a subsequent call to *readdir()* returns an entry for that file is unspecified.

21365 The *readdir()* function may buffer several directory entries per actual read operation; *readdir()*
 21366 marks for update the *st_atime* field of the directory each time the directory is actually read.

21367 After a call to *fork()*, either the parent or child (but not both) may continue processing the
 21368 EX directory stream using *readdir()*, *rewinddir()* or *seekdir()*. If both the parent and child processes
 21369 use these functions, the result is undefined.

21370 EX If the entry names a symbolic link, the value of the **d_ino** member is unspecified.

21371 The *readdir()* interface need not be reentrant.

21372 The *readdir_r()* function initialises the **dirent** structure referenced by *entry* to represent the
 21373 directory entry at the current position in the directory stream referred to by *dirp*, store a pointer
 21374 to this structure at the location referenced by *result*, and positions the directory stream at the
 21375 next entry.

21376 The storage pointed to by *entry* will be large enough for a **dirent** with an array of **char d_name**
 21377 member containing at least {NAME_MAX} plus one elements.

21378 On successful return, the pointer returned at **result* will the same value as the argument *entry*.
 21379 Upon reaching the end of the directory stream, this pointer will have the value NULL.

21380 The *readdir_r()* function will not return directory entries containing empty names. It is
 21381 unspecified whether entries are returned for dot or dot-dot.

21382 If a file is removed from or added to the directory after the most recent call to *opendir()* or
 21383 *rewinddir()*, whether a subsequent call to *readdir_r()* returns an entry for that file is unspecified.

21384 The *readdir_r()* function may buffer several directory entries per actual read operation; the
 21385 *readdir_r()* function marks for update the *st_atime* field of the directory each time the directory is
 21386 actually read.

21387 Applications wishing to check for error situations should set *errno* to 0 before calling *readdir()*. If
 21388 *errno* is set to non-zero on return, an error occurred.

21389 RETURN VALUE

21390 Upon successful completion, *readdir()* returns a pointer to an object of type **struct dirent**. When
 21391 an error is encountered, a null pointer is returned and *errno* is set to indicate the error. When the
 21392 end of the directory is encountered, a null pointer is returned and *errno* is not changed.

21393 If successful, the *readdir_r()* function returns zero. Otherwise, an error number is returned to
 21394 indicate the error.

21395 ERRORS

21396 EX The *readdir()* function will fail if:

21397 [EOVERFLOW] One of the values in the structure to be returned cannot be represented
 21398 correctly.

21399 The *readdir()* function may fail if:

21400 [EBADF] The *dirp* argument does not refer to an open directory stream.

21401 EX [ENOENT] The current position of the directory stream is invalid.

21402 The *readdir_r()* function may fail if:

21403 [EBADF] The *dirp* argument does not refer to an open directory stream.

21404 EXAMPLES

21405 The following sample code will search the current directory for the entry *name*:

```
21406     dirp = opendir(".");
21407     while (dirp) {
21408         errno = 0;
21409         if ((dp = readdir(dirp)) != NULL) {
21410             if (strcmp(dp->d_name, name) == 0) {
21411                 closedir(dirp);
21412                 return FOUND;
21413             }
21414         } else {
21415             if (errno == 0) {
21416                 closedir(dirp);
21417                 return NOT_FOUND;
21418             }
21419             closedir(dirp);
21420             return READ_ERROR;
21421         }
21422     }
21423     return OPEN_ERROR;
```

21424 APPLICATION USAGE

21425 The *readdir()* function should be used in conjunction with *opendir()*, *closedir()* and *rewinddir()* to
 21426 examine the contents of the directory.

21427 FUTURE DIRECTIONS

21428 None.

21429 **SEE ALSO**21430 `closedir()`, `lstat()`, `opendir()`, `rewinddir()`, `symlink()`, `<dirent.h>`, `<sys/types.h>`.21431 **CHANGE HISTORY**

21432 First released in Issue 2.

21433 **Issue 4**

21434 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 21435
- The last paragraph of the DESCRIPTION describing a restriction after `fork()` is added.

21436 Other changes are incorporated as follows:

- 21437 • The `<sys/types.h>` header is now marked as optional (OH); this header need not be included
- 21438 on XSI-conformant systems.
- 21439 • In the DESCRIPTION, the fact that XSI-conformant systems will return entries for dot and
- 21440 dot-dot is marked as an extension. This functionality is not specified in the ISO POSIX-1
- 21441 standard.
- 21442 • There is some rewording of the DESCRIPTION and RETURN VALUE sections. No
- 21443 functional changes are made other than as noted elsewhere in this CHANGE HISTORY
- 21444 section.

21445 **Issue 4, Version 2**

21446 The following changes are incorporated for X/OPEN UNIX conformance:

- 21447 • A statement is added to the DESCRIPTION indicating the disposition of certain fields in
- 21448 **struct dirent** when an entry refers to a symbolic link.
- 21449 • The [ENOENT] error is added to the ERRORS section as an optional error.

21450 **Issue 5**

21451 Large File Summit extensions added.

21452 The `readdir_r()` function is included for alignment with the POSIX Threads Extension.

21453 A note indicating that the `readdir()` interface need not be reentrant is added to the

21454 DESCRIPTION.

21455 **NAME**

21456 readlink — read the contents of a symbolic link

21457 **SYNOPSIS**21458 EX `#include <unistd.h>`21459 `int readlink(const char *path, char *buf, size_t bufsize);`

21460

21461 **DESCRIPTION**

21462 The *readlink()* function places the contents of the symbolic link referred to by *path* in the buffer
 21463 *buf* which has size *bufsize*. If the number of bytes in the symbolic link is less than *bufsize*, the
 21464 contents of the remainder of *buf* are unspecified.

21465 **RETURN VALUE**

21466 Upon successful completion, *readlink()* returns the count of bytes placed in the buffer.
 21467 Otherwise, it returns a value of -1 , leaves the buffer unchanged, and sets *errno* to indicate the
 21468 error.

21469 **ERRORS**21470 The *readlink()* function will fail if:21471 [EACCES] Search permission is denied for a component of the path prefix of *path*.21472 [EINVAL] The *path* argument names a file that is not a symbolic link.

21473 [EIO] An I/O error occurred while reading from the file system.

21474 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.21475 [ELOOP] Too many symbolic links were encountered in resolving *path*.

21476 [ENAMETOOLONG]

21477 The length of *path* exceeds {PATH_MAX}, or a pathname component is longer
 21478 than {NAME_MAX}.

21479 [ENOTDIR] A component of the path prefix is not a directory.

21480 The *readlink()* function may fail if:

21481 [EACCES] Read permission is denied for the directory.

21482 [ENAMETOOLONG]

21483 Pathname resolution of a symbolic link produced an intermediate result
 21484 whose length exceeds {PATH_MAX}.

21485 **EXAMPLES**

21486 None.

21487 **APPLICATION USAGE**

21488 Portable applications should not assume that the returned contents of the symbolic link are
 21489 null-terminated.

21490 **FUTURE DIRECTIONS**

21491 The return value may change in a future issue to align with IEEE PASC.

21492 **SEE ALSO**21493 *stat()*, *symlink()*, *<unistd.h>*.21494 **CHANGE HISTORY**

21495 First released in Issue 4, Version 2.

21496 **Issue 5**

21497 Moved from X/OPEN UNIX extension to BASE.

|

21498 **NAME**

21499 readv — vectored read from file

21500 **SYNOPSIS**21501 EX `#include <sys/uio.h>`21502 `ssize_t readv(int filides, const struct iovec *iov, int iovcnt);`

21503

21504 **DESCRIPTION**21505 Refer to *read()*.21506 **CHANGE HISTORY**

21507 First released in Issue 4, Version 2.

21508 **NAME**

21509 realloc — memory reallocator

21510 **SYNOPSIS**

21511 #include <stdlib.h>

21512 void *realloc(void *ptr, size_t size);

21513 **DESCRIPTION**

21514 The *realloc()* function changes the size of the memory object pointed to by *ptr* to the size
 21515 specified by *size*. The contents of the object will remain unchanged up to the lesser of the new
 21516 and old sizes. If the new size of the memory object would require movement of the object, the
 21517 space for the previous instantiation of the object is freed. If the new size is larger, the contents of
 21518 the newly allocated portion of the object are unspecified. If *size* is 0 and *ptr* is not a null pointer,
 21519 the object pointed to is freed. If the space cannot be allocated, the object remains unchanged.

21520 If *ptr* is a null pointer, *realloc()* behaves like *malloc()* for the specified size.

21521 If *ptr* does not match a pointer returned earlier by *calloc()*, *malloc()* or *realloc()* or if the space has
 21522 previously been deallocated by a call to *free()* or *realloc()*, the behaviour is undefined.

21523 The order and contiguity of storage allocated by successive calls to *realloc()* is unspecified. The
 21524 pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a
 21525 pointer to any type of object and then used to access such an object in the space allocated (until
 21526 the space is explicitly freed or reallocated). Each such allocation will yield a pointer to an object
 21527 disjoint from any other object. The pointer returned points to the start (lowest byte address) of
 21528 the allocated space. If the space cannot be allocated, a null pointer is returned.

21529 **RETURN VALUE**

21530 Upon successful completion with a size not equal to 0, *realloc()* returns a pointer to the (possibly
 21531 moved) allocated space. If *size* is 0, either a null pointer or a unique pointer that can be
 21532 successfully passed to *free()* is returned. If there is not enough available memory, *realloc()*
 21533 EX returns a null pointer and sets *errno* to [ENOMEM].

21534 **ERRORS**21535 The *realloc()* function will fail if:

21536 EX [ENOMEM] Insufficient memory is available.

21537 **EXAMPLES**

21538 None.

21539 **APPLICATION USAGE**

21540 None.

21541 **FUTURE DIRECTIONS**

21542 None.

21543 **SEE ALSO**21544 *calloc()*, *free()*, *malloc()*, <stdlib.h>.21545 **CHANGE HISTORY**

21546 First released in Issue 1.

21547 Derived from Issue 1 of the SVID.

21548 **Issue 4**

21549 The following changes are incorporated for alignment with the ISO C standard:

21550 • The DESCRIPTION is updated to indicate (a) that the order and contiguity of storage
21551 allocated by successive calls to this function is unspecified, (b) that each allocation yields a
21552 pointer to an object disjoint from any other object, and (c) that the returned pointer points to
21553 the lowest byte address of the allocation.

21554 • The RETURN VALUE section is updated to indicate what will be returned if *size* is 0.

21555 Other changes are incorporated as follows:

21556 • The setting of *errno* and the [ENOMEM] error are marked as extensions.

21557 • The APPLICATION USAGE section is removed.

21558 **NAME**

21559 realpath — resolve a pathname

21560 **SYNOPSIS**

21561 EX #include <stdlib.h>

21562 char *realpath(const char *file_name, char *resolved_name);

21563

21564 **DESCRIPTION**

21565 The *realpath()* function derives, from the pathname pointed to by *file_name*, an absolute
 21566 pathname that names the same file, whose resolution does not involve ".", "..", or symbolic links.
 21567 The generated pathname is stored, up to a maximum of {PATH_MAX} bytes, in the buffer
 21568 pointed to by *resolved_name*.

21569 **RETURN VALUE**

21570 On successful completion, *realpath()* returns a pointer to the resolved name. Otherwise,
 21571 *realpath()* returns a null pointer and sets *errno* to indicate the error, and the contents of the buffer
 21572 pointed to by *resolved_name* are undefined.

21573 **ERRORS**21574 The *realpath()* function will fail if:

- | | | |
|-------|----------------|--|
| 21575 | [EACCES] | Read or search permission was denied for a component of <i>file_name</i> . |
| 21576 | [EINVAL] | Either the <i>file_name</i> or <i>resolved_name</i> argument is a null pointer. |
| 21577 | [EIO] | An error occurred while reading from the file system. |
| 21578 | [ELOOP] | Too many symbolic links were encountered in resolving <i>path</i> . |
| 21579 | [ENAMETOOLONG] | |
| 21580 | | The <i>file_name</i> argument is longer than {PATH_MAX} or a pathname |
| 21581 | | component is longer than {NAME_MAX}. |
| 21582 | [ENOENT] | A component of <i>file_name</i> does not name an existing file or <i>file_name</i> points to |
| 21583 | | an empty string. |
| 21584 | [ENOTDIR] | A component of the path prefix is not a directory. |
| 21585 | | The <i>realpath()</i> function may fail if: |
| 21586 | [ENAMETOOLONG] | |
| 21587 | | Pathname resolution of a symbolic link produced an intermediate result |
| 21588 | | whose length exceeds {PATH_MAX}. |
| 21589 | [ENOMEM] | Insufficient storage space is available. |

21590 **EXAMPLES**

21591 None.

21592 **APPLICATION USAGE**

21593 None.

21594 **FUTURE DIRECTIONS**

21595 None.

21596 **SEE ALSO**21597 *getcwd()*, *sysconf()*, <stdlib.h>.21598 **CHANGE HISTORY**

21599 First released in Issue 4, Version 2.

21600 **Issue 5**

21601 Moved from X/OPEN UNIX extension to BASE.

|

21602 **NAME**21603 re_comp, re_exec — compile and execute regular expressions (**LEGACY**)21604 **SYNOPSIS**21605 EX `#include <re_comp.h>`21606 `char *re_comp(const char *string);`21607 `int re_exec(const char *string);`

21608

21609 **DESCRIPTION**

21610 The *re_comp()* function converts a regular expression string (RE) into an internal form suitable
 21611 for pattern matching. The *re_exec()* function compares the string pointed to by the *string*
 21612 argument with the last regular expression passed to *re_comp()*.

21613 If *re_comp()* is called with a null pointer argument, the current regular expression remains
 21614 unchanged.

21615 Strings passed to both *re_comp()* and *re_exec()* must be terminated by a null byte, and may
 21616 include newline characters.

21617 The *re_comp()* and *re_exec()* functions support *simple regular expressions*, which are defined
 21618 below.

21619 The following one-character REs match a single character:

21620 1.1 An ordinary character (not one of those discussed in 1.2 below) is a one-character RE that
 21621 matches itself.

21622 1.2 A backslash (\) followed by any special character is a one-character RE that matches the
 21623 special character itself. The special characters are:

21624 a. ., *, [, and \ (period, asterisk, left square bracket, and backslash, respectively), which
 21625 are always special, except when they appear within square brackets ([]); see 1.4
 21626 below).

21627 b. ^ (caret or circumflex), which is special at the beginning of an entire RE (see 3.1 and 3.2
 21628 below), or when it immediately follows the left of a pair of square brackets ([]) (see
 21629 1.4 below).

21630 c. \$ (dollar symbol), which is special at the end of an entire RE (see 3.2 below).

21631 d. The character used to bound (delimit) an entire RE, which is special for that RE.

21632 1.3 A period (.) is a one-character RE that matches any character except new-line.

21633 1.4 A non-empty string of characters enclosed in square brackets ([]) is a one-character RE that
 21634 matches any one character in that string. If, however, the first character of the string is a
 21635 circumflex (^), the one-character RE matches any character except new-line and the
 21636 remaining characters in the string. The ^ has this special meaning only if it occurs first in the
 21637 string. The minus (-) may be used to indicate a range of consecutive ASCII characters; for
 21638 example, [0-9] is equivalent to [0123456789]. The - loses this special meaning if it occurs
 21639 first (after an initial ^, if any) or last in the string. The right square bracket (]) does not
 21640 terminate such a string when it is the first character within it (after an initial ^, if any); for
 21641 example, [ja-f] matches either a right square bracket (]) or one of the letters **a** through **f**
 21642 inclusive. The four characters listed in 1.2.a above stand for themselves within such a string
 21643 of characters.

- 21644 The following rules may be used to construct REs from one-character REs:
- 21645 2.1 A one-character RE is a RE that matches whatever the one-character RE matches.
- 21646 2.2 A one-character RE followed by an asterisk (*) is a RE that matches zero or more
21647 occurrences of the one-character RE. If there is any choice, the longest leftmost string that
21648 permits a match is chosen.
- 21649 2.3 A one-character RE followed by `\{m\}`, `\{m,\}`, or `\{m,n\}` is a RE that matches a range of
21650 occurrences of the one-character RE. The values of *m* and *n* must be non-negative integers
21651 less than 256; `\{m\}` matches exactly *m* occurrences; `\{m,\}` matches at least *m* occurrences;
21652 `\{m,n\}` matches any number of occurrences between *m* and *n* inclusive. Whenever a
21653 choice exists, the RE matches as many occurrences as possible.
- 21654 2.4 The concatenation of REs is a RE that matches the concatenation of the strings matched by
21655 each component of the RE.
- 21656 2.5 A RE enclosed between the character sequences `\(` and `\)` is a RE that matches whatever the
21657 unadorned RE matches.
- 21658 2.6 The expression `\n` matches the same string of characters as was matched by an expression
21659 enclosed between `\(` and `\)` earlier in the same RE. Here *n* is a digit; the sub-expression
21660 specified is that beginning with the *n*-th occurrence of `\(` counting from the left. For
21661 example, the expression `^\(.*\)\1$` matches a line consisting of two repeated appearances of
21662 the same string.
- 21663 Finally, an entire RE may be constrained to match only an initial segment or final segment of a
21664 line (or both).
- 21665 3.1 A circumflex (^) at the beginning of an entire RE constrains that RE to match an initial
21666 segment of a line.
- 21667 3.2 A dollar symbol (\$) at the end of an entire RE constrains that RE to match a final segment of
21668 a line. The construction `^entire RE$` constrains the entire RE to match the entire line.
- 21669 The null RE (that is, `//`) is equivalent to the last RE encountered.
- 21670 The behaviour of `re_comp()` and `re_exec()` in locales other than the POSIX locale is unspecified.
- 21671 These interfaces need not be reentrant.
- 21672 **RETURN VALUE**
- 21673 The `re_comp()` function returns a null pointer when the string pointed to by the *string* argument
21674 is successfully converted. Otherwise, a pointer to an unspecified error message string is
21675 returned.
- 21676 Upon successful completion, `re_exec()` returns 1 if *string* matches the last compiled regular
21677 expression. Otherwise, `re_exec()` returns 0 if *string* fails to match the last compiled regular
21678 expression, and -1 if the compiled regular expression is invalid (indicating an internal error).
- 21679 **ERRORS**
- 21680 No errors are defined.
- 21681 **EXAMPLES**
- 21682 None.
- 21683 **APPLICATION USAGE**
- 21684 For portability to implementations conforming to earlier versions of this specification, `regcomp()`
21685 and `regexec()` are preferred to these functions.

21686 **FUTURE DIRECTIONS**

21687 None.

21688 **SEE ALSO**21689 *regcomp()*, <re_comp.h>.21690 **CHANGE HISTORY**

21691 First released in Issue 4, Version 2.

21692 **Issue 5**

21693 Marked LEGACY.

21694 A note indicating that these interfaces need not be reentrant is added to the DESCRIPTION.

21695 NAME

21696 regcmp, regex — compile and execute a regular expression (**LEGACY**)

21697 SYNOPSIS

21698 EX

```
#include <libgen.h>
```

21699

```
char *regcmp (const char *string1 , ... /*, (char *)0 */);
```

21700

```
char *regex (const char *re, const char *subject , ... );
```

21701

```
extern char *__loc1;
```

21702

21703 DESCRIPTION

21704 The *regcmp()* function compiles a regular expression consisting of the concatenated arguments
 21705 and returns a pointer to the compiled form. The end of arguments is indicated by a null pointer.
 21706 The *malloc()* function is used to create space for the compiled form. It is the process'
 21707 responsibility to free unneeded space so allocated. A null pointer returned from *regcmp()*
 21708 indicates an invalid argument.

21709 The *regex()* function executes a compiled pattern against the *subject* string. Additional
 21710 arguments of type **char** * must be passed to receive matched subexpressions back. If an
 21711 insufficient number of arguments is passed to accept all the values that the regular expression
 21712 returns, the behaviour is undefined. A global character pointer *__loc1* points to the first
 21713 matched character in the *subject* string. Both *regcmp()* and *regex()* were largely borrowed from
 21714 the editor, and are defined in *re_comp()*, but the syntax and semantics have been changed
 21715 slightly. The following are the valid symbols and their associated meanings:

21716 **[]*.^** These symbols retain their meaning as defined in *re_comp()*.21717 **\$** Matches the end of the string; **\n** matches a new-line.

21718 **-** Used within brackets, the hyphen signifies an ASCII character range. For example,
 21719 **[a-z]** is equivalent to **[abcd ... xyz]** . The **-** can represent itself only if used as the
 21720 first or last character. For example, the character class expression **[|-]** matches the
 21721 characters **|** and **-**.

21722 **+** A regular expression followed by **+** means one or more times. For example, **[0-9]+**
 21723 is equivalent to **[0-9][0-9]*** .

21724 **{m} {m,} {m,u}**

21725 Integer values enclosed in **{ }** indicate the number of times the preceding regular
 21726 expression can be applied. The value *m* is the minimum number and *u* is a
 21727 number, less than 256, which is the maximum. If the value of either *m* or *u* is 256
 21728 or greater, the behaviour is undefined. The syntax **{m}** indicates the exact number
 21729 of times the regular expression can be applied. The syntax **{m,}** is analogous to
 21730 **{m,infinity}**. The plus (+) and asterisk (*) operations are equivalent to **{1,}** and **{0,}**
 21731 respectively.

21732 **(...)\$n** The value of the enclosed regular expression is returned. The value is stored in the
 21733 (*n*+1)th argument following the *subject* argument. A maximum of ten enclosed
 21734 regular expressions are allowed. The *regex()* function makes its assignments
 21735 unconditionally.

21736 **(...)** Parentheses are used for grouping. An operator, such as *****, **+**, or **{ }**, can work on a
 21737 single character or a regular expression enclosed in parentheses. For example,
 21738 **(a*(cb+)*)\$0** .

21739 Since all of the above defined symbols are special characters, they must be escaped to be used as
 21740 themselves.

- 21741 The behaviour of *regcmp()* and *regex()* in locales other than the POSIX locale is unspecified. |
- 21742 These interfaces need not be reentrant.
- 21743 **RETURN VALUE**
- 21744 Upon successful completion, *regcmp()* returns a pointer to the compiled regular expression.
- 21745 Otherwise, a null pointer is returned and *errno* may be set to indicate the error.
- 21746 Upon successful completion, *regex()* returns a pointer to the next unmatched character in the
- 21747 subject string. Otherwise, a null pointer is returned.
- 21748 The *regex()* function returns a null pointer on failure, or a pointer to the next unmatched
- 21749 character on success.
- 21750 **ERRORS**
- 21751 The *regcmp()* function may fail if:
- 21752 [ENOMEM] Insufficient storage space was available.
- 21753 No errors are defined for *regex()*. |
- 21754 **EXAMPLES** |
- 21755 None.
- 21756 **APPLICATION USAGE**
- 21757 For portability to implementations conforming to earlier versions of this specification, *regcomp()* |
- 21758 is preferred over this function.
- 21759 User programs that use *regcmp()* may run out of memory if *regcmp()* is called iteratively without
- 21760 freeing compiled regular expression strings that are no longer required. |
- 21761 **FUTURE DIRECTIONS** |
- 21762 None.
- 21763 **SEE ALSO**
- 21764 *malloc()*, *regcomp()*, <libgen.h>.
- 21765 **CHANGE HISTORY**
- 21766 First released in Issue 4, Version 2. |
- 21767 **Issue 5** |
- 21768 Marked LEGACY.
- 21769 A note indicating that these interfaces need not be reentrant is added to the DESCRIPTION.

21770 NAME

21771 regcomp, regexec, regerror, regfree — regular expression matching

21772 SYNOPSIS

21773 OH `#include <sys/types.h>`

21774 `#include <regex.h>`

21775 `int regcomp(regex_t *preg, const char *pattern, int cflags);`

21776 `int regexec(const regex_t *preg, const char *string,`

21777 `size_t nmatch, regmatch_t pmatch[], int eflags);`

21778 `size_t regerror(int errcode, const regex_t *preg,`

21779 `char *errbuf, size_t errbuf_size);`

21780 `void regfree(regex_t *preg);`

21781 DESCRIPTION

21782 These functions interpret *basic* and *extended* regular expressions as described in the XBD
21783 specification, **Chapter 7, Regular Expressions**.

21784 The structure type **regex_t** contains at least the following member:

21785

21786

21787

Member Type	Member Name	Description
size_t	re_nsub	Number of parenthesised subexpressions.

21788 The structure type **regmatch_t** contains at least the following members:

21789

21790

21791

21792

21793

Member Type	Member Name	Description
regoff_t	rm_so	Byte offset from start of <i>string</i> to start of substring.
regoff_t	rm_eo	Byte offset from start of <i>string</i> of the first character after the end of substring.

21794 The *regcomp()* function will compile the regular expression contained in the string pointed to by
21795 the *pattern* argument and place the results in the structure pointed to by *preg*. The *cflags*
21796 argument is the bitwise inclusive OR of zero or more of the following flags, which are defined in
21797 the header **<regex.h>**:

21798 **REG_EXTENDED** Use Extended Regular Expressions.

21799 **REG_ICASE** Ignore case in match. (See the XBD specification, **Chapter 7, Regular**
21800 **Expressions**.)

21801 **REG_NOSUB** Report only success/fail in *regexec()*.

21802 **REG_NEWLINE** Change the handling of newline characters, as described in the text.

21803 The default regular expression type for *pattern* is a Basic Regular Expression. The application
21804 can specify Extended Regular Expressions using the **REG_EXTENDED** *cflags* flag.

21805 On successful completion, it returns 0; otherwise it returns non-zero, and the content of *preg* is
21806 undefined.

21807 If the **REG_NOSUB** flag was not set in *cflags*, then *regcomp()* will set *re_nsub* to the number of
21808 parenthesised subexpressions (delimited by `\(\)` in basic regular expressions or `()` in extended
21809 regular expressions) found in *pattern*.

21810 The *regexec()* function compares the null-terminated string specified by *string* with the compiled
21811 regular expression *preg* initialised by a previous call to *regcomp()*. If it finds a match, *regexec()*
21812 returns 0; otherwise it returns non-zero indicating either no match or an error. The *eflags*

21813 argument is the bitwise inclusive OR of zero or more of the following flags, which are defined in
21814 the header `<regex.h>`:

21815 REG_NOTBOL The first character of the string pointed to by *string* is not the beginning of the
21816 line. Therefore, the circumflex character (^), when taken as a special
21817 character, will not match the beginning of *string*.

21818 REG_NOTEOL The last character of the string pointed to by *string* is not the end of the line.
21819 Therefore, the dollar sign (\$), when taken as a special character, will not
21820 match the end of *string*.

21821 If *nmatch* is 0 or REG_NOSUB was set in the *cflags* argument to *regcomp()*, then *regexexec()* will
21822 ignore the *pmatch* argument. Otherwise, the *pmatch* argument must point to an array with at
21823 least *nmatch* elements, and *regexexec()* will fill in the elements of that array with offsets of the
21824 substrings of *string* that correspond to the parenthesised subexpressions of *pattern*:
21825 *pmatch[i].rm_so* will be the byte offset of the beginning and *pmatch[i].rm_eo* will be one greater
21826 than the byte offset of the end of substring *i*. (Subexpression *i* begins at the *i*th matched open
21827 parenthesis, counting from 1.) Offsets in *pmatch[0]* identify the substring that corresponds to the
21828 entire regular expression. Unused elements of *pmatch* up to *pmatch[nmatch-1]* will be filled with
21829 -1. If there are more than *nmatch* subexpressions in *pattern* (*pattern* itself counts as a
21830 subexpression), then *regexexec()* will still do the match, but will record only the first *nmatch*
21831 substrings.

21832 When matching a basic or extended regular expression, any given parenthesised subexpression
21833 of *pattern* might participate in the match of several different substrings of *string*, or it might not
21834 match any substring even though the pattern as a whole did match. The following rules are
21835 used to determine which substrings to report in *pmatch* when matching regular expressions:

- 21836 1. If subexpression *i* in a regular expression is not contained within another subexpression,
21837 and it participated in the match several times, then the byte offsets in *pmatch[i]* will delimit
21838 the last such match.
- 21839 2. If subexpression *i* is not contained within another subexpression, and it did not participate
21840 in an otherwise successful match, the byte offsets in *pmatch[i]* will be -1. A subexpression
21841 does not participate in the match when:

21842 * or \{ \} appears immediately after the subexpression in a basic regular expression, or
21843 *, ?, or { } appears immediately after the subexpression in an extended regular
21844 expression, and the subexpression did not match (matched 0 times)

21845 or:

21846 | is used in an extended regular expression to select this subexpression or another, and
21847 the other subexpression matched.

- 21848 3. If subexpression *i* is contained within another subexpression *j*, and *i* is not contained
21849 within any other subexpression that is contained within *j*, and a match of subexpression *j*
21850 is reported in *pmatch[j]*, then the match or non-match of subexpression *i* reported in
21851 *pmatch[i]* will be as described in 1. and 2. above, but within the substring reported in
21852 *pmatch[j]* rather than the whole string.

- 21853 4. If subexpression *i* is contained in subexpression *j*, and the byte offsets in *pmatch[j]* are -1,
21854 then the pointers in *pmatch[i]* also will be -1.

- 21855 5. If subexpression *i* matched a zero-length string, then both byte offsets in *pmatch[i]* will be
21856 the byte offset of the character or null terminator immediately following the zero-length
21857 string.

21858 If, when *regexexec()* is called, the locale is different from when the regular expression was
 21859 compiled, the result is undefined.

21860 If REG_NEWLINE is not set in *cflags*, then a newline character in *pattern* or *string* will be treated
 21861 as an ordinary character. If REG_NEWLINE is set, then newline will be treated as an ordinary
 21862 character except as follows:

- 21863 1. A newline character in *string* will not be matched by a period outside a bracket expression
 21864 or by any form of a non-matching list (see the **XBD** specification, **Chapter 7, Regular**
 21865 **Expressions**).
- 21866 2. A circumflex (^) in *pattern*, when used to specify expression anchoring (see the **XBD**
 21867 specification, **Section 7.3.8, BRE Expression Anchoring**), will match the zero-length string
 21868 immediately after a newline in *string*, regardless of the setting of REG_NOTBOL.
- 21869 3. A dollar-sign (\$) in *pattern*, when used to specify expression anchoring, will match the
 21870 zero-length string immediately before a newline in *string*, regardless of the setting of
 21871 REG_NOTEOL.

21872 The *regfree()* function frees any memory allocated by *regcomp()* associated with *preg*.

21873 The following constants are defined as error return values:

21874	REG_NOMATCH	<i>regexexec()</i> failed to match.
21875	REG_BADPAT	Invalid regular expression.
21876	REG_ECOLLATE	Invalid collating element referenced.
21877	REG_ECTYPE	Invalid character class type referenced.
21878	REG_EESCAPE	Trailing \ in pattern.
21879	REG_ESUBREG	Number in \digit invalid or in error.
21880	REG_EBRACK	[] imbalance.
21881	REG_ENOSYS	The function is not supported.
21882	REG_EPAREN	\(\) or () imbalance.
21883	REG_EBRACE	\{ \} imbalance.
21884	REG_BADBR	Content of \{ \} invalid: not a number, number too large, more than two 21885 numbers, first larger than second.
21886	REG_ERANGE	Invalid endpoint in range expression.
21887	REG_ESPACE	Out of memory.
21888	REG_BADRPT	?, * or + not preceded by valid regular expression.

21889 The *regerror()* function provides a mapping from error codes returned by *regcomp()* and
 21890 *regexexec()* to unspecified printable strings. It generates a string corresponding to the value of the
 21891 *errcode* argument, which must be the last non-zero value returned by *regcomp()* or *regexexec()* with
 21892 the given value of *preg*. If *errcode* is not such a value, the content of the generated string is
 21893 unspecified.

21894 If *preg* is a null pointer, but *errcode* is a value returned by a previous call to *regexexec()* or *regcomp()*,
 21895 the *regerror()* still generates an error string corresponding to the value of *errcode*, but it might not
 21896 be as detailed under some implementations.

21897 If the *errbuf_size* argument is not 0, *regerror()* will place the generated string into the buffer of
 21898 size *errbuf_size* bytes pointed to by *errbuf*. If the string (including the terminating null) cannot fit

21899 in the buffer, *regerror()* will truncate the string and null-terminate the result.

21900 If *errbuf_size* is 0, *regerror()* ignores the *errbuf* argument, and returns the size of the buffer needed
 21901 to hold the generated string.

21902 If the *preg* argument to *regexexec()* or *regfree()* is not a compiled regular expression returned by
 21903 *regcomp()*, the result is undefined. A *preg* is no longer treated as a compiled regular expression
 21904 after it is given to *regfree()*.

21905 **RETURN VALUE**

21906 On successful completion, the *regcomp()* function returns 0. Otherwise, it returns an integer
 21907 value indicating an error as described in <regex.h>, and the content of *preg* is undefined.

21908 On successful completion, the *regexexec()* function returns 0. Otherwise it returns
 21909 REG_NOMATCH to indicate no match, or REG_ENOSYS to indicate that the function is not
 21910 supported.

21911 Upon successful completion, the *regerror()* function returns the number of bytes needed to hold
 21912 the entire generated string. Otherwise, it returns 0 to indicate that the function is not
 21913 implemented.

21914 The *regfree()* function returns no value.

21915 **ERRORS**

21916 No errors are defined.

21917 **EXAMPLES**

21918 #include <regex.h>

21919 /*
 21920 * Match string against the extended regular expression in
 21921 * pattern, treating errors as no match.
 21922 *
 21923 * return 1 for match, 0 for no match
 21924 */

21925 int
 21926 match(const char *string, char *pattern)
 21927 {
 21928 int status;
 21929 regex_t re;

21930 if (regcomp(&re, pattern, REG_EXTENDED | REG_NOSUB) != 0) {
 21931 return(0); /* report error */
 21932 }
 21933 status = regexexec(&re, string, (size_t) 0, NULL, 0);
 21934 regfree(&re);
 21935 if (status != 0) {
 21936 return(0); /* report error */
 21937 }
 21938 return(1);
 21939 }

21940 The following demonstrates how the REG_NOTBOL flag could be used with *regexexec()* to find all
 21941 substrings in a line that match a pattern supplied by a user. (For simplicity of the example, very
 21942 little error checking is done.)

```

21943      (void) regcomp (&re, pattern, 0);
21944      /* this call to regexec() finds the first match on the line */
21945      error = regexec (&re, &buffer[0], 1, &pm, 0);
21946      while (error == 0) {      /* while matches found */
21947          /* substring found between pm.rm_so and pm.rm_eo */
21948          /* This call to regexec() finds the next match */
21949          error = regexec (&re, buffer + pm.rm_eo, 1, &pm, REG_NOTBOL);
21950      }

```

21951 APPLICATION USAGE

21952 An application could use:

```

21953      regerror(code, preg, (char *)NULL, (size_t)0)

```

21954 to find out how big a buffer is needed for the generated string, *malloc()* a buffer to hold the
 21955 string, and then call *regerror()* again to get the string. Alternatively, it could allocate a fixed,
 21956 static buffer that is big enough to hold most strings, and then use *malloc()* to allocate a larger
 21957 buffer if it finds that this is too small.

21958 To match a pattern as described in the XCU specification, **Section 2.13, Pattern Matching**
 21959 **Notation** use the *fnmatch()* function.

21960 FUTURE DIRECTIONS

21961 None.

21962 SEE ALSO

21963 *fnmatch()*, *glob()*, *<regex.h>*, *<sys/types.h>*.

21964 CHANGE HISTORY

21965 First released in Issue 4.

21966 Derived from the ISO POSIX-2 standard.

21967 Issue 5

21968 Moved from POSIX2 C-language Binding to BASE.

21969 **NAME**21970 regex — execute a regular expression (**LEGACY**)21971 **SYNOPSIS**21972 EX `#include <libgen.h>`21973 `char *regex (const char *re, const char *subject , ...);`

21974

21975 **DESCRIPTION**21976 Refer to *regcmp()*.21977 **CHANGE HISTORY**

21978 First released in Issue 4, Version 2.

21979 **Issue 5**

21980 Marked LEGACY.

21981 **NAME**

21982 advance, compile, step, loc1, loc2, locs — compile and match regular expressions (**LEGACY**)

21983 **SYNOPSIS**

```
21984 EX #define INIT declarations
21985 #define GETC( ) getc code
21986 #define PEEK( ) peek code
21987 #define UNGETC( ) ungetc code
21988 #define RETURN(ptr) return code
21989 #define ERROR(val) error code

21990 #include <regex.h>

21991 char *compile(char *instring, char *expbuf,
21992             const char *endbuf, int eof);

21993 int step(const char *string, const char *expbuf);

21994 int advance(const char *string, const char *expbuf);

21995 extern char *loc1, *loc2, *locs;
21996
```

21997 **DESCRIPTION**

21998 These are general-purpose, regular expression-matching functions to be used in programs that
 21999 perform regular expression matching, using the Regular Expressions described in **Simple**
 22000 **Regular Expressions (Historical Version)** on page 716. These functions are defined by the
 22001 **<regex.h>** header.

22002 Implementations may also accept internationalised simple regular expressions as input.

22003 Programs must have the following five macros declared before the **#include <regex.h>**
 22004 statement. These macros are used by *compile()*. The macros GETC(), PEEKC() and UNGETC()
 22005 operate on the regular expression given as input to *compile()*.

22006 GETC() This macro returns the value of the next character (byte) in the regular
 22007 expression pattern. Successive calls to GETC() should return successive
 22008 characters of the regular expression.

22009 PEEKC() This macro returns the next character (byte) in the regular expression.
 22010 Immediately successive calls to PEEKC() should return the same byte, which
 22011 should also be the next character returned by GETC().

22012 UNGETC(c) This macro causes the argument *c* to be returned by the next call to GETC()
 22013 and PEEKC(). No more than one character of pushback is ever needed and
 22014 this character is guaranteed to be the last character read by GETC(). The
 22015 value of the macro UNGETC(c) is always ignored.

22016 RETURN(ptr) This macro is used on normal exit of the *compile()* function. The value of the
 22017 argument *ptr* is a pointer to the character after the last character of the
 22018 compiled regular expression. This is useful to programs that have memory
 22019 allocation to manage.

22020 ERROR(val) This macro is the abnormal return from *compile()*. The argument *val* is an
 22021 error number (see the **ERRORS** section below for meanings). This call should
 22022 never return.

22023 The *step()* and *advance()* functions do pattern matching given a character string and a compiled
 22024 regular expression as input.

22025 The *compile()* function takes as input a simple regular expression (see **Simple Regular**
 22026 **Expressions (Historical Version)** on page 716) and produces a compiled expression that can be
 22027 used with *step()* and *advance()*.

22028 The first parameter *instring* is never used explicitly by *compile()* but is useful for programs that
 22029 pass down different pointers to input characters. It is sometimes used in the INIT declaration
 22030 (see below). Programs which invoke functions to input characters or have characters in an
 22031 external array can pass down (**char** *) 0 for this parameter.

22032 The next parameter *expbuf* is a character pointer. It points to the place where the compiled
 22033 regular expression will be placed.

22034 The parameter *endbuf* is one more than the highest address where the compiled regular
 22035 expression may be placed. If the compiled expression cannot fit in (*endbuf-expbuf*) bytes, a call
 22036 to ERROR(50) is made.

22037 The parameter *eof* is the character which marks the end of the regular expression.

22038 Each program that includes the <**regexp.h**> header must have a **#define** statement for INIT. It is
 22039 used for dependent declarations and initialisations. Most often it is used to set a register
 22040 variable to point to the beginning of the regular expression so that this register variable can be
 22041 used in the declarations for GETC(), PEEKC() and UNGETC(). Otherwise it can be used to
 22042 declare external variables that might be used by GETC(), PEEKC() and UNGETC(). See the
 22043 EXAMPLES section below.

22044 The first parameter to *step()* is a pointer to a string of characters to be checked for a match. This
 22045 string should be null-terminated.

22046 The second parameter, *expbuf*, is the compiled regular expression which was obtained by a call
 22047 to *compile*.

22048 The *step()* function returns non-zero if some substring of *string* matches the regular expression
 22049 in *expbuf*, and 0, if there is no match. If there is a match, two external character pointers are set
 22050 as a side effect to the call to *step()*. The variable *loc1* points to the first character that matched
 22051 the regular expression; the variable *loc2* points to the character after the last character that
 22052 matches the regular expression. Thus if the regular expression matches the entire input string,
 22053 *loc1* will point to the first character of *string* and *loc2* will point to the null at the end of *string*.

22054 The *advance()* function returns non-zero if the initial substring of *string* matches the regular
 22055 expression in *expbuf*. If there is a match an external character pointer, *loc2*, is set as a side effect.
 22056 The variable *loc2* points to the next character in *string* after the last character that matched.

22057 When *advance()* encounters a "*" or "{\}" sequence in the regular expression, it will advance its
 22058 pointer to the string to be matched as far as possible and will recursively call itself trying to
 22059 match the rest of the string to the rest of the regular expression. As long as there is no match,
 22060 *advance()* will back up along the string until it finds a match or reaches the point in the string
 22061 that initially matched the * or "{\}". It is sometimes desirable to stop this backing up before the
 22062 initial point in the string is reached. If the external character pointer *locs* is equal to the point in
 22063 the string at some time during the backing up process, *advance()* will break out of the loop that
 22064 backs up and will return 0.

22065 The external variables *circf*, *sed* and *nbra* are reserved.

22066 **Simple Regular Expressions (Historical Version)**

22067 A Simple Regular Expression (SRE) specifies a set of character strings. A member of this set of
22068 strings is said to be *matched* by the SRE.

22069 A *pattern* is constructed from one or more SREs. An SRE consists of *ordinary characters* or
22070 *metacharacters*.

22071 Within a pattern, all alphanumeric characters that are not part of a bracket expression, back-
22072 reference or duplication match themselves; that is, the SRE pattern *abc*, when applied to a set of
22073 strings, will match only those strings containing the character sequence *abc* anywhere in them.

22074 Most other characters also match themselves. However, a small set of characters, known as the
22075 *metacharacters*, have special meanings when encountered in patterns. They are described below.

22076 **Simple Regular Expression Construction**

22077 SREs are constructed as follows:

22078 **Expression Meaning**

22079 *c* The character *c*, where *c* is not a special character.

22080 *\c* The character *c*, where *c* is any character with special meaning, see below.

22081 *^* The beginning of the string being compared.

22082 *\$* The end of the string being compared.

22083 *.* Any character.

22084 [*s*] Any character in the non-empty set *s*, where *s* is a sequence of characters. Ranges
22085 may be specified as *c–c*. The character *]* may be included in the set by placing it
22086 first in the set. The character *"–"* may be included in the set by placing it first or
22087 last in the set. The character *"^"* may be included in the set by placing it anywhere
22088 other than first in the set, see below. Ranges in Simple Regular Expressions are
22089 only valid if the *LC_COLLATE* category is set to the C locale. Otherwise, the effect
22090 of using the range notation is unspecified.

22091 [*^s*] Any character not in the set *s*, where *s* is defined as above.

22092 *r*^{*} Zero or more successive occurrences of the regular expression *r*. The longest
22093 leftmost match is chosen.

22094 *rx* The occurrence of regular expression *r* followed by the occurrence of regular
22095 expression *x*. (Concatenation.)

22096 *r*^{*m*,*n*} Any number of *m* through *n* successive occurrences of the regular expression *r*.
22097 The regular expression *r*^{*m*} matches exactly *m* occurrences, *r*^{*m*,\} matches at
22098 least *m* occurrences. The maximum number of occurrences is matched.

22099 *\(r\)* The regular expression *r*. The *\(* and *\)* sequences are ignored.

22100 *\n* When *\n* (where *n* is a number in the range 1 to 9) appears in a concatenated
22101 regular expression, it stands for the regular expression *x*, where *x* is the *n*th regular
22102 expression enclosed in *\(* and *\)* sequences that appeared earlier in the
22103 concatenated regular expression. For example, in the pattern *\(r\)**x*^{\2}(*y* the *\2*
22104 matches the regular expression *y*, giving *rxzy*.

22105 Characters that have special meaning except where they appear within square brackets, `[]`, or
 22106 are preceded by `"\"` are:

22107 `. * [\`

22108 Other special characters, such as `$` have special meaning in more restricted contexts.

22109 The character `"^"` at the beginning of an expression permits a successful match only immediately
 22110 after a newline or at the beginning of each of the strings to which the match is applied, and the
 22111 character `"$"` at the end of an expression requires a trailing newline.

22112 Two characters have special meaning only when used within square brackets. The character `"–"`
 22113 denotes a range, `[c–c]`, unless it is just after the left square bracket or before the right square
 22114 bracket, `[–c]` or `[c–]`, in which case it has no special meaning. The character `"^"` has the meaning
 22115 *complement of* if it immediately follows the left square bracket, `[^c]`. Elsewhere between brackets,
 22116 `[c^]`, it stands for the ordinary character `"^"`. The right square bracket (`]`) loses its special meaning
 22117 and represents itself in a bracket expression if it occurs first in the list after any initial circumflex
 22118 (`^`) character.

22119 The special meaning of the `"\"` operator can be escaped *only* by preceding it with another `"\"`;
 22120 that is, `"\\\"`.

22121 **SRE Operator Precedence**

22122 The precedence of the operators is as shown below:

22123 `[...]` High precedence.

22124 `*` `.`

22125 concatenation Low precedence.

22126 **Internationalised SREs**

22127 Character expressions within square brackets are constructed as follows:

22128 **Expression Meaning**

22129 `c` The single character `c` where `c` is not a special character.

22130 `[[:class:]]` A character class expression. Any character of type *class*, as defined by category
 22131 `LC_CTYPE` in the program's locale (see the **XBD** specification, **Chapter 5, Locale**).
 22132 For *class*, one of the following should be substituted:

22133 `alpha` A letter.

22134 `upper` An upper-case letter.

22135 `lower` A lower-case letter.

22136 `digit` A decimal digit.

22137 `xdigit` A hexadecimal digit.

22138 `alnum` An alphanumeric (letter or digit).

22139 `space` A character producing white space in displayed text.

22140 `punct` A punctuation character.

22141 `print` A printing character.

22142 `graph` A character with a visible representation.

22143 `cntrl` A control character.

22144 `[[=c=]]` An equivalence class. Any collation element defined as having the same relative
 22145 order in the current collation sequence as `c`. As an example, if **A** and **a** belong to
 22146 the same equivalence class, then both `[[=A=]b]` and `[[=a=]b]` are equivalent to `[`
 22147 `Aab]`.

22148 [[.cc.]] A collating symbol. Multi-character collating elements must be represented as
 22149 collating symbols to distinguish them from single-character collating elements. As
 22150 an example, if the string *ch* is a valid collating element, then [[.ch.]] will be treated
 22151 as an element matching the same string of characters, while *ch* will be treated as a
 22152 simple list of *c* and *h*. If the string is not a valid collating element in the current
 22153 collating sequence definition, the symbol will be treated as an invalid expression.

22154 [c-c] Any collation element in the character expression range *c-c*, where *c* can identify a
 22155 collating symbol or an equivalence class. If the character "-" appears immediately
 22156 after an opening square bracket (for example, [-c]) or immediately prior to a
 22157 closing square bracket (for example, [c-]), it has no special meaning.

22158 ^ Immediately following an opening square bracket, means the complement of, for
 22159 example, [^c]. Otherwise, it has no special meaning.

22160 Within square brackets, a "." that is not part of a [[.cc.]] sequence, or a ":" that is not part of a
 22161 [[:class:]] sequence, or an "=" that is not part of a [[=c=]] sequence, matches itself.

22162 SRE Examples

22163 Below are examples of regular expressions:

22164 22165	Pattern	Meaning
22166	ab.d	ab <i>any character</i> d
22167	ab.*d	ab <i>any sequence of characters (including none)</i> d
22168	ab[xyz]d	ab <i>one of x y or z</i> d
22169	ab[^c]d	ab <i>anything except c</i> d
22170	^abcd\$	<i>a line containing only</i> abcd
22171	[a-d]	<i>any one of a b c or d</i>

22172 These interfaces need not be reentrant.

22173 RETURN VALUE

22174 The *compile()* function uses the macro RETURN() on success and the macro ERROR() on failure,
 22175 see above. The *step()* and *advance()* functions return non-zero on a successful match and 0 if
 22176 there is no match.

22177 ERRORS

22178 11 Range endpoint too large.
 22179 16 Bad number.
 22180 25 \digit out of range.
 22181 36 Illegal or missing delimiter.
 22182 41 No remembered search string.
 22183 42 \(\) imbalance.
 22184 43 Too many \(.
 22185 44 More than two numbers given in \{ \} .
 22186 45 } expected after \ .
 22187 46 First number exceeds second in \{ \} .
 22188 49 [] imbalance.
 22189 50 Regular expression overflow.

22190 **EXAMPLES**

22191 The following is an example of how the regular expression macros and calls might be defined by
 22192 an application program:

```

22193     #define INIT          char *sp = instring;
22194     #define GETC( )       (*sp++)
22195     #define PEEKC( )      (*sp)
22196     #define UNGETC(c)     (--sp)
22197     #define RETURN(c)     return;
22198     #define ERROR(c)      regerr( )

22199     #include <regex.h>
22200     . . .
22201     (void) compile(*argv, expbuf, &expbuf[ESIZE], '\0');
22202     . . .
22203     if (step(linebuf, expbuf) )
22204         succeed( );
  
```

22205 **APPLICATION USAGE**

22206 Applications should migrate to the *fnmatch()*, *glob()*, *regcomp()* and *regexexec()* functions which
 22207 provide full internationalised regular expression functionality compatible with the ISO POSIX-2
 22208 standard, as described in the **XBD** specification, **Chapter 7, Regular Expressions**.

22209 **FUTURE DIRECTIONS**

22210 None.

22211 **SEE ALSO**

22212 *fnmatch()*, *glob()*, *regcomp()*, *regexexec()*, *setlocale()*, **<regex.h>**, **<regex.h>**, the **XBD** specification,
 22213 **Chapter 7, Regular Expressions**.

22214 **CHANGE HISTORY**

22215 First released in Issue 2.

22216 Derived from Issue 2 of the SVID.

22217 **Issue 4**

22218 The following changes are incorporated in this issue:

- 22219 • The interface is marked TO BE WITHDRAWN, because improved functionality is now
 22220 provided by interfaces introduced for alignment with the ISO POSIX-2 standard.
- 22221 • The type of the arguments *endbuf*, *string* and *expbuf* is changed from **char *** to **const char ***.
- 22222 • In the DESCRIPTION some of the text is reworded to improve clarity.
- 22223 • The APPLICATION USAGE section is added.
- 22224 • The example is corrected.
- 22225 • The FUTURE DIRECTIONS section is removed.

22226 **Issue 5**

22227 Marked LEGACY.

22228 A note indicating that these interfaces need not be reentrant is added to the DESCRIPTION.

22229 **NAME**

22230 remainder — remainder function

22231 **SYNOPSIS**22232 EX `#include <math.h>`22233 `double remainder(double x, double y);`

22234

22235 **DESCRIPTION**

22236 The *remainder()* function returns the floating point remainder $r = x - ny$ when y is non-zero. The
 22237 value n is the integral value nearest the exact value x/y . When $|n - x/y| = 1/2$, the value n is
 22238 chosen to be even.

22239 The behaviour of *remainder()* is independent of the rounding mode.

22240 **RETURN VALUE**

22241 The *remainder()* function returns the floating point remainder $r = x - ny$ when y is non-zero.

22242 When y is 0, *remainder()* returns (NaN or equivalent if available) and sets *errno* to [EDOM].

22243 If the value of x is $\pm\text{Inf}$, *remainder()* returns NaN and sets *errno* to [EDOM].

22244 If x or y is NaN, then the function returns NaN and *errno* may be set to [EDOM].

22245 **ERRORS**

22246 The *remainder()* function will fail if:

22247 [EDOM] The y argument is 0 or the x argument is positive or negative infinity.

22248 The *remainder()* function may fail if:

22249 [EDOM] The x or y argument is NaN.

22250 **EXAMPLES**

22251 None.

22252 **APPLICATION USAGE**

22253 None.

22254 **FUTURE DIRECTIONS**

22255 None.

22256 **SEE ALSO**

22257 *abs()*, <math.h>.

22258 **CHANGE HISTORY**

22259 First released in Issue 4, Version 2.

22260 **Issue 5**

22261 Moved from X/OPEN UNIX extension to BASE.

22262 **NAME**

22263 remove — remove files

22264 **SYNOPSIS**

22265 #include <stdio.h>

22266 int remove(const char *path);

22267 **DESCRIPTION**

22268 The *remove()* function causes the file named by the pathname pointed to by *path* to be no longer
 22269 accessible by that name. A subsequent attempt to open that file using that name will fail, unless
 22270 it is created anew.

22271 EX If *path* does not name a directory, *remove(path)* is equivalent to *unlink(path)*.

22272 If *path* names a directory, *remove(path)* is equivalent to *rmdir(path)*.

22273 **RETURN VALUE**22274 EX Refer to *rmdir()* or *unlink()*.22275 **ERRORS**22276 EX Refer to *rmdir()* or *unlink()*.22277 **EXAMPLES**

22278 None.

22279 **APPLICATION USAGE**

22280 None.

22281 **FUTURE DIRECTIONS**

22282 None.

22283 **SEE ALSO**22284 *rmdir()*, *unlink()*, <stdio.h>.22285 **CHANGE HISTORY**

22286 First released in Issue 3.

22287 Entry included for alignment with the POSIX.1-1988 standard and the ISO C standard.

22288 **Issue 4**

22289 The following changes are incorporated for alignment with the ISO C standard:

- 22290 • The type of argument *path* is changed from **char *** to **const char ***.
- 22291 • The DESCRIPTION is expanded to describe the operation of *remove()* more completely.

22292 Another change is incorporated as follows:

- 22293 • All statements containing references to *unlink()* and *rmdir()* in the DESCRIPTION, RETURN
 22294 VALUE and ERRORS sections are marked as extensions.

22295 **NAME**

22296 remque — remove an element from a queue

22297 **SYNOPSIS**

22298 EX #include <search.h>

22299 void remque(void *element);

22300

22301 **DESCRIPTION**22302 Refer to *insque()*.22303 **CHANGE HISTORY**

22304 First released in Issue 4, Version 2.

22305 **Issue 5**

22306 Moved from X/OPEN UNIX extension to BASE.

22307 **NAME**

22308 rename — rename a file

22309 **SYNOPSIS**

22310 #include <stdio.h>

22311 int rename(const char *old, const char *new);

22312 **DESCRIPTION**

22313 The *rename()* function changes the name of a file. The *old* argument points to the pathname of
 22314 the file to be renamed. The *new* argument points to the new pathname of the file.

22315 If the *old* argument and the *new* argument both refer to, and both link to the same existing file,
 22316 *rename()* returns successfully and performs no other action.

22317 If the *old* argument points to the pathname of a file that is not a directory, the *new* argument
 22318 must not point to the pathname of a directory. If the link named by the *new* argument exists, it is
 22319 removed and *old* renamed to *new*. In this case, a link named *new* will remain visible to other
 22320 processes throughout the renaming operation and will refer either to the file referred to by *new*
 22321 or *old* before the operation began. Write access permission is required for both the directory
 22322 containing *old* and the directory containing *new*.

22323 If the *old* argument points to the pathname of a directory, the *new* argument must not point to
 22324 the pathname of a file that is not a directory. If the directory named by the *new* argument exists,
 22325 it will be removed and *old* renamed to *new*. In this case, a link named *new* will exist throughout
 22326 the renaming operation and will refer either to the file referred to by *new* or *old* before the
 22327 operation began. Thus, if *new* names an existing directory, it must be an empty directory.

22328 EX If *old* points to a pathname that names a symbolic link, the symbolic link is renamed. If *new*
 22329 points to a pathname that names a symbolic link, the symbolic link is removed.

22330 The *new* pathname must not contain a path prefix that names *old*. Write access permission is
 22331 required for the directory containing *old* and the directory containing *new*. If the *old* argument
 22332 points to the pathname of a directory, write access permission may be required for the directory
 22333 named by *old*, and, if it exists, the directory named by *new*.

22334 If the link named by the *new* argument exists and the file's link count becomes 0 when it is
 22335 removed and no process has the file open, the space occupied by the file will be freed and the file
 22336 will no longer be accessible. If one or more processes have the file open when the last link is
 22337 removed, the link will be removed before *rename()* returns, but the removal of the file contents
 22338 will be postponed until all references to the file are closed.

22339 Upon successful completion, *rename()* will mark for update the *st_ctime* and *st_mtime* fields of
 22340 the parent directory of each file.

22341 **RETURN VALUE**

22342 Upon successful completion, *rename()* returns 0. Otherwise, -1 is returned, *errno* is set to
 22343 indicate the error, and neither the file named by *old* nor the file named by *new* will be changed or
 22344 created.

22345 **ERRORS**

22346 The *rename()* function will fail if:

22347 [EACCES] A component of either path prefix denies search permission; or one of the
 22348 directories containing *old* or *new* denies write permissions; or, write
 22349 permission is required and is denied for a directory pointed to by the *old* or
 22350 *new* arguments.

22351 [EBUSY] The directory named by *old* or *new* is currently in use by the system or another
 22352 process, and the implementation considers this an error.

22353 [EEXIST] or [ENOTEMPTY]

22354 The link named by *new* is a directory that is not an empty directory.

22355 [EINVAL] The *new* directory pathname contains a path prefix that names the *old*
 22356 directory.

22357 EX [EIO] A physical I/O error has occurred.

22358 [EISDIR] The *new* argument points to a directory and the *old* argument points to a file
 22359 that is not a directory.

22360 EX [ELOOP] Too many symbolic links were encountered in resolving either pathname.

22361 [EMLINK] The file named by *old* is a directory, and the link count of the parent directory
 22362 of *new* would exceed {LINK_MAX}.

22363 [ENAMETOOLONG]

22364 FIPS The length of the *old* or *new* argument exceeds {PATH_MAX} or a pathname
 22365 component is longer than {NAME_MAX}.

22366 [ENOENT] The link named by *old* does not name an existing file, or either *old* or *new*
 22367 points to an empty string.

22368 [ENOSPC] The directory that would contain *new* cannot be extended.

22369 [ENOTDIR] A component of either path prefix is not a directory; or the *old* argument
 22370 names a directory and *new* argument names a non-directory file.

22371 EX [EPERM] or [EACCES]

22372 The S_ISVTX flag is set on the directory containing the file referred to by *old*
 22373 and the caller is not the file owner, nor is the caller the directory owner, nor
 22374 does the caller have appropriate privileges; or *new* refers to an existing file, the
 22375 S_ISVTX flag is set on the directory containing this file and the caller is not the
 22376 file owner, nor is the caller the directory owner, nor does the caller have
 22377 appropriate privileges.

22378 [EROFS] The requested operation requires writing in a directory on a read-only file
 22379 system.

22380 [EXDEV] The links named by *new* and *old* are on different file systems and the
 22381 implementation does not support links between file systems.

22382 The *rename()* function may fail if:

22383 EX [EBUSY] The file named by the *old* or *new* arguments is a named STREAM.

22384 EX [ENAMETOOLONG]

22385 Pathname resolution of a symbolic link produced an intermediate result
 22386 whose length exceeds {PATH_MAX}.

22387 [ETXTBSY] The file to be renamed is a pure procedure (shared text) file that is being
 22388 executed.

22389 EXAMPLES

22390 None.

22391 APPLICATION USAGE

22392 None.

22393 FUTURE DIRECTIONS

22394 None.

22395 SEE ALSO

22396 *link()*, *rmdir()*, *symlink()*, *unlink()*, *<stdio.h>*.

22397 CHANGE HISTORY

22398 First released in Issue 3.

22399 Entry included for alignment with the POSIX.1-1988 standard.

22400 Issue 4

22401 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- 22402 • The type of arguments *old* and *new* are changed from **char *** to **const char ***.
- 22403 • The RETURN VALUE section now states that if an error occurs, neither file will be changed
 22404 or created.

22405 The following change is incorporated for alignment with the FIPS requirements:

- 22406 • In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a
 22407 pathname component is larger than {NAME_MAX}, is now defined as mandatory and
 22408 marked as an extension.

22409 Another change is incorporated as follows:

- 22410 • The [EMLINK] error is added to the ERRORS section.

22411 Issue 4, Version 2

22412 The following changes are made for X/OPEN UNIX conformance:

- 22413 • The DESCRIPTION is updated to indicate the results of naming a symbolic link in either *old*
 22414 or *new*.
- 22415 • In the ERRORS section, [EIO] is added to indicate that a physical I/O error has occurred,
 22416 [ELOOP] to indicate that too many symbolic links were encountered during pathname
 22417 resolution, and [EPERM] or [EACCES] to indicate a permission check failure when operating
 22418 on directories with S_ISVTX set.
- 22419 • In the ERRORS section, a second [ENAMETOOLONG] condition is defined that may report
 22420 excessive length of an intermediate result of pathname resolution of a symbolic link.

22421 Issue 5

22422 The [EBUSY] error is added to the “may fail” part of the ERRORS section.

22423 **NAME**

22424 rewind — reset file position indicator in a stream

22425 **SYNOPSIS**

22426 #include <stdio.h>

22427 void rewind(FILE **stream*);

22428 **DESCRIPTION**

22429 The call:

22430 rewind(*stream*)

22431 is equivalent to:

22432 (void) fseek(*stream*, 0L, SEEK_SET)

22433 except that *rewind()* also clears the error indicator.

22434 **RETURN VALUE**

22435 The *rewind()* function returns no value.

22436 **ERRORS**

22437 Refer to *fseek()* with the exception of [EINVAL] which does not apply.

22438 **EXAMPLES**

22439 None.

22440 **APPLICATION USAGE**

22441 Because *rewind()* does not return a value, an application wishing to detect errors should clear
22442 *errno*, then call *rewind()*, and if *errno* is non-zero, assume an error has occurred.

22443 **FUTURE DIRECTIONS**

22444 None.

22445 **SEE ALSO**

22446 *fseek()*, <stdio.h>.

22447 **CHANGE HISTORY**

22448 First released in Issue 1.

22449 Derived from Issue 1 of the SVID.

22450 **NAME**

22451 rewinddir — reset position of directory stream to the beginning of a directory

22452 **SYNOPSIS**

22453 OH #include <sys/types.h>

22454 #include <dirent.h>

22455 void rewinddir(DIR *dirp);

22456 **DESCRIPTION**

22457 The *rewinddir()* function resets the position of the directory stream to which *dirp* refers to the
 22458 beginning of the directory. It also causes the directory stream to refer to the current state of the
 22459 corresponding directory, as a call to *opendir()* would have done. If *dirp* does not refer to a
 22460 directory stream, the effect is undefined.

22461 After a call to the *fork()* function, either the parent or child (but not both) may continue
 22462 EX processing the directory stream using *readdir()*, *rewinddir()* or *seekdir()*. If both the parent and
 22463 child processes use these functions, the result is undefined.

22464 **RETURN VALUE**22465 The *rewinddir()* function does not return a value.22466 **ERRORS**

22467 No errors are defined.

22468 **EXAMPLES**

22469 None.

22470 **APPLICATION USAGE**

22471 The *rewinddir()* function should be used in conjunction with *opendir()*, *readdir()* and *closedir()* to
 22472 examine the contents of the directory. This method is recommended for portability.

22473 **FUTURE DIRECTIONS**

22474 None.

22475 **SEE ALSO**22476 *closedir()*, *opendir()*, *readdir()*, <dirent.h>, <sys/types.h>.22477 **CHANGE HISTORY**

22478 First released in Issue 2.

22479 **Issue 4**

22480 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 22481 • The last paragraph of the DESCRIPTION, describing a restriction after a *fork()* function is
- 22482 added.

22483 Other changes are incorporated as follows:

- 22484 • The <sys/types.h> header is now marked as optional (OH); this header need not be included
- 22485 on XSI-conformant systems.

22486 **NAME**

22487 rindex — character string operations

22488 **SYNOPSIS**

22489 EX #include <strings.h>

22490 char *rindex(const char *s, int c);

22491

22492 **DESCRIPTION**22493 The *rindex()* function is identical to *strrchr()*.22494 **RETURN VALUE**22495 See *strrchr()*.22496 **ERRORS**22497 See *strrchr()*.22498 **EXAMPLES**

22499 None.

22500 **APPLICATION USAGE**22501 For portability to implementations conforming to earlier versions of this specification, *strrchr()*
22502 is preferred over this function.22503 **FUTURE DIRECTIONS**

22504 None.

22505 **SEE ALSO**22506 *strrchr()*, <strings.h>.22507 **CHANGE HISTORY**

22508 First released in Issue 4, Version 2.

22509 **Issue 5**

22510 Moved from X/OPEN UNIX extension to BASE.

22511 NAME

22512 rint — round-to-nearest integral value

22513 SYNOPSIS

22514 EX #include <math.h>

22515 double rint(double x);

22516

22517 DESCRIPTION

22518 The *rint()* function returns the integral value (represented as a **double**) nearest *x* in the direction
22519 of the current rounding mode. The current rounding mode is implementation-dependent.

22520 If the current rounding mode rounds toward negative infinity, then *rint()* is identical to *floor()*.

22521 If the current rounding mode rounds toward positive infinity, then *rint()* is identical to *ceil()*.

22522 RETURN VALUE

22523 Upon successful completion, the *rint()* function returns the integer (represented as a double
22524 precision number) nearest *x* in the direction of the current rounding mode.

22525 When *x* is $\pm\text{Inf}$, *rint()* returns *x*.

22526 If the value of *x* is NaN, NaN is returned and *errno* may be set to [EDOM].

22527 ERRORS

22528 The *rint()* function may fail if:

22529 [EDOM] The *x* argument is NaN.

22530 EXAMPLES

22531 None.

22532 APPLICATION USAGE

22533 None.

22534 FUTURE DIRECTIONS

22535 None.

22536 SEE ALSO

22537 *abs()*, *isnan()*, <math.h>.

22538 CHANGE HISTORY

22539 First released in Issue 4, Version 2.

22540 Issue 5

22541 Moved from X/OPEN UNIX extension to BASE.

22542 **NAME**

22543 rmdir — remove a directory

22544 **SYNOPSIS**

22545 #include <unistd.h>

22546 int rmdir(const char *path);

22547 **DESCRIPTION**

22548 The *rmdir()* function removes a directory whose name is given by *path*. The directory is
 22549 removed only if it is an empty directory.

22550 If the directory is the root directory or the current working directory of any process, it is
 22551 unspecified whether the function succeeds, or whether it fails and sets *errno* to [EBUSY].

22552 EX If *path* names a symbolic link, then *rmdir()* fails and sets *errno* to [ENOTDIR].

22553 If the directory's link count becomes 0 and no process has the directory open, the space occupied
 22554 by the directory will be freed and the directory will no longer be accessible. If one or more
 22555 processes have the directory open when the last link is removed, the dot and dot-dot entries, if
 22556 present, are removed before *rmdir()* returns and no new entries may be created in the directory,
 22557 but the directory is not removed until all references to the directory are closed.

22558 Upon successful completion, the *rmdir()* function marks for update the *st_ctime* and *st_mtime*
 22559 fields of the parent directory.

22560 **RETURN VALUE**

22561 Upon successful completion, the function *rmdir()* returns 0. Otherwise, -1 is returned, and *errno*
 22562 is set to indicate the error. If -1 is returned, the named directory is not changed.

22563 **ERRORS**22564 The *rmdir()* function will fail if:

22565 [EACCES] Search permission is denied on a component of the path prefix, or write
 22566 permission is denied on the parent directory of the directory to be removed.

22567 [EBUSY] The directory to be removed is currently in use by the system or another
 22568 process and the implementation considers this to be an error.

22569 [EEXIST] or [ENOTEMPTY]

22570 The *path* argument names a directory that is not an empty directory.

22571 EX [EIO] A physical I/O error has occurred.

22572 EX [ELOOP] Too many symbolic links were encountered in resolving *path*.

22573 [ENAMETOOLONG]

22574 FIPS The length of the *path* argument exceeds {PATH_MAX} or a pathname
 22575 component is longer than {NAME_MAX}.

22576 [ENOENT] A component of *path* does not name an existing file, or the *path* argument
 22577 names a non-existent directory or points to an empty string.

22578 [ENOTDIR] A component of the path is not a directory.

22579 [EPERM] or [EACCES]

22580 EX The S_ISVTX flag is set on the parent directory of the directory to be removed
 22581 and the caller is not the owner of the directory to be removed, nor is the caller
 22582 the owner of the parent directory, nor does the caller have the appropriate
 22583 privileges.

22584	[EROFS]	The directory entry to be removed resides on a read-only file system.	
22585		The <code>rmdir()</code> function may fail if:	
22586 EX	[ENAMETOOLONG]		
22587		Pathname resolution of a symbolic link produced an intermediate result	
22588		whose length exceeds {PATH_MAX}.	
22589	EXAMPLES		
22590		None.	
22591	APPLICATION USAGE		
22592		None.	
22593	FUTURE DIRECTIONS		
22594		None.	
22595	SEE ALSO		
22596		<code>mkdir()</code> , <code>remove()</code> , <code>unlink()</code> , <code><unistd.h></code> .	
22597	CHANGE HISTORY		
22598		First released in Issue 3.	
22599		Entry included for alignment with the POSIX.1-1988 standard.	
22600	Issue 4		
22601		The following changes are incorporated for alignment with the ISO POSIX-1 standard:	
22602		<ul style="list-style-type: none"> The type of argument <i>path</i> is changed from <code>char *</code> to <code>const char *</code>. 	
22603		<ul style="list-style-type: none"> 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 <i>errno</i> to [EBUSY]. In Issue 3, the behaviour under these circumstances was defined as “implementation-dependent”. 	
22604			
22605			
22606			
22607		<ul style="list-style-type: none"> The RETURN VALUE section is expanded to direct that if <code>-1</code> is returned, the directory will not be changed. 	
22608			
22609		The following change is incorporated for alignment with the FIPS requirements:	
22610		<ul style="list-style-type: none"> In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a pathname component is larger than {NAME_MAX} is now defined as mandatory and marked as an extension. 	
22611			
22612			
22613		Other changes are incorporated as follows:	
22614		<ul style="list-style-type: none"> The header <code><unistd.h></code> is added to the SYNOPSIS section. 	
22615		<ul style="list-style-type: none"> The [ENAMETOOLONG] description is amended. 	
22616	Issue 4, Version 2		
22617		The following changes are made for X/OPEN UNIX conformance:	
22618		<ul style="list-style-type: none"> The DESCRIPTION is updated to indicate the results of naming a symbolic link in <i>path</i>. 	
22619		<ul style="list-style-type: none"> 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 pathname resolution, and [EPERM] or [EACCES] to indicate a permission check failure when operating on directories with S_ISVTX set. 	
22620			
22621			
22622			
22623		<ul style="list-style-type: none"> In the ERRORS section, a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link. 	
22624			

22625 **NAME**22626 sbrk — change space allocation (**LEGACY**)22627 **SYNOPSIS**

22628 EX #include <unistd.h>

22629 void *sbrk(intptr_t incr);

22630

22631 **DESCRIPTION**22632 Refer to *brk()*.22633 **CHANGE HISTORY**

22634 First released in Issue 4, Version 2.

22635 **Issue 5**

22636 Moved from X/OPEN UNIX extension to BASE.

22637 Marked LEGACY.

22638 The type of the argument to *sbrk()* is changed from **int** to **intptr_t**.

22639 **NAME**

22640 scalb — load exponent of a radix-independent floating-point number

22641 **SYNOPSIS**

22642 EX #include <math.h>

22643 double scalb(double x, double n);

22644

22645 **DESCRIPTION**22646 The *scalb()* function computes $x * r^n$, where r is the radix of the machine's floating point
22647 arithmetic. When r is 2, *scalb()* is equivalent to *ldexp()*.22648 An application wishing to check for error situations should set *errno* to 0 before calling *scalb()*. If
22649 *errno* is non-zero on return, or the return value is NaN, an error has occurred.22650 **RETURN VALUE**22651 Upon successful completion, the *scalb()* function returns $x * r^n$.22652 If the correct value would overflow, *scalb()* returns $\pm\text{HUGE_VAL}$ (according to the sign of x) and
22653 sets *errno* to [ERANGE].22654 If the correct value would underflow, *scalb()* returns 0 and sets *errno* to [ERANGE].22655 The *scalb()* function returns x when x is $\pm\text{Inf}$.22656 If x or n is NaN, then *scalb()* returns NaN and may set *errno* to [EDOM].22657 **ERRORS**22658 The *scalb()* function will fail if:

22659 [ERANGE] The correct value would overflow or underflow.

22660 The *scalb()* function may fail if:22661 [EDOM] The x or n argument is NaN.22662 **EXAMPLES**

22663 None.

22664 **APPLICATION USAGE**

22665 None.

22666 **FUTURE DIRECTIONS**

22667 None.

22668 **SEE ALSO**22669 *ldexp()*, <math.h>.22670 **CHANGE HISTORY**

22671 First released in Issue 4, Version 2.

22672 **Issue 5**

22673 Moved from X/OPEN UNIX extension to BASE.

22674 The DESCRIPTION is updated to indicate how an application should check for an error. This
22675 text was previously published in the APPLICATION USAGE section.

22676 **NAME**

22677 scanf — convert formatted input

22678 **SYNOPSIS**

22679 #include <stdio.h>

22680 int scanf(const char **format*, . . .);22681 **DESCRIPTION**22682 Refer to *fscanf()*.22683 **CHANGE HISTORY**

22684 First released in Issue 1.

22685 Derived from Issue 1 of the SVID.

22686 **Issue 4**

22687 The following change is incorporated for alignment with the ISO C standard:

- 22688
- The type of the argument *format* is changed from **char *** to **const char ***.

22689 Other changes are incorporated as follows:

- 22690
- The description of this function, including its change history, is located under *fscanf()*.

22691 NAME

22692 `sched_get_priority_max`, `sched_get_priority_min` — get priority limits (**REALTIME**)

22693 SYNOPSIS

22694 RT `#include <sched.h>`

22695 `int sched_get_priority_max(int policy);`

22696 `int sched_get_priority_min(int policy);`

22697

22698 DESCRIPTION

22699 The `sched_get_priority_max()` and `sched_get_priority_min()` functions return the appropriate
22700 maximum or minimum, respectfully, for the scheduling policy specified by *policy*.

22701 The value of *policy* is one of the scheduling policy values defined in `<sched.h>`.

22702 RETURN VALUE

22703 If successful, the `sched_get_priority_max()` and `sched_get_priority_min()` functions return the
22704 appropriate maximum or minimum values, respectively. If unsuccessful, they return a value of
22705 `-1` and set *errno* to indicate the error.

22706 ERRORS

22707 The `sched_get_priority_max()` and `sched_get_priority_min()` functions will fail if:

22708 [EINVAL] The value of the *policy* parameter does not represent a defined scheduling
22709 policy.

22710 [ENOSYS] The `sched_get_priority_max()`, `sched_get_priority_min()` and
22711 `sched_rr_get_interval()` functions are not supported by this implementation.

22712 EXAMPLES

22713 None.

22714 APPLICATION USAGE

22715 None.

22716 FUTURE DIRECTIONS

22717 None.

22718 SEE ALSO

22719 `sched_getparam()`, `sched_setparam()`, `sched_getscheduler()`, `sched_rr_get_interval()`,
22720 `sched_setscheduler()`, `<sched.h>`.

22721 CHANGE HISTORY

22722 First released in Issue 5.

22723 Included for alignment with the POSIX Realtime Extension.

22724 NAME

22725 sched_getparam — get scheduling parameters (**REALTIME**)

22726 SYNOPSIS

22727 RT `#include <sched.h>`

22728 `int sched_getparam(pid_t pid, struct sched_param *param);`

22729

22730 DESCRIPTION

22731 The *sched_getparam()* function returns the scheduling parameters of a process specified by *pid* in
22732 the **sched_param** structure pointed to by *param*.

22733 If a process specified by *pid* exists and if the calling process has permission, the scheduling
22734 parameters for the process whose process ID is equal to *pid* will be returned.

22735 If *pid* is zero, the scheduling parameters for the calling process will be returned. The behaviour
22736 of the *sched_getparam()* function is unspecified if the value of *pid* is negative.

22737 RETURN VALUE

22738 Upon successful completion, the *sched_getparam()* function returns zero. If the call to
22739 *sched_getparam()* is unsuccessful, the function returns a value of -1 and sets *errno* to indicate the
22740 error.

22741 ERRORS

22742 The *sched_getparam()* function will fail if:

22743 [ENOSYS] The function *sched_getparam()* is not supported by this implementation.

22744 [EPERM] The requesting process does not have permission to obtain the scheduling
22745 parameters of the specified process.

22746 [ESRCH] No process can be found corresponding to that specified by *pid*.

22747 EXAMPLES

22748 None.

22749 APPLICATION USAGE

22750 None.

22751 FUTURE DIRECTIONS

22752 None.

22753 SEE ALSO

22754 *sched_getscheduler()*, *sched_setparam()*, *sched_setscheduler()*, **<sched.h>**.

22755 CHANGE HISTORY

22756 First released in Issue 5.

22757 Included for alignment with the POSIX Realtime Extension.

22758 **NAME**22759 sched_getscheduler — get scheduling policy (**REALTIME**)22760 **SYNOPSIS**

22761 RT #include <sched.h>

22762 int sched_getscheduler(pid_t pid);

22763

22764 **DESCRIPTION**22765 The *sched_getscheduler()* function returns the scheduling policy of the process specified by *pid*. If
22766 the value of *pid* is negative, the behaviour of the *sched_getscheduler()* function is unspecified.22767 The values that can be returned by *sched_getscheduler()* are defined in the header file <**sched.h**>22768 If a process specified by *pid* exists and if the calling process has permission, the scheduling
22769 policy will be returned for the process whose process ID is equal to *pid*.22770 If *pid* is zero, the scheduling policy will be returned for the calling process.22771 **RETURN VALUE**22772 Upon successful completion, the *sched_getscheduler()* function returns the scheduling policy of
22773 the specified process. If unsuccessful, the function returns -1 and sets *errno* to indicate the error.22774 **ERRORS**22775 The *sched_getscheduler()* function will fail if:22776 [ENOSYS] The function *sched_getscheduler()* is not supported by this implementation.22777 [EPERM] The requesting process does not have permission to determine the scheduling
22778 policy of the specified process.22779 [ESRCH] No process can be found corresponding to that specified by *pid*.22780 **EXAMPLES**

22781 None.

22782 **APPLICATION USAGE**

22783 None.

22784 **FUTURE DIRECTIONS**

22785 None.

22786 **SEE ALSO**22787 *sched_getparam()*, *sched_setparam()*, *sched_setscheduler()*, <**sched.h**>.22788 **CHANGE HISTORY**

22789 First released in Issue 5.

22790 Included for alignment with the POSIX Realtime Extension.

22791 NAME

22792 sched_rr_get_interval — get execution time limits (**REALTIME**)

22793 SYNOPSIS

22794 RT `#include <sched.h>`

22795 `int sched_rr_get_interval(pid_t pid, struct timespec *interval);`

22796

22797 DESCRIPTION

22798 The *sched_rr_get_interval()* function updates the **timespec** structure referenced by the *interval*
22799 argument to contain the current execution time limit (that is, time quantum) for the process
22800 specified by *pid*. If *pid* is zero, the current execution time limit for the calling process will be
22801 returned.

22802 RETURN VALUE

22803 If successful, the *sched_rr_get_interval()* function returns zero. Otherwise, it returns a value of -1
22804 and sets *errno* to indicate the error.

22805 ERRORS

22806 The *sched_rr_get_interval()* function will fail if:

- 22807 [ENOSYS] The *sched_get_priority_max()*, *sched_get_priority_min()* and
22808 *sched_rr_get_interval()* functions are not supported by this implementation.
- 22809 [ESRCH] No process can be found corresponding to that specified by *pid*.

22810 EXAMPLES

22811 None.

22812 APPLICATION USAGE

22813 None.

22814 FUTURE DIRECTIONS

22815 None.

22816 SEE ALSO

22817 *sched_getparam()*, *sched_setparam()*, *sched_get_priority_max()*, *sched_getscheduler()*,
22818 *sched_setscheduler()*, <**sched.h**>.

22819 CHANGE HISTORY

22820 First released in Issue 5.

22821 Included for alignment with the POSIX Realtime Extension.

22822 NAME

22823 sched_setparam — set scheduling parameters (**REALTIME**)

22824 SYNOPSIS

22825 RT `#include <sched.h>`22826 `int sched_setparam(pid_t pid, const struct sched_param *param);`

22827

22828 DESCRIPTION

22829 The *sched_setparam()* function sets the scheduling parameters of the process specified by *pid* to
 22830 the values specified by the **sched_param** structure pointed to by *param*. The value of the
 22831 *sched_priority* member in the **sched_param** structure is any integer within the inclusive priority
 22832 range for the current scheduling policy of the process specified by *pid*. Higher numerical values
 22833 for the priority represent higher priorities. If the value of *pid* is negative, the behaviour of the
 22834 *sched_setparam()* function is unspecified.

22835 If a process specified by *pid* exists and if the calling process has permission, the scheduling
 22836 parameters will be set for the process whose process ID is equal to *pid*.

22837 If *pid* is zero, the scheduling parameters will be set for the calling process.

22838 The conditions under which one process has permission to change the scheduling parameters of
 22839 another process are implementation-dependent.

22840 Implementations may require the requesting process to have the appropriate privilege to set its
 22841 own scheduling parameters or those of another process.

22842 The target process, whether it is running or not running, resumes execution after all other
 22843 runnable processes of equal or greater priority have been scheduled to run.

22844 If the priority of the process specified by the *pid* argument is set higher than that of the lowest
 22845 priority running process and if the specified process is ready to run, the process specified by the
 22846 *pid* argument preempts a lowest priority running process. Similarly, if the process calling
 22847 *sched_setparam()* sets its own priority lower than that of one or more other non-empty process
 22848 lists, then the process that is the head of the highest priority list also preempts the calling
 22849 process. Thus, in either case, the originating process might not receive notification of the
 22850 completion of the requested priority change until the higher priority process has executed.

22851 If the current scheduling policy for the process specified by *pid* is not SCHED_FIFO or
 22852 SCHED_RR, including SCHED_OTHER, the result is implementation-dependent.

22853 The effect of this function on individual threads is dependent on the scheduling contention
 22854 scope of the threads:

- 22855 • For threads with system scheduling contention scope, these functions have no effect on their
 22856 scheduling.

- 22857 EX • For threads with process scheduling contention scope, the threads' scheduling parameters
 22858 will not be affected. However, the scheduling of these threads with respect to threads in
 22859 other processes may be dependent on the scheduling parameters of their process, which are
 22860 governed using these functions.

22861 EX If an implementation supports a two-level scheduling model in which library threads are
 22862 multiplexed on top of several kernel scheduled entities, then the underlying kernel scheduled
 22863 entities for the system contention scope threads will not be affected by these functions.

22864 The underlying kernel scheduled entities for the process contention scope threads will have their
 22865 scheduling parameters changed to the value specified in *param*. Kernel scheduled entities for use
 22866 by process contention scope threads that are created after this call completes inherit their

22867	scheduling policy and associated scheduling parameters from the process.	
22868	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.	
22869		
22870		
22871	RETURN VALUE	
22872	If successful, the <i>sched_setparam()</i> function returns zero.	
22873	If the call to <i>sched_setparam()</i> is unsuccessful, the priority remains unchanged, and the function returns a value of <i>-1</i> and sets <i>errno</i> to indicate the error.	
22874		
22875	ERRORS	
22876	The <i>sched_setparam()</i> function will fail if:	
22877	[EINVAL]	One or more of the requested scheduling parameters is outside the range defined for the scheduling policy of the specified <i>pid</i> .
22878		
22879	[ENOSYS]	The function <i>sched_setparam()</i> is not supported by this implementation.
22880	[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 <i>sched_setparam()</i> .
22881		
22882		
22883	[ESRCH]	No process can be found corresponding to that specified by <i>pid</i> .
22884	EXAMPLES	
22885	None.	
22886	APPLICATION USAGE	
22887	None.	
22888	FUTURE DIRECTIONS	
22889	None.	
22890	SEE ALSO	
22891	<i>sched_getparam()</i> , <i>sched_getscheduler()</i> , <i>sched_setscheduler()</i> , < sched.h >.	
22892	CHANGE HISTORY	
22893	First released in Issue 5.	
22894	Included for alignment with the POSIX Realtime Extension.	

22895 NAME

22896 sched_setscheduler — set scheduling policy and parameters (**REALTIME**)

22897 SYNOPSIS

22898 RT `#include <sched.h>`

```
22899 int sched_setscheduler(pid_t pid, int policy,
22900     const struct sched_param *param);
```

22901

22902 DESCRIPTION

22903 The *sched_setscheduler()* function sets the scheduling policy and scheduling parameters of the
 22904 process specified by *pid* to *policy* and the parameters specified in the **sched_param** structure
 22905 pointed to by *param*, respectively. The value of the *sched_priority* member in the **sched_param**
 22906 structure is any integer within the inclusive priority range for the scheduling policy specified by
 22907 *policy*. If the value of *pid* is negative, the behaviour of the *sched_setscheduler()* function is
 22908 unspecified.

22909 The possible values for the *policy* parameter are defined in the header file **<sched.h>**.

22910 If a process specified by *pid* exists and if the calling process has permission, the scheduling
 22911 policy and scheduling parameters will be set for the process whose process ID is equal to *pid*.

22912 If *pid* is zero, the scheduling policy and scheduling parameters will be set for the calling process.

22913 The conditions under which one process has the appropriate privilege to change the scheduling
 22914 parameters of another process are implementation-dependent.

22915 Implementations may require that the requesting process have permission to set its own
 22916 scheduling parameters or those of another process. Additionally, implementation-dependent
 22917 restrictions may apply as to the appropriate privileges required to set a process's own
 22918 scheduling policy, or another process's scheduling policy, to a particular value.

22919 The *sched_setscheduler()* function is considered successful if it succeeds in setting the scheduling
 22920 policy and scheduling parameters of the process specified by *pid* to the values specified by *policy*
 22921 and the structure pointed to by *param*, respectively.

22922 The effect of this function on individual threads is dependent on the scheduling contention
 22923 scope of the threads:

- 22924 • For threads with system scheduling contention scope, these functions have no effect on their
 22925 scheduling.

- 22926 EX • For threads with process scheduling contention scope, the threads' scheduling policy and
 22927 associated parameters will not be affected. However, the scheduling of these threads with
 22928 respect to threads in other processes may be dependent on the scheduling parameters of their
 22929 process, which are governed using these functions.

22930 EX If an implementation supports a two-level scheduling model in which library threads are
 22931 multiplexed on top of several kernel scheduled entities, then the underlying kernel scheduled
 22932 entities for the system contention scope threads will not be affected by these functions.

22933 The underlying kernel scheduled entities for the process contention scope threads will have their
 22934 scheduling policy and associated scheduling parameters changed to the values specified in
 22935 *policy* and *param*, respectively. Kernel scheduled entities for use by process contention scope
 22936 threads that are created after this call completes inherit their scheduling policy and associated
 22937 scheduling parameters from the process.

22938 This function is not atomic with respect to other threads in the process. Threads are allowed to
 22939 continue to execute while this function call is in the process of changing the scheduling policy

22940 and associated scheduling parameters for the underlying kernel scheduled entities used by the
22941 process contention scope threads.

22942 RETURN VALUE

22943 Upon successful completion, the function returns the former scheduling policy of the specified
22944 process. If the *sched_setscheduler()* function fails to complete successfully, the policy and
22945 scheduling parameters remain unchanged, and the function returns a value of *-1* and sets *errno*
22946 to indicate the error.

22947 ERRORS

22948 The *sched_setscheduler()* function will fail if:

22949 [EINVAL] The value of the *policy* parameter is invalid, or one or more of the parameters
22950 contained in *param* is outside the valid range for the specified scheduling
22951 policy.

22952 [ENOSYS] The function *sched_setscheduler()* is not supported by this implementation.

22953 [EPERM] The requesting process does not have permission to set either or both of the
22954 scheduling parameters or the scheduling policy of the specified process.

22955 [ESRCH] No process can be found corresponding to that specified by *pid*.

22956 EXAMPLES

22957 None.

22958 APPLICATION USAGE

22959 None.

22960 FUTURE DIRECTIONS

22961 None.

22962 SEE ALSO

22963 *sched_getparam()*, *sched_getscheduler()*, *sched_setparam()*, **<sched.h>**.

22964 CHANGE HISTORY

22965 First released in Issue 5.

22966 Included for alignment with the POSIX Realtime Extension.

22967 **NAME**

22968 sched_yield — yield processor

22969 **SYNOPSIS**

22970 #include <sched.h>

22971 int sched_yield(void);

22972 **DESCRIPTION**

22973 The *sched_yield()* function forces the running thread to relinquish the processor until it again
22974 becomes the head of its thread list. It takes no arguments.

22975 **RETURN VALUE**

22976 The *sched_yield()* function returns 0 if it completes successfully, or it returns a value of -1 and
22977 sets *errno* to indicate the error.

22978 **ERRORS**

22979 No errors are defined.

22980 **EXAMPLES**

22981 None.

22982 **APPLICATION USAGE**

22983 None.

22984 **FUTURE DIRECTIONS**

22985 None.

22986 **SEE ALSO**

22987 <sched.h>.

22988 **CHANGE HISTORY**

22989 First released in Issue 5.

22990 Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

22991 **NAME**

22992 seed48 — seed uniformly distributed pseudo-random non-negative long integer generator

22993 **SYNOPSIS**

22994 EX `#include <stdlib.h>`

22995 `unsigned short int *seed48(unsigned short int seed16v[3]);`

22996

22997 **DESCRIPTION**

22998 Refer to *drand48()*.

22999 **CHANGE HISTORY**

23000 First released in Issue 1.

23001 Derived from Issue 1 of the SVID.

23002 **Issue 4**

23003 The following change is incorporated in this issue:

- 23004 • The header `<stdlib.h>` is added to the SYNOPSIS section.

23005 **NAME**

23006 seekdir — set position of directory stream

23007 **SYNOPSIS**

23008 EX OH #include <sys/types.h>

23009 EX #include <dirent.h>

23010 void seekdir(DIR *dirp, long int loc);

23011

23012 **DESCRIPTION**

23013 The *seekdir()* function sets the position of the next *readdir()* operation on the directory stream specified by *dirp* to the position specified by *loc*. The value of *loc* should have been returned from an earlier call to *telldir()*. The new position reverts to the one associated with the directory stream when *telldir()* was performed.

23017 If the value of *loc* was not obtained from an earlier call to *telldir()* or if a call to *rewinddir()* occurred between the call to *telldir()* and the call to *seekdir()*, the results of subsequent calls to *readdir()* are unspecified.

23020 **RETURN VALUE**23021 The *seekdir()* function returns no value.23022 **ERRORS**

23023 No errors are defined.

23024 **EXAMPLES**

23025 None.

23026 **APPLICATION USAGE**

23027 None.

23028 **FUTURE DIRECTIONS**

23029 None.

23030 **SEE ALSO**23031 *opendir()*, *readdir()*, *telldir()*, <dirent.h> <stdio.h>, <sys/types.h>.23032 **CHANGE HISTORY**

23033 First released in Issue 2.

23034 **Issue 4**

23035 The following changes are incorporated in this issue:

23036 • The <sys/types.h> header is now marked as optional (OH); this header need not be included
23037 on XSI-conformant systems.

23038 • The type of argument *loc* is expanded to **long int**.

23039 **Issue 4, Version 2**

23040 The DESCRIPTION is updated for X/OPEN UNIX conformance to indicate that a call to
23041 *readdir()* may produce unspecified results if either *loc* was not obtained by a previous call to
23042 *telldir()*, or if there is an intervening call to *rewinddir()*.

23043 NAME

23044 select — synchronous I/O multiplexing

23045 SYNOPSIS

23046 EX `#include <sys/time.h>`

```

23047 int select(int nfds, fd_set *readfds, fd_set *writefds,
23048           fd_set *errorfds, struct timeval *timeout);
23049 void FD_CLR(int fd, fd_set *fdset);
23050 int FD_ISSET(int fd, fd_set *fdset);
23051 void FD_SET(int fd, fd_set *fdset);
23052 void FD_ZERO(fd_set *fdset);
23053

```

23054 DESCRIPTION

23055 The *select()* function indicates which of the specified file descriptors is ready for reading, ready
 23056 for writing, or has an error condition pending. If the specified condition is false for all of the
 23057 specified file descriptors, *select()* blocks, up to the specified timeout interval, until the specified
 23058 condition is true for at least one of the specified file descriptors.

23059 The *select()* function supports regular files, terminal and pseudo-terminal devices, STREAMS-
 23060 based files, FIFOs and pipes. The behaviour of *select()* on file descriptors that refer to other types
 23061 of file is unspecified.

23062 The *nfds* argument specifies the range of file descriptors to be tested. The *select()* function tests
 23063 file descriptors in the range of 0 to *nfds*–1.

23064 If the *readfds* argument is not a null pointer, it points to an object of type **fd_set** that on input
 23065 specifies the file descriptors to be checked for being ready to read, and on output indicates
 23066 which file descriptors are ready to read.

23067 If the *writefs* argument is not a null pointer, it points to an object of type **fd_set** that on input
 23068 specifies the file descriptors to be checked for being ready to write, and on output indicates
 23069 which file descriptors are ready to write.

23070 If the *errorfds* argument is not a null pointer, it points to an object of type **fd_set** that on input
 23071 specifies the file descriptors to be checked for error conditions pending, and on output indicates
 23072 which file descriptors have error conditions pending.

23073 On successful completion, the objects pointed to by the *readfs*, *writefs*, and *errorfds* arguments are
 23074 modified to indicate which file descriptors are ready for reading, ready for writing, or have an
 23075 error condition pending, respectively. For each file descriptor less than *nfds*, the corresponding
 23076 bit will be set on successful completion if it was set on input and the associated condition is true
 23077 for that file descriptor.

23078 If the *timeout* argument is not a null pointer, it points to an object of type **struct timeval** that
 23079 specifies a maximum interval to wait for the selection to complete. If the *timeout* argument
 23080 points to an object of type **struct timeval** whose members are 0, *select()* does not block. If the
 23081 *timeout* argument is a null pointer, *select()* blocks until an event causes one of the masks to be
 23082 returned with a valid (non-zero) value. If the time limit expires before any event occurs that
 23083 would cause one of the masks to be set to a non-zero value, *select()* completes successfully and
 23084 returns 0.

23085 The use of a timeout does not affect any pending timers set up by *alarm()*, *ualarm()* or *settimer()*.

23086 On successful completion, the object pointed to by the *timeout* argument may be modified.

23087 Implementations may place limitations on the maximum timeout interval supported. On all
 23088 implementations, the maximum timeout interval supported will be at least 31 days. If the

23089 *timeout* argument specifies a timeout interval greater than the implementation-dependent
 23090 maximum value, the maximum value will be used as the actual timeout value. Implementations
 23091 may also place limitations on the granularity of timeout intervals. If the requested timeout
 23092 interval requires a finer granularity than the implementation supports, the actual timeout
 23093 interval will be rounded up to the next supported value.

23094 If the *readfs*, *writefs*, and *errorfds* arguments are all null pointers and the *timeout* argument is not a
 23095 null pointer, *select()* blocks for the time specified, or until interrupted by a signal. If the *readfs*,
 23096 *writefs*, and *errorfds* arguments are all null pointers and the *timeout* argument is a null pointer,
 23097 *select()* blocks until interrupted by a signal.

23098 File descriptors associated with regular files always select true for ready to read, ready to write,
 23099 and error conditions.

23100 On failure, the objects pointed to by the *readfs*, *writefs*, and *errorfds* arguments are not modified.
 23101 If the timeout interval expires without the specified condition being true for any of the specified
 23102 file descriptors, the objects pointed to by the *readfs*, *writefs*, and *errorfds* arguments have all bits
 23103 set to 0.

23104 File descriptor masks of type **fd_set** can be initialised and tested with **FD_CLR()**, **FD_ISSET()**,
 23105 **FD_SET()**, and **FD_ZERO()**. It is unspecified whether each of these is a macro or a function. If a
 23106 macro definition is suppressed in order to access an actual function, or a program defines an
 23107 external identifier with any of these names, the behaviour is undefined.

23108 **FD_CLR(*fd*, &*fdset*)** Clears the bit for the file descriptor *fd* in the file descriptor set *fdset*.

23109 **FD_ISSET(*fd*, &*fdset*)** Returns a non-zero value if the bit for the file descriptor *fd* is set in the file
 23110 descriptor set pointed to by *fdset*, and 0 otherwise.

23111 **FD_SET(*fd*, &*fdset*)** Sets the bit for the file descriptor *fd* in the file descriptor set *fdset*.

23112 **FD_ZERO(&*fdset*)** Initialises the file descriptor set *fdset* to have zero bits for all file
 23113 descriptors.

23114 The behaviour of these macros is undefined if the *fd* argument is less than 0 or greater than or
 23115 equal to **FD_SETSIZE**, or if any of the arguments are expressions with side effects.

23116 RETURN VALUE

23117 **FD_CLR()**, **FD_SET()** and **FD_ZERO()** return no value. **FD_ISSET()** a non-zero value if the bit
 23118 for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and 0 otherwise.

23119 On successful completion, *select()* returns the total number of bits set in the bit masks.
 23120 Otherwise, **-1** is returned, and *errno* is set to indicate the error.

23121 ERRORS

23122 Under the following conditions, *select()* fails and sets *errno* to:

23123 **[EBADF]** One or more of the file descriptor sets specified a file descriptor that is not a
 23124 valid open file descriptor.

23125 **[EINTR]** The *select()* function was interrupted before any of the selected events
 23126 occurred and before the timeout interval expired.

23127 If **SA_RESTART** has been set for the interrupting signal, it is implementation-
 23128 dependent whether *select()* restarts or returns with **[EINTR]**.

23129 **[EINVAL]** An invalid timeout interval was specified.

23130 **[EINVAL]** The *nfds* argument is less than 0 or greater than **FD_SETSIZE**.

23131	[EINVAL]	One of the specified file descriptors refers to a STREAM or multiplexer that is	
23132		linked (directly or indirectly) downstream from a multiplexer.	
23133	EXAMPLES		
23134	None.		
23135	APPLICATION USAGE		
23136	None.		
23137	FUTURE DIRECTIONS		
23138	None.		
23139	SEE ALSO		
23140	<i>fcntl()</i> , <i>poll()</i> , <i>read()</i> , <i>write()</i> , <i><sys/time.h></i> .		
23141	CHANGE HISTORY		
23142	First released in Issue 4, Version 2.		
23143	Issue 5		
23144	Moved from X/OPEN UNIX extension to BASE.		
23145	In the ERRORS section, the text has been changed to indicate that [EINVAL] will be returned		
23146	when <i>nfds</i> is less than 0 or greater than FD_SETSIZE. It previously stated less than 0, or greater		
23147	than or equal to FD_SETSIZE.		
23148	Text about timeout is moved from the APPLICATION USAGE section to the DESCRIPTION.		

23149 **NAME**23150 sem_close — close a named semaphore (**REALTIME**)23151 **SYNOPSIS**23152 RT `#include <semaphore.h>`23153 `int sem_close(sem_t *sem);`

23154

23155 **DESCRIPTION**

23156 The *sem_close()* function is used to indicate that the calling process is finished using the named
 23157 semaphore indicated by *sem*. The effects of calling *sem_close()* for an unnamed semaphore (one
 23158 created by *sem_init()*) are undefined. The *sem_close()* function deallocates (that is, make
 23159 available for reuse by a subsequent *sem_open()* by this process) any system resources allocated
 23160 by the system for use by this process for this semaphore. The effect of subsequent use of the
 23161 semaphore indicated by *sem* by this process is undefined. If the semaphore has not been
 23162 removed with a successful call to *sem_unlink()*, then *sem_close()* has no effect on the state of the
 23163 semaphore. If the *sem_unlink()* function has been successfully invoked for *name* after the most
 23164 recent call to *sem_open()* with *O_CREAT* for this semaphore, then when all processes that have
 23165 opened the semaphore close it, the semaphore is no longer be accessible.

23166 **RETURN VALUE**

23167 Upon successful completion, a value of zero is returned. Otherwise, a value of *-1* is returned
 23168 and *errno* is set to indicate the error.

23169 **ERRORS**23170 The *sem_close()* function will fail if:

- | | | |
|-------|----------|--|
| 23171 | [EINVAL] | The <i>sem</i> argument is not a valid semaphore descriptor. |
| 23172 | [ENOSYS] | The function <i>sem_close()</i> is not supported by this implementation. |

23173 **EXAMPLES**

23174 None.

23175 **APPLICATION USAGE**

23176 None.

23177 **FUTURE DIRECTIONS**

23178 None.

23179 **SEE ALSO**23180 *semctl()*, *semget()*, *semop()*, *sem_init()*, *sem_open()*, *sem_unlink()*, <semaphore.h>.23181 **CHANGE HISTORY**

23182 First released in Issue 5.

23183 Included for alignment with the POSIX Realtime Extension.

23184 **NAME**23185 sem_destroy — destroy an unnamed semaphore (**REALTIME**)23186 **SYNOPSIS**23187 RT `#include <semaphore.h>`23188 `int sem_destroy(sem_t *sem);`

23189

23190 **DESCRIPTION**

23191 The *sem_destroy()* function is used to destroy the unnamed semaphore indicated by *sem*. Only a
 23192 semaphore that was created using *sem_init()* may be destroyed using *sem_destroy()*; the effect of
 23193 calling *sem_destroy()* with a named semaphore is undefined. The effect of subsequent use of the
 23194 semaphore *sem* is undefined until *sem* is re-initialised by another call to *sem_init()*.

23195 It is safe to destroy an initialised semaphore upon which no threads are currently blocked. The
 23196 effect of destroying a semaphore upon which other threads are currently blocked is undefined.

23197 **RETURN VALUE**

23198 Upon successful completion, a value of zero is returned. Otherwise, a value of -1 is returned
 23199 and *errno* is set to indicate the error.

23200 **ERRORS**23201 The *sem_destroy()* function will fail if:23202 [EINVAL] The *sem* argument is not a valid semaphore.23203 [ENOSYS] The function *sem_destroy()* is not supported by this implementation.23204 The *sem_destroy()* function may fail if:

23205 [EBUSY] There are currently processes blocked on the semaphore.

23206 **EXAMPLES**

23207 None.

23208 **APPLICATION USAGE**

23209 None.

23210 **FUTURE DIRECTIONS**

23211 None.

23212 **SEE ALSO**23213 *semctl()*, *semget()*, *semop()*, *sem_init()*, *sem_open()*, <semaphore.h>.23214 **CHANGE HISTORY**

23215 First released in Issue 5.

23216 Included for alignment with the POSIX Realtime Extension.

23217 **NAME**23218 `sem_getvalue` — get the value of a semaphore (**REALTIME**)23219 **SYNOPSIS**23220 RT `#include <semaphore.h>`23221 `int sem_getvalue(sem_t *sem, int *sval);`

23222

23223 **DESCRIPTION**

23224 The `sem_getvalue()` function updates the location referenced by the `sval` argument to have the
 23225 value of the semaphore referenced by `sem` without affecting the state of the semaphore. The
 23226 updated value represents an actual semaphore value that occurred at some unspecified time
 23227 during the call, but it need not be the actual value of the semaphore when it is returned to the
 23228 calling process.

23229 If `sem` is locked, then the value returned by `sem_getvalue()` is either zero or a negative number
 23230 whose absolute value represents the number of processes waiting for the semaphore at some
 23231 unspecified time during the call.

23232 **RETURN VALUE**

23233 Upon successful completion, the function returns a value of zero. Otherwise, the function
 23234 returns a value of `-1` and sets `errno` to indicate the error.

23235 **ERRORS**23236 The `sem_getvalue()` function will fail if:23237 [EINVAL] The `sem` argument does not refer to a valid semaphore.23238 [ENOSYS] The function `sem_getvalue()` is not supported by this implementation.23239 **EXAMPLES**

23240 None.

23241 **APPLICATION USAGE**

23242 None.

23243 **FUTURE DIRECTIONS**

23244 None.

23245 **SEE ALSO**23246 `semctl()`, `semget()`, `semop()`, `sem_post()`, `sem_trywait()`, `sem_wait()`, `<semaphore.h>`.23247 **CHANGE HISTORY**

23248 First released in Issue 5.

23249 Included for alignment with the POSIX Realtime Extension.

23250 **NAME**23251 sem_init — initialise an unnamed semaphore (**REALTIME**)23252 **SYNOPSIS**23253 RT `#include <semaphore.h>`23254 `int sem_init(sem_t *sem, int pshared, unsigned int value);`

23255

23256 **DESCRIPTION**

23257 The *sem_init()* function is used to initialise the unnamed semaphore referred to by *sem*. The
 23258 value of the initialised semaphore is *value*. Following a successful call to *sem_init()*, the
 23259 semaphore may be used in subsequent calls to *sem_wait()*, *sem_trywait()*, *sem_post()*, and
 23260 *sem_destroy()*. This semaphore remains usable until the semaphore is destroyed.

23261 If the *pshared* argument has a non-zero value, then the semaphore is shared between processes;
 23262 in this case, any process that can access the semaphore *sem* can use *sem* for performing
 23263 *sem_wait()*, *sem_trywait()*, *sem_post()*, and *sem_destroy()* operations.

23264 Only *sem* itself may be used for performing synchronisation. The result of referring to copies of
 23265 *sem* in calls to *sem_wait()*, *sem_trywait()*, *sem_post()*, and *sem_destroy()*, is undefined.

23266 If the *pshared* argument is zero, then the semaphore is shared between threads of the process; any
 23267 thread in this process can use *sem* for performing *sem_wait()*, *sem_trywait()*, *sem_post()*, and
 23268 *sem_destroy()* operations. The use of the semaphore by threads other than those created in the
 23269 same process is undefined.

23270 Attempting to initialise an already initialised semaphore results in undefined behaviour.

23271 **RETURN VALUE**

23272 Upon successful completion, the function initialises the semaphore in *sem*. Otherwise, it returns
 23273 `-1` and sets *errno* to indicate the error.

23274 **ERRORS**

23275 The *sem_init()* function will fail if:

- | | | |
|-------|----------|--|
| 23276 | [EINVAL] | The <i>value</i> argument exceeds <i>SEM_VALUE_MAX</i> . |
| 23277 | [ENOSPC] | A resource required to initialise the semaphore has been exhausted, or the |
| 23278 | | limit on semaphores (<i>SEM_NSEMS_MAX</i>) has been reached. |
| 23279 | [ENOSYS] | The function <i>sem_init()</i> is not supported by this implementation. |
| 23280 | [EPERM] | The process lacks the appropriate privileges to initialise the semaphore. |

23281 **EXAMPLES**

23282 None.

23283 **APPLICATION USAGE**

23284 None.

23285 **FUTURE DIRECTIONS**

23286 None.

23287 **SEE ALSO**

23288 *sem_destroy()*, *sem_post()*, *sem_trywait()*, *sem_wait()*, *<semaphore.h>*.

23289 **CHANGE HISTORY**

23290 First released in Issue 5.

23291 Included for alignment with the POSIX Realtime Extension.

23292 NAME

23293 sem_open — initialise and open a named semaphore (**REALTIME**)

23294 SYNOPSIS

23295 RT

```
#include <semaphore.h>
```

23296

```
sem_t *sem_open(const char *name, int oflag, ...);
```

23297

23298 DESCRIPTION

23299 The *sem_open()* function establishes a connection between a named semaphore and a process.
 23300 Following a call to *sem_open()* with semaphore name *name*, the process may reference the
 23301 semaphore associated with *name* using the address returned from the call. This semaphore may
 23302 be used in subsequent calls to *sem_wait()*, *sem_trywait()*, *sem_post()*, and *sem_close()*. The
 23303 semaphore remains usable by this process until the semaphore is closed by a successful call to
 23304 *sem_close()*, *_exit()*, or one of the *exec* functions.

23305 The *oflag* argument controls whether the semaphore is created or merely accessed by the call to
 23306 *sem_open()*. The following flag bits may be set in *oflag*:

23307 **O_CREAT** This flag is used to create a semaphore if it does not already exist. If
 23308 **O_CREAT** is set and the semaphore already exists, then **O_CREAT** has no
 23309 effect, except as noted under **O_EXCL**. Otherwise, *sem_open()* creates a
 23310 named semaphore. The **O_CREAT** flag requires a third and a fourth
 23311 argument: *mode*, which is of type **mode_t**, and *value*, which is of type
 23312 **unsigned int**. The semaphore is created with an initial value of *value*. Valid
 23313 initial values for semaphores are less than or equal to **SEM_VALUE_MAX**.

23314 The user ID of the semaphore is set to the effective user ID of the process; the
 23315 group ID of the semaphore is set to a system default group ID or to the
 23316 effective group ID of the process. The permission bits of the semaphore are
 23317 set to the value of the *mode* argument except those set in the file mode creation
 23318 mask of the process. When bits in *mode* other than the file permission bits are
 23319 specified, the effect is unspecified.

23320 After the semaphore named *name* has been created by *sem_open()* with the
 23321 **O_CREAT** flag, other processes can connect to the semaphore by calling
 23322 *sem_open()* with the same value of *name*.

23323 **O_EXCL** If **O_EXCL** and **O_CREAT** are set, *sem_open()* fails if the semaphore *name*
 23324 exists. The check for the existence of the semaphore and the creation of the
 23325 semaphore if it does not exist are atomic with respect to other processes
 23326 executing *sem_open()* with **O_EXCL** and **O_CREAT** set. If **O_EXCL** is set and
 23327 **O_CREAT** is not set, the effect is undefined.

23328 If flags other than **O_CREAT** and **O_EXCL** are specified in the *oflag* parameter,
 23329 the effect is unspecified.

23330 The *name* argument points to a string naming a semaphore object. It is unspecified whether the
 23331 name appears in the file system and is visible to functions that take pathnames as arguments.
 23332 The *name* argument conforms to the construction rules for a pathname. If *name* begins with the
 23333 slash character, then processes calling *sem_open()* with the same value of *name* will refer to the
 23334 same semaphore object, as long as that name has not been removed. If *name* does not begin with
 23335 the slash character, the effect is implementation-dependent. The interpretation of slash
 23336 characters other than the leading slash character in *name* is implementation-dependent.

23337 If a process makes multiple successful calls to *sem_open()* with the same value for *name*, the
 23338 same semaphore address is returned for each such successful call, provided that there have been

23339	no calls to <i>sem_unlink()</i> for this semaphore.	
23340	References to copies of the semaphore produce undefined results.	
23341	RETURN VALUE	
23342	Upon successful completion, the function returns the address of the semaphore. Otherwise, it	
23343	will return a value of SEM_FAILED and set <i>errno</i> to indicate the error. The symbol SEM_FAILED	
23344	is defined in the header <semaphore.h>. No successful return from <i>sem_open()</i> will return the	
23345	value SEM_FAILED.	
23346	ERRORS	
23347	If any of the following conditions occur, the <i>sem_open()</i> function will return SEM_FAILED and	
23348	set <i>errno</i> to the corresponding value:	
23349	[EACCES]	The named semaphore exists and the permissions specified by <i>oflag</i> are
23350		denied, or the named semaphore does not exist and permission to create the
23351		named semaphore is denied.
23352	[EEXIST]	O_CREAT and O_EXCL are set and the named semaphore already exists.
23353	[EINTR]	The <i>sem_open()</i> operation was interrupted by a signal.
23354	[EINVAL]	The <i>sem_open()</i> operation is not supported for the given name, or O_CREAT
23355		was specified in <i>oflag</i> and <i>value</i> was greater than SEM_VALUE_MAX.
23356	[EMFILE]	Too many semaphore descriptors or file descriptors are currently in use by
23357		this process.
23358	[ENAMETOOLONG]	
23359		The length of the <i>name</i> string exceeds PATH_MAX, or a pathname component
23360		is longer than NAME_MAX while _POSIX_NO_TRUNC is in effect.
23361	[ENFILE]	Too many semaphores are currently open in the system.
23362	[ENOENT]	O_CREAT is not set and the named semaphore does not exist.
23363	[ENOSPC]	There is insufficient space for the creation of the new named semaphore.
23364	[ENOSYS]	The function <i>sem_open()</i> is not supported by this implementation.
23365	EXAMPLES	
23366	None.	
23367	APPLICATION USAGE	
23368	None.	
23369	FUTURE DIRECTIONS	
23370	None.	
23371	SEE ALSO	
23372	<i>semctl()</i> , <i>semget()</i> , <i>semop()</i> , <i>sem_close()</i> , <i>sem_post()</i> , <i>sem_trywait()</i> , <i>sem_unlink()</i> , <i>sem_wait()</i> ,	
23373	<semaphore.h>.	
23374	CHANGE HISTORY	
23375	First released in Issue 5.	
23376	Included for alignment with the POSIX Realtime Extension.	

23377 **NAME**23378 sem_post — unlock a semaphore (**REALTIME**)23379 **SYNOPSIS**23380 RT

```
#include <semaphore.h>
```

23381

```
int sem_post(sem_t *sem);
```

23382

23383 **DESCRIPTION**23384 The *sem_post()* function unlocks the semaphore referenced by *sem* by performing a semaphore
23385 unlock operation on that semaphore.23386 If the semaphore value resulting from this operation is positive, then no threads were blocked
23387 waiting for the semaphore to become unlocked; the semaphore value is simply incremented.23388 If the value of the semaphore resulting from this operation is zero, then one of the threads
23389 blocked waiting for the semaphore will be allowed to return successfully from its call to
23390 *sem_wait()*. If the symbol `_POSIX_PRIORITY_SCHEDULING` is defined, the thread to be
23391 unblocked will be chosen in a manner appropriate to the scheduling policies and parameters in
23392 effect for the blocked threads. In the case of the schedulers `SCHED_FIFO` and `SCHED_RR`, the
23393 highest priority waiting thread will be unblocked, and if there is more than one highest priority
23394 thread blocked waiting for the semaphore, then the highest priority thread that has been waiting
23395 the longest will be unblocked. If the symbol `_POSIX_PRIORITY_SCHEDULING` is not defined,
23396 the choice of a thread to unblock is unspecified.23397 The *sem_post()* interface is reentrant with respect to signals and may be invoked from a signal-
23398 catching function.23399 **RETURN VALUE**23400 If successful, the *sem_post()* function returns zero; otherwise the function returns `-1` and sets
23401 *errno* to indicate the error.23402 **ERRORS**23403 The *sem_post()* function will fail if:23404 `[EINVAL]` The *sem* does not refer to a valid semaphore.23405 `[ENOSYS]` The function *sem_post()* is not supported by this implementation.23406 **EXAMPLES**

23407 None.

23408 **APPLICATION USAGE**

23409 None.

23410 **FUTURE DIRECTIONS**

23411 None.

23412 **SEE ALSO**23413 *semctl()*, *semget()*, *semop()*, *sem_trywait()*, *sem_wait()*, `<semaphore.h>`.23414 **CHANGE HISTORY**

23415 First released in Issue 5.

23416 Included for alignment with the POSIX Realtime Extension.

23417 **NAME**23418 sem_unlink — remove a named semaphore (**REALTIME**)23419 **SYNOPSIS**23420 RT `#include <semaphore.h>`23421 `int sem_unlink(const char *name);`

23422

23423 **DESCRIPTION**

23424 The *sem_unlink()* function removes the semaphore named by the string *name*. If the semaphore
 23425 named by *name* is currently referenced by other processes, then *sem_unlink()* has no effect on the
 23426 state of the semaphore. If one or more processes have the semaphore open when *sem_unlink()* is
 23427 called, destruction of the semaphore is postponed until all references to the semaphore have
 23428 been destroyed by calls to *sem_close()*, *_exit()*, or *exec*. Calls to *sem_open()* to re-create or re-
 23429 connect to the semaphore refer to a new semaphore after *sem_unlink()* is called. The
 23430 *sem_unlink()* call does not block until all references have been destroyed; it returns immediately.

23431 **RETURN VALUE**

23432 Upon successful completion, the function returns a value of 0. Otherwise, the semaphore is not
 23433 changed and the function returns a value of -1 and sets *errno* to indicate the error.

23434 **ERRORS**23435 The *sem_unlink()* function will fail if:

23436 [EACCES] Permission is denied to unlink the named semaphore.

23437 [ENAMETOOLONG]

23438 The length of the *name* string exceeds {NAME_MAX} while
 23439 {POSIX_NO_TRUNC} is in effect.

23440 [ENOENT] The named semaphore does not exist.

23441 [ENOSYS] The function *sem_unlink()* is not supported by this implementation.23442 **EXAMPLES**

23443 None.

23444 **APPLICATION USAGE**

23445 None.

23446 **FUTURE DIRECTIONS**

23447 None.

23448 **SEE ALSO**23449 *semctl()*, *semget()*, *semop()*, *sem_close()*, *sem_open()*, <semaphore.h>.23450 **CHANGE HISTORY**

23451 First released in Issue 5.

23452 Included for alignment with the POSIX Realtime Extension.

23453 **NAME**23454 sem_wait, sem_trywait — lock a semaphore (**REALTIME**)23455 **SYNOPSIS**23456 RT

```
#include <semaphore.h>
```

23457

```
int sem_wait(sem_t *sem);
```

23458

```
int sem_trywait(sem_t *sem);
```

23459

23460 **DESCRIPTION**

23461 The *sem_wait()* function locks the semaphore referenced by *sem* by performing a semaphore lock
 23462 operation on that semaphore. If the semaphore value is currently zero, then the calling thread
 23463 will not return from the call to *sem_wait()* until it either locks the semaphore or the call is
 23464 interrupted by a signal. The *sem_trywait()* function locks the semaphore referenced by *sem* only
 23465 if the semaphore is currently not locked; that is, if the semaphore value is currently positive.
 23466 Otherwise, it does not lock the semaphore.

23467 Upon successful return, the state of the semaphore is locked and remains locked until the
 23468 *sem_post()* function is executed and returns successfully.

23469 The *sem_wait()* function is interruptible by the delivery of a signal.

23470 **RETURN VALUE**

23471 The *sem_wait()* and *sem_trywait()* functions return zero if the calling process successfully
 23472 performed the semaphore lock operation on the semaphore designated by *sem*. If the call was
 23473 unsuccessful, the state of the semaphore is unchanged, and the function returns a value of -1
 23474 and sets *errno* to indicate the error.

23475 **ERRORS**

23476 The *sem_wait()* and *sem_trywait()* functions will fail if:

23477 [EAGAIN] The semaphore was already locked, so it cannot be immediately locked by the
 23478 *sem_trywait()* operation (*sem_trywait()* only).

23479 [EINVAL] The *sem* argument does not refer to a valid semaphore.

23480 [ENOSYS] The functions *sem_wait()* and *sem_trywait()* are not supported by this
 23481 implementation.

23482 The *sem_wait()* and *sem_trywait()* functions may fail if:

23483 [EDEADLK] A deadlock condition was detected.

23484 [EINTR] A signal interrupted this function.

23485 **EXAMPLES**

23486 None.

23487 **APPLICATION USAGE**

23488 Realtime applications may encounter priority inversion when using semaphores. The problem
 23489 occurs when a high priority thread “locks” (that is, waits on) a semaphore that is about to be
 23490 “unlocked” (that is, posted) by a low priority thread, but the low priority thread is preempted
 23491 by a medium priority thread. This scenario leads to priority inversion; a high priority thread is
 23492 blocked by lower priority threads for an unlimited period of time. During system design,
 23493 realtime programmers must take into account the possibility of this kind of priority inversion.
 23494 They can deal with it in a number of ways, such as by having critical sections that are guarded
 23495 by semaphores execute at a high priority, so that a thread cannot be preempted while executing
 23496 in its critical section.

23497 **FUTURE DIRECTIONS**

23498 None.

23499 **SEE ALSO**23500 *semctl()*, *semget()*, *semop()*, *sem_post()*, <**semaphore.h**>.23501 **CHANGE HISTORY**

23502 First released in Issue 5.

23503 Included for alignment with the POSIX Realtime Extension.

23504 **NAME**

23505 semctl — semaphore control operations

23506 **SYNOPSIS**

23507 EX #include <sys/sem.h>

23508 int semctl(int *semid*, int *semnum*, int *cmd*, ...);

23509

23510 **DESCRIPTION**

23511 The *semctl()* function provides a variety of semaphore control operations as specified by *cmd*.
 23512 The fourth argument is optional and depends upon the operation requested. If required, it is of
 23513 type **union semun**, which the application program must explicitly declare:

```
23514       union semun {
23515               int val;
23516               struct semid_ds *buf;
23517               unsigned short *array;
23518       } arg;
```

23519 The following semaphore control operations as specified by *cmd* are executed with respect to the
 23520 semaphore specified by *semid* and *semnum*. The level of permission required for each operation
 23521 is shown with each command, see Section 2.6 on page 36. The symbolic names for the values of
 23522 *cmd* are defined by the <sys/sem.h> header:

23523 **GETVAL** Return the value of *semval*, see <sys/sem.h>. Requires read permission.
 23524 **SETVAL** Set the value of *semval* to *arg.val*, where *arg* is the value of the fourth argument
 23525 to *semctl()*. When this command is successfully executed, the *semadj* value
 23526 corresponding to the specified semaphore in all processes is cleared. Requires alter
 23527 permission, see Section 2.6 on page 36.

23528 **GETPID** Return the value of *sempid*. Requires read permission.

23529 **GETNCNT** Return the value of *semmcnt*. Requires read permission.

23530 **GETZCNT** Return the value of *semzcnt*. Requires read permission.

23531 The following values of *cmd* operate on each *semval* in the set of semaphores:

23532 **GETALL** Return the value of *semval* for each semaphore in the semaphore set and place
 23533 into the array pointed to by *arg.array*, where *arg* is the fourth argument to
 23534 *semctl()*. Requires read permission.

23535 **SETALL** Set the value of *semval* for each semaphore in the semaphore set according to
 23536 the array pointed to by *arg.array*, where *arg* is the fourth argument to *semctl()*.
 23537 When this command is successfully executed, the *semadj* values corresponding
 23538 to each specified semaphore in all processes are cleared. Requires alter
 23539 permission.

23540 The following values of *cmd* are also available:

23541 **IPC_STAT** Place the current value of each member of the **semid_ds** data structure
 23542 associated with *semid* into the structure pointed to by *arg.buf*, where *arg* is the
 23543 fourth argument to *semctl()*. The contents of this structure are defined in
 23544 <sys/sem.h>. Requires read permission.

23545	IPC_SET	Set the value of the following members of the semid_ds data structure associated with <i>semid</i> to the corresponding value found in the structure pointed to by <i>arg.buf</i> , where <i>arg</i> is the fourth argument to <i>semctl()</i> :
23546		
23547		
23548		sem_perm.uid
23549		sem_perm.gid
23550		sem_perm.mode
23551		The mode bits specified in Section 2.6.1 on page 36 are copied into the corresponding bits of the sem_perm.mode associated with <i>semid</i> . The stored values of any other bits are unspecified.
23552		
23553		
23554		This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of sem_perm.cuid or sem_perm.uid in the semid_ds data structure associated with <i>semid</i> .
23555		
23556		
23557		
23558	IPC_RMID	Remove the semaphore-identifier specified by <i>semid</i> from the system and destroy the set of semaphores and semid_ds data structure associated with it. This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of sem_perm.cuid or sem_perm.uid in the semid_ds data structure associated with <i>semid</i> .
23559		
23560		
23561		
23562		
23563		
23564	RETURN VALUE	
23565	If successful, the value returned by <i>semctl()</i> depends on <i>cmd</i> as follows:	
23566	GETVAL	The value of <i>semval</i> .
23567	GETPID	The value of <i>sempid</i> .
23568	GETNCNT	The value of <i>semmcnt</i> .
23569	GETZCNT	The value of <i>semzcnt</i> .
23570	All others	0.
23571	Otherwise, <i>semctl()</i> returns -1 and <i>errno</i> indicates the error.	
23572	ERRORS	
23573	The <i>semctl()</i> function will fail if:	
23574	[EACCES]	Operation permission is denied to the calling process, see Section 2.6 on page 36.
23575		
23576	[EINVAL]	The value of <i>semid</i> is not a valid semaphore identifier, or the value of <i>semnum</i> is less than 0 or greater than or equal to <i>sem_nsems</i> , or the value of <i>cmd</i> is not a valid command.
23577		
23578		
23579	[EPERM]	The argument <i>cmd</i> is equal to IPC_RMID or IPC_SET and the effective user ID of the calling process is not equal to that of a process with appropriate privileges and it is not equal to the value of sem_perm.cuid or sem_perm.uid in the data structure associated with <i>semid</i> .
23580		
23581		
23582		
23583	[ERANGE]	The argument <i>cmd</i> is equal to SETVAL or SETALL and the value to which <i>semval</i> is to be set is greater than the system-imposed maximum.
23584		
23585	EXAMPLES	
23586	None.	

23587 **APPLICATION USAGE**

23588 The fourth parameter in the SYNOPSIS section is now specified as ... in order to avoid a clash
 23589 with the ISO C standard when referring to the union *semun* (as defined in XPG3) and for
 23590 backward compatibility.

23591 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
 23592 Application developers who need to use IPC should design their applications so that modules
 23593 using the IPC routines described in Section 2.6 on page 36 can be easily modified to use the
 23594 alternative interfaces.

23595 **FUTURE DIRECTIONS**

23596 None.

23597 **SEE ALSO**

23598 *semget()*, *semop()*, *sem_close()*, *sem_destroy()*, *sem_getvalue()*, *sem_init()*, *sem_open()*, *sem_post()*,
 23599 *sem_unlink()*, *sem_wait()*, <sys/sem.h>, Section 2.6 on page 36.

23600 **CHANGE HISTORY**

23601 First released in Issue 2.

23602 Derived from Issue 2 of the SVID.

23603 **Issue 4**

23604 The following changes are incorporated in this issue:

- 23605 • The interface is no longer marked as OPTIONAL FUNCTIONALITY.
- 23606 • Inclusion of the <sys/types.h> and <sys/ipc.h> headers is removed from the SYNOPSIS
 23607 section.
- 23608 • The last argument is now defined by an ellipsis symbol. In previous issues it was defined as
 23609 a union of the various types required by settings of *cmd*. These are now defined individually
 23610 in each description of permitted *cmd* settings. The text of the description of SETALL in the
 23611 DESCRIPTION now refers to the fourth argument instead of *arg.buf*.
- 23612 • In the DESCRIPTION the type of the array is specified in the descriptions of GETALL and
 23613 SETALL.
- 23614 • The [ENOSYS] error is removed from the ERRORS section.
- 23615 • A FUTURE DIRECTIONS section is added warning application developers about migration
 23616 to IEEE 1003.4 interfaces for interprocess communication.

23617 **Issue 4, Version 2**

23618 The fourth argument to *semctl()*, formerly specified in APPLICATION USAGE, is moved to the
 23619 DESCRIPTION, and references to its elements are made more precise.

23620 **Issue 5**

23621 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
 23622 DIRECTIONS to the APPLICATION USAGE section.

23623 NAME

23624 semget — get set of semaphores

23625 SYNOPSIS

23626 EX

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

23627

```
int semget(key_t key, int nsems, int semflg);
```

23628

23629 DESCRIPTION

23630 The *semget()* function returns the semaphore identifier associated with *key*.23631 A semaphore identifier with its associated **semid_ds** data structure and its associated set of
23632 *nsems* semaphores, see <sys/sem.h>, are created for *key* if one of the following is true:

- 23633 • The argument *key* is equal to IPC_PRIVATE.
- 23634 • The argument *key* does not already have a semaphore identifier associated with it and
23635 (*semflg* & IPC_CREAT) is non-zero.

23636 Upon creation, the **semid_ds** data structure associated with the new semaphore identifier is
23637 initialised as follows:

- 23638 • In the operation permissions structure *sem_perm.cuid*, *sem_perm.uid*, *sem_perm.cgid* and
23639 *sem_perm.gid* are set equal to the effective user ID and effective group ID, respectively, of the
23640 calling process.
- 23641 • The low-order 9 bits of *sem_perm.mode* are set equal to the low-order 9 bits of *semflg*.
- 23642 • The variable *sem_nsems* is set equal to the value of *nsems*.
- 23643 • The variable *sem_otime* is set equal to 0 and *sem_ctime* is set equal to the current time.
- 23644 • The data structure associated with each semaphore in the set is not initialised. The *semctl()*
23645 function with the command SETVAL or SETALL can be used to initialise each semaphore.

23646 RETURN VALUE

23647 Upon successful completion, *semget()* returns a non-negative integer, namely a semaphore
23648 identifier; otherwise, it returns -1 and *errno* will be set to indicate the error.

23649 ERRORS

23650 The *semget()* function will fail if:

- | | | |
|-------|----------|---|
| 23651 | [EACCES] | A semaphore identifier exists for <i>key</i> , but operation permission as specified by the low-order 9 bits of <i>semflg</i> would not be granted. See Section 2.6 on page 36. |
| 23652 | | |
| 23653 | | |
| 23654 | [EEXIST] | A semaphore identifier exists for the argument <i>key</i> but ((<i>semflg</i> & IPC_CREAT) && (<i>semflg</i> & IPC_EXCL)) is non-zero. |
| 23655 | | |
| 23656 | [EINVAL] | The value of <i>nsems</i> is either less than or equal to 0 or greater than the system-imposed limit, or a semaphore identifier exists for the argument <i>key</i> , but the number of semaphores in the set associated with it is less than <i>nsems</i> and <i>nsems</i> is not equal to 0. |
| 23657 | | |
| 23658 | | |
| 23659 | | |
| 23660 | [ENOENT] | A semaphore identifier does not exist for the argument <i>key</i> and (<i>semflg</i> & IPC_CREAT) is equal to 0. |
| 23661 | | |
| 23662 | [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. |
| 23663 | | |

23664 **EXAMPLES**

23665 None.

23666 **APPLICATION USAGE**

23667 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
 23668 Application developers who need to use IPC should design their applications so that modules
 23669 using the IPC routines described in Section 2.6 on page 36 can be easily modified to use the
 23670 alternative interfaces.

23671 **FUTURE DIRECTIONS**

23672 None.

23673 **SEE ALSO**

23674 *semctl()*, *semop()*, *sem_close()*, *sem_destroy()*, *sem_getvalue()*, *sem_init()*, *sem_open()*, *sem_post()*,
 23675 *sem_unlink()*, *sem_wait()*, **<sys/sem.h>**, Section 2.6 on page 36.

23676 **CHANGE HISTORY**

23677 First released in Issue 2.

23678 Derived from Issue 2 of the SVID.

23679 **Issue 4**

23680 The following changes are incorporated in this issue:

- 23681 • The interface is no longer marked as OPTIONAL FUNCTIONALITY.
- 23682 • Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the SYNOPSIS
 23683 section.
- 23684 • The [ENOSYS] error is removed from the ERRORS section.
- 23685 • A FUTURE DIRECTIONS section is added warning application developers about migration
 23686 to IEEE 1003.4 interfaces for interprocess communication.

23687 **Issue 5**

23688 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
 23689 DIRECTIONS to a new APPLICATION USAGE section.

23690 NAME

23691 semop — semaphore operations

23692 SYNOPSIS

23693 EX

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

23694

```
int semop(int semid, struct sembuf *sops, size_t nsops);
```

23695

23696 DESCRIPTION

23697 The *semop()* function is used to perform atomically a user-defined array of semaphore
 23698 operations on the set of semaphores associated with the semaphore identifier specified by the
 23699 argument *semid*.

23700 The argument *sops* is a pointer to a user-defined array of semaphore operation structures. The
 23701 implementation will not modify elements of this array unless the application uses
 23702 implementation-dependent extensions.

23703 The argument *nsops* is the number of such structures in the array.

23704 Each structure, **sembuf**, includes the following members:

23705

23706

23707

23708

23709

Member Type	Member Name	Description
short	sem_num	semaphore number
short	sem_op	semaphore operation
short	sem_flg	operation flags

23710 Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore
 23711 specified by *semid* and *sem_num*.

23712 The variable *sem_op* specifies one of three semaphore operations:

23713 1. If *sem_op* is a negative integer and the calling process has alter permission, one of the
 23714 following will occur:

23715 • If *semval*, see <sys/sem.h>, is greater than or equal to the absolute value of *sem_op*, the
 23716 absolute value of *sem_op* is subtracted from *semval*. Also, if (*sem_flg* & SEM_UNDO) is
 23717 non-zero, the absolute value of *sem_op* is added to the calling process' *semadj* value for
 23718 the specified semaphore.

23719 • If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & IPC_NOWAIT) is non-
 23720 zero, *semop()* will return immediately.

23721 • If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & IPC_NOWAIT) is 0,
 23722 *semop()* will increment the *semncnt* associated with the specified semaphore and
 23723 suspend execution of the calling thread until one of the following conditions occurs:

23724 — The value of *semval* becomes greater than or equal to the absolute value of *sem_op*.
 23725 When this occurs, the value of *semncnt* associated with the specified semaphore is
 23726 decremented, the absolute value of *sem_op* is subtracted from *semval* and, if
 23727 (*sem_flg* & SEM_UNDO) is non-zero, the absolute value of *sem_op* is added to the
 23728 calling process' *semadj* value for the specified semaphore.

23729 — The *semid* for which the calling thread is awaiting action is removed from the
 23730 system. When this occurs, *errno* is set equal to [EIDRM] and -1 is returned.

- 23731 — The calling thread receives a signal that is to be caught. When this occurs, the value
23732 of *semncnt* associated with the specified semaphore is decremented, and the calling
23733 thread resumes execution in the manner prescribed in *sigaction()*.
- 23734 2. If *sem_op* is a positive integer and the calling process has alter permission, the value of
23735 *sem_op* is added to *semval* and, if (*sem_flg* & SEM_UNDO) is non-zero, the value of *sem_op* is
23736 subtracted from the calling process' *semadj* value for the specified semaphore.
- 23737 3. If *sem_op* is 0 and the calling process has read permission, one of the following will occur:
- 23738 • If *semval* is 0, *semop()* will return immediately.
- 23739 • If *semval* is non-zero and (*sem_flg* & IPC_NOWAIT) is non-zero, *semop()* will return
23740 immediately.
- 23741 • If *semval* is non-zero and (*sem_flg* & IPC_NOWAIT) is 0, *semop()* will increment the
23742 *semzcnt* associated with the specified semaphore and suspend execution of the calling
23743 thread until one of the following occurs:
- 23744 — The value of *semval* becomes 0, at which time the value of *semzcnt* associated with
23745 the specified semaphore is decremented.
- 23746 — The *semid* for which the calling thread is awaiting action is removed from the
23747 system. When this occurs, *errno* is set equal to [EIDRM] and -1 is returned.
- 23748 — The calling thread receives a signal that is to be caught. When this occurs, the value
23749 of *semzcnt* associated with the specified semaphore is decremented, and the calling
23750 thread resumes execution in the manner prescribed in *sigaction()*.

23751 Upon successful completion, the value of *sempid* for each semaphore specified in the array
23752 pointed to by *sops* is set equal to the process ID of the calling process.

23753 RETURN VALUE

23754 Upon successful completion, *semop()* returns 0. Otherwise, it returns -1 and *errno* will be set to
23755 indicate the error.

23756 ERRORS

23757 The *semop()* function will fail if:

- 23758 [E2BIG] The value of *nsops* is greater than the system-imposed maximum.
- 23759 [EACCES] Operation permission is denied to the calling process, see Section 2.6 on page
23760 36.
- 23761 [EAGAIN] The operation would result in suspension of the calling process but
23762 (*sem_flg* & IPC_NOWAIT) is non-zero.
- 23763 [EFBIG] The value of *sem_num* is less than 0 or greater than or equal to the number of
23764 semaphores in the set associated with *semid*.
- 23765 [EIDRM] The semaphore identifier *semid* is removed from the system.
- 23766 [EINTR] The *semop()* function was interrupted by a signal.
- 23767 [EINVAL] The value of *semid* is not a valid semaphore identifier, or the number of
23768 individual semaphores for which the calling process requests a SEM_UNDO
23769 would exceed the system-imposed limit.
- 23770 [ENOSPC] The limit on the number of individual processes requesting a SEM_UNDO
23771 would be exceeded.

23772 [ERANGE] An operation would cause a *semval* to overflow the system-imposed limit, or
 23773 an operation would cause a *semadj* value to overflow the system-imposed
 23774 limit.

23775 EXAMPLES

23776 None.

23777 APPLICATION USAGE

23778 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
 23779 Application developers who need to use IPC should design their applications so that modules
 23780 using the IPC routines described in Section 2.6 on page 36 can be easily modified to use the
 23781 alternative interfaces.

23782 FUTURE DIRECTIONS

23783 None.

23784 SEE ALSO

23785 *exec*, *exit()*, *fork()*, *semctl()*, *semget()*, *sem_close()*, *sem_destroy()*, *sem_getvalue()*, *sem_init()*,
 23786 *sem_open()*, *sem_post()*, *sem_unlink()*, *sem_wait()*, **<sys/ipc.h>**, **<sys/sem.h>**, **<sys/types.h>**,
 23787 Section 2.6 on page 36.

23788 CHANGE HISTORY

23789 First released in Issue 2.

23790 Derived from Issue 2 of the SVID.

23791 Issue 4

23792 The following changes are incorporated in this issue:

- 23793 • The interface is no longer marked as OPTIONAL FUNCTIONALITY.
- 23794 • Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the SYNOPSIS
 23795 section.
- 23796 • The type of *nsops* is changed to **size_t**.
- 23797 • The DESCRIPTION is updated to indicate that an implementation will not modify the
 23798 elements of *sops* unless the application uses implementation-dependent extensions.
- 23799 • The [ENOSYS] error is removed from the ERRORS section.
- 23800 • A FUTURE DIRECTIONS section is added warning application developers about migration
 23801 to IEEE 1003.4 interfaces for interprocess communication.

23802 Issue 5

23803 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
 23804 DIRECTIONS to a new APPLICATION USAGE section.

23805 **NAME**

23806 setbuf — assign buffering to a stream

23807 **SYNOPSIS**

23808 #include <stdio.h>

23809 void setbuf(FILE **stream*, char **buf*);23810 **DESCRIPTION**

23811 Except that it returns no value, the function call:

23812 setbuf(*stream*, *buf*)

23813 is equivalent to:

23814 setvbuf(*stream*, *buf*, _IOFBF, BUFSIZ)23815 if *buf* is not a null pointer, or to:23816 setvbuf(*stream*, *buf*, _IONBF, BUFSIZ)23817 if *buf* is a null pointer.23818 **RETURN VALUE**23819 The *setbuf()* function returns no value.23820 **ERRORS**

23821 No errors are defined.

23822 **EXAMPLES**

23823 None.

23824 **APPLICATION USAGE**23825 A common source of error is allocating buffer space as an “automatic” variable in a code block,
23826 and then failing to close the stream in the same block.23827 With *setbuf()*, allocating a buffer of BUFSIZ bytes does not necessarily imply that all of BUFSIZ
23828 bytes are used for the buffer area.23829 **FUTURE DIRECTIONS**

23830 None.

23831 **SEE ALSO**23832 *fopen()*, *setvbuf()*, <stdio.h>.23833 **CHANGE HISTORY**

23834 First released in Issue 1.

23835 Derived from Issue 1 of the SVID.

23836 **NAME**

23837 setcontext — set current user context

23838 **SYNOPSIS**

23839 EX #include <ucontext.h>

23840 int setcontext(const ucontext_t *ucp);

23841

23842 **DESCRIPTION**23843 Refer to *getcontext()*.23844 **CHANGE HISTORY**

23845 First released in Issue 4, Version 2.

23846 **Issue 5**

23847 Moved from X/OPEN UNIX extension to BASE.

23848 **NAME**

23849 setgid — set-group-ID

23850 **SYNOPSIS**

23851 OH #include <sys/types.h>

23852 #include <unistd.h>

23853 int setgid(gid_t gid);

23854 **DESCRIPTION**23855 FIPS If the process has appropriate privileges, *setgid()* sets the real group ID, effective group ID and the saved set-group-ID to *gid*.23857 FIPS If the process does not have appropriate privileges, but *gid* is equal to the real group ID or the saved set-group-ID, *setgid()* function sets the effective group ID to *gid*; the real group ID and saved set-group-ID remain unchanged.

23860 Any supplementary group IDs of the calling process remain unchanged.

23861 **RETURN VALUE**23862 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.23864 **ERRORS**23865 The *setgid()* function will fail if:23866 [EINVAL] The value of the *gid* argument is invalid and is not supported by the implementation.23868 [EPERM] The process does not have appropriate privileges and *gid* does not match the real group ID or the saved set-group-ID.

23869 FIPS

23870 **EXAMPLES**

23871 None.

23872 **APPLICATION USAGE**

23873 None.

23874 **FUTURE DIRECTIONS**

23875 None.

23876 **SEE ALSO**23877 *exec*, *getgid()*, *setuid()*, <sys/types.h>, <unistd.h>.23878 **CHANGE HISTORY**

23879 First released in Issue 1.

23880 Derived from Issue 1 of the SVID.

23881 **Issue 4**

23882 The following change is incorporated for alignment with the FIPS requirements:

- 23883 • All references to the saved set-user-ID are marked as extensions. This is because Issue 4 defines this mechanism as mandatory, whereas the ISO POSIX-1 standard defines that it is only supported if {POSIX_SAVED_IDS} is set.

23886 Another change is incorporated as follows:

- 23887 • The <sys/types.h> header is now marked as optional (OH); this header need not be included on XSI-conformant systems.

23889 **NAME**

23890 setgrent — reset group database to first entry

23891 **SYNOPSIS**

23892 EX #include <grp.h>

23893 void setgrent(void);

23894

23895 **DESCRIPTION**23896 Refer to *endgrent()*.23897 **CHANGE HISTORY**

23898 First released in Issue 4, Version 2.

23899 **Issue 5**

23900 Moved from X/OPEN UNIX extension to BASE.

23901 **NAME**

23902 setitimer — set value of interval timer

23903 **SYNOPSIS**

23904 EX #include <sys/time.h>

23905 int setitimer(int *which*, const struct itimerval **value*,
23906 struct itimerval **ovalue*);

23907

23908 **DESCRIPTION**23909 Refer to *getitimer()*.23910 **CHANGE HISTORY**

23911 First released in Issue 4, Version 2.

23912 **Issue 5**

23913 Moved from X/OPEN UNIX extension to BASE.

23914 **NAME**

23915 _setjmp — set jump point for a non-local goto

23916 **SYNOPSIS**

23917 EX #include <setjmp.h>

23918 int _setjmp(jmp_buf env);

23919

23920 **DESCRIPTION**

23921 Refer to *_longjmp()*.

23922 **CHANGE HISTORY**

23923 First released in Issue 4, Version 2.

23924 **Issue 5**

23925 Moved from X/OPEN UNIX extension to BASE.

23926 **NAME**

23927 setjmp — set jump point for a non-local goto

23928 **SYNOPSIS**

23929 #include <setjmp.h>

23930 int setjmp(jmp_buf env);

23931 **DESCRIPTION**23932 A call to *setjmp()*, saves the calling environment in its *env* argument for later use by *longjmp()*.

23933 It is unspecified whether *setjmp()* 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 *setjmp* the behaviour is undefined.

23936 All accessible objects have values as of the time *longjmp()* was called, except that the values of objects of automatic storage duration which are local to the function containing the invocation of the corresponding *setjmp()* which do not have volatile-qualified type and which are changed between the *setjmp()* invocation and *longjmp()* call are indeterminate.

23940 An invocation of *setjmp()* must appear in one of the following contexts only:

- 23941 • the entire controlling expression of a selection or iteration statement
- 23942 • 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
- 23943 • the operand of a unary "!" operator with the resulting expression being the entire controlling expression of a selection or iteration
- 23946 • the entire expression of an expression statement (possibly cast to **void**).

23948 **RETURN VALUE**

23949 If the return is from a direct invocation, *setjmp()* returns 0. If the return is from a call to *longjmp()*, *setjmp()* returns a non-zero value.

23951 **ERRORS**

23952 No errors are defined.

23953 **EXAMPLES**

23954 None.

23955 **APPLICATION USAGE**

23956 In general, *sigsetjmp()* is more useful in dealing with errors and interrupts encountered in a low-level subroutine of a program.

23958 **FUTURE DIRECTIONS**

23959 None.

23960 **SEE ALSO**23961 *longjmp()*, *sigsetjmp()*, <setjmp.h>.23962 **CHANGE HISTORY**

23963 First released in Issue 1.

23964 Derived from Issue 1 of the SVID.

23965 **Issue 4**

23966 The following changes are incorporated in this issue:

- 23967 • This issue states that *setjmp()* is a macro or a function; previous issues stated that it was a
23968 macro. Warnings have also been added about the suppression of a *setjmp()* macro definition.
- 23969 • Text describing the accessibility of objects after a *longjmp()* call is added to the |
23970 DESCRIPTION. This text is imported from the entry for *longjmp()*.
- 23971 • Text describing the contexts in which calls to *setjmp()* are valid is moved to the |
23972 DESCRIPTION from the APPLICATION USAGE section.
- 23973 • The APPLICATION USAGE section is changed to refer to *sigsetjmp()*. |

23974 **NAME**23975 setkey — set encoding key (**CRYPT**)23976 **SYNOPSIS**

23977 EX #include <stdlib.h>

23978 void setkey(const char *key);

23979

23980 **DESCRIPTION**

23981 The *setkey()* function provides (rather primitive) access to an implementation-dependent
 23982 encoding algorithm. The argument of *setkey()* is an array of length 64 bytes containing only the
 23983 bytes with numerical value of 0 and 1. If this string is divided into groups of 8, the low-order bit
 23984 in each group is ignored; this gives a 56-bit key which is used by the algorithm. This is the key
 23985 that will be used with the algorithm to encode a string *block* passed to *encrypt()*.

23986 The *setkey()* function will not change the setting of **errno** if successful.

23987 This interface need not be reentrant.

23988 **RETURN VALUE**

23989 No values are returned.

23990 **ERRORS**23991 The *setkey()* function will fail if:

23992 [ENOSYS] The functionality is not supported on this implementation.

23993 **EXAMPLES**

23994 None.

23995 **APPLICATION USAGE**

23996 Decoding need not be implemented in all environments. This is related to U.S. Government
 23997 restrictions on encryption and decryption routines: the DES decryption algorithm cannot be
 23998 exported outside the U.S.A. Historical practice has been to ship a different version of the
 23999 encryption library without the decryption feature in the routines supplied. Thus the exported
 24000 version of *encrypt()* does encoding but not decoding.

24001 **FUTURE DIRECTIONS**

24002 None.

24003 **SEE ALSO**24004 *crypt()*, *encrypt()*, <stdlib.h>.24005 **CHANGE HISTORY**

24006 First released in Issue 1.

24007 Derived from Issue 1 of the SVID.

24008 **Issue 4**

24009 The following changes are incorporated in this issue:

- 24010 • The type of argument *key* is changed from **char *** to **const char ***.
- 24011 • The description of the array is put in terms of bytes instead of characters.
- 24012 • The APPLICATION USAGE section is added.

24013 **Issue 5**

24014 The DESCRIPTION is updated to indicate that **errno** will not be changed if the function is
 24015 successful.

24016 NAME

24017 setlocale — set program locale

24018 SYNOPSIS

24019 #include <locale.h>

24020 char *setlocale(int *category*, const char **locale*);

24021 DESCRIPTION

24022 The *setlocale()* function selects the appropriate piece of the program's locale, as specified by the
 24023 *category* and *locale* arguments, and may be used to change or query the program's entire locale or
 24024 portions thereof. The value LC_ALL for *category* names the program's entire locale; other values
 24025 for *category* name only a part of the program's locale:

24026 LC_COLLATE Affects the behaviour of regular expressions and the collation functions.

24027 LC_CTYPE Affects the behaviour of regular expressions, character classification, character
 24028 conversion functions and wide-character functions.

24029 LC_MESSAGES Affects what strings are expected by commands and utilities as affirmative or
 24030 EX negative responses, what strings are given by commands and utilities as
 24031 affirmative or negative responses, and the content of messages.

24032 LC_MONETARY Affects the behaviour of functions that handle monetary values.

24033 LC_NUMERIC Affects the radix character for the formatted input/output functions and the
 24034 string conversion functions.

24035 LC_TIME Affects the behaviour of the time conversion functions.

24036 The *locale* argument is a pointer to a character string containing the required setting of *category*.
 24037 The contents of this string are implementation-dependent. In addition, the following preset
 24038 values of *locale* are defined for all settings of *category*:

24039 "POSIX" Specifies the minimal environment for C-language translation called POSIX
 24040 locale. If *setlocale()* is not invoked, the POSIX locale is the default.

24041 "C" Same as POSIX.

24042 "" Specifies an implementation-dependent native environment. For XSI-
 24043 conformant systems, this corresponds to the value of the associated
 24044 environment variables, *LC_** and *LANG*; see the XBD specification, **Chapter 5,**
 24045 **Locale** and the XBD specification, **Chapter 6, Environment Variables.**

24046 A null pointer

24047 Used to direct *setlocale()* to query the current internationalised environment
 24048 and return the name of the *locale*().

24049 The locale state is common to all threads within a process.

24050 RETURN VALUE

24051 Upon successful completion, *setlocale()* returns the string associated with the specified category
 24052 for the new locale. Otherwise, *setlocale()* returns a null pointer and the program's locale is not
 24053 changed.

24054 A null pointer for *locale* causes *setlocale()* to return a pointer to the string associated with the
 24055 *category* for the program's current locale. The program's locale is not changed.

24056 The string returned by *setlocale()* is such that a subsequent call with that string and its associated
 24057 *category* will restore that part of the program's locale. The string returned must not be modified
 24058 by the program, but may be overwritten by a subsequent call to *setlocale()*.

24059 **ERRORS**

24060 No errors are defined.

24061 **EXAMPLES**

24062 None.

24063 **APPLICATION USAGE**

24064 The following code illustrates how a program can initialise the international environment for
 24065 one language, while selectively modifying the program's locale such that regular expressions
 24066 and string operations can be applied to text recorded in a different language:

```
24067     setlocale(LC_ALL, "De");
24068     setlocale(LC_COLLATE, "Fr@dict");
```

24069 Internationalised programs must call *setlocale()* to initiate a specific language operation. This
 24070 can be done by calling *setlocale()* as follows:

```
24071     setlocale(LC_ALL, " ");
```

24072 Changing the setting of LC_MESSAGES has no effect on catalogues that have already been
 24073 opened by calls to *catopen()*.

24074 **FUTURE DIRECTIONS**

24075 None.

24076 **SEE ALSO**

24077 *exec*, *isalnum()*, *isalpha()*, *iscntrl()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*,
 24078 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*, *iswspace()*,
 24079 *iswupper()*, *localeconv()*, *mblen()*, *mbstowcs()*, *mbtowc()*, *nl_langinfo()*, *printf()*, *scanf()*, *setlocale()*,
 24080 *strcoll()*, *strerror()*, *strfmon()*, *strtod()*, *strxfrm()*, *tolower()*, *toupper()*, *towlower()*, *towupper()*,
 24081 *wscoll()*, *wctod()*, *wcstombs()*, *wcsxfrm()*, *wctomb()*, **<langinfo.h>**, **<locale.h>**.

24082 **CHANGE HISTORY**

24083 First released in Issue 3.

24084 **Issue 4**

24085 The following changes are incorporated for alignment with the ISO C standard and the
 24086 ISO POSIX-1 standard:

- 24087 • The type of the argument *locale* is changed from **char *** to **const char ***.
- 24088 • The name POSIX is added to the list of standard locale names.

24089 The following change is incorporated for alignment with the ISO POSIX-2 standard:

- 24090 • The LC_MESSAGES value for *category* is added to the DESCRIPTION.

24091 Other changes are incorporated as follows:

- 24092 • The description of LC_MESSAGES is extended to indicate that this category also determines
 24093 what strings are produced by commands and utilities for affirmative and negative responses,
 24094 and that it affects the content of other program messages. This is marked as an extension.
- 24095 • References to *nl_langinfo()* are removed.
- 24096 • The description of the implementation-dependent native locale ("") is clarified by stating the
 24097 related environment variables explicitly.
- 24098 • The APPLICATION USAGE section is expanded.

24099 **Issue 5**

24100 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

|

24101 **NAME**

24102 setlogmask — set log priority mask

24103 **SYNOPSIS**24104 EX `#include <syslog.h>`24105 `int setlogmask(int maskpri);`

24106

24107 **DESCRIPTION**24108 Refer to *closelog()*.24109 **CHANGE HISTORY**

24110 First released in Issue 4, Version 2.

24111 **Issue 5**

24112 Moved from X/OPEN UNIX extension to BASE.

24113 **NAME**

24114 setpgid — set process group ID for job control

24115 **SYNOPSIS**

24116 OH #include <sys/types.h>

24117 #include <unistd.h>

24118 int setpgid(pid_t pid, pid_t pgid);

24119 **DESCRIPTION**

24120 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 will not change. Upon successful completion, the process group ID of the process with a process ID that matches *pid* will be set to *pgid*. As a special case, if *pid* is 0, the process ID of the calling process will be used. Also, if *pgid* is 0, the process group ID of the indicated process will be used.

24125 **RETURN VALUE**

24126 Upon successful completion, *setpgid()* returns 0. Otherwise -1 is returned and *errno* is set to indicate the error.

24128 **ERRORS**24129 The *setpgid()* function will fail if:

24130 [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.

24133 [EINVAL] The value of the *pgid* argument is less than 0, or is not a value supported by the implementation.

24135 [EPERM] The process indicated by the *pid* argument is a session leader.

24136 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.

24139 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.

24143 [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.

24145 **EXAMPLES**

24146 None.

24147 **APPLICATION USAGE**

24148 None.

24149 **FUTURE DIRECTIONS**

24150 None.

24151 **SEE ALSO**24152 *exec*, *getpgrp()*, *setsid()*, *tcsetpgrp()*, <sys/types.h>, <unistd.h>.24153 **CHANGE HISTORY**

24154 First released in Issue 3.

24155 Entry included for alignment with the POSIX.1-1988 standard.

24156 **Issue 4**

24157 The following changes are incorporated in this issue:

- 24158 • The interface is no longer marked as OPTIONAL FUNCTIONALITY.
- 24159 • The <**sys/types.h**> header is now marked as optional (OH); this header need not be included |
- 24160 on XSI-conformant systems.
- 24161 • The header <**unistd.h**> is added to the SYNOPSIS section. |
- 24162 • The DESCRIPTION in Issue 3 defined the behaviour of this function for implementations |
- 24163 that either supported or did not support job control. As job control is defined as mandatory |
- 24164 in Issue 4, only the former of these is now described.
- 24165 • The [ENOSYS] error is removed from the ERRORS section. |

24166 **NAME**

24167 setpgrp — set process group ID

24168 **SYNOPSIS**

24169 EX #include <unistd.h>

24170 pid_t setpgrp(void);

24171

24172 **DESCRIPTION**

24173 If the calling process is not already a session leader, *setpgrp()* sets the process group ID of the
24174 calling process to the process ID of the calling process. If *setpgrp()* creates a new session, then
24175 the new session has no controlling terminal.

24176 The *setpgrp()* function has no effect when the calling process is a session leader.

24177 **RETURN VALUE**24178 Upon completion, *setpgrp()* returns the process group ID.24179 **ERRORS**

24180 No errors are defined.

24181 **EXAMPLES**

24182 None.

24183 **APPLICATION USAGE**

24184 None.

24185 **FUTURE DIRECTIONS**

24186 None.

24187 **SEE ALSO**24188 *exec*, *fork()*, *getpid()*, *getsid()*, *kill()*, *setsid()*, <unistd.h>.24189 **CHANGE HISTORY**

24190 First released in Issue 4, Version 2.

24191 **Issue 5**

24192 Moved from X/OPEN UNIX extension to BASE.

24193 **NAME**

24194 setpriority — set the nice value

24195 **SYNOPSIS**

24196 EX #include <sys/resource.h>

24197 int setpriority(int *which*, id_t *who*, int *nice*);

24198

24199 **DESCRIPTION**24200 Refer to *getpriority()*.24201 **CHANGE HISTORY**

24202 First released in Issue 4, Version 2.

24203 **Issue 5**

24204 Moved from X/OPEN UNIX extension to BASE.

24205 Nice value added.

24206 **NAME**

24207 setpwent — user database function

24208 **SYNOPSIS**

24209 EX #include <pwd.h>

24210 void setpwent(void);

24211

24212 **DESCRIPTION**24213 Refer to *endpwent()*.24214 **CHANGE HISTORY**

24215 First released in Issue 4, Version 2.

24216 **Issue 5**

24217 Moved from X/OPEN UNIX extension to BASE.

24218 **NAME**

24219 setregid — set real and effective group IDs

24220 **SYNOPSIS**

24221 EX #include <unistd.h>

24222 int setregid(gid_t rgid, gid_t egid);

24223

24224 **DESCRIPTION**

24225 The *setregid()* function is used to set the real and effective group IDs of the calling process. If
 24226 *rgid* is *-1*, the real group ID is not changed; if *egid* is *-1*, the effective group ID is not changed.
 24227 The real and effective group IDs may be set to different values in the same call.

24228 Only a process with appropriate privileges can set the real group ID and the effective group ID
 24229 to any valid value.

24230 A non-privileged process can set either the real group ID to the saved set-group-ID from *exec*()*,
 24231 or the effective group ID to the saved set-group-ID or the real group ID.

24232 Any supplementary group IDs of the calling process remain unchanged.

24233 **RETURN VALUE**

24234 Upon successful completion, 0 is returned. Otherwise, *-1* is returned and *errno* is set to indicate
 24235 the error and neither of the group IDs will be changed.

24236 **ERRORS**24237 The *setregid()* function will fail if:

24238 [EINVAL] The value of the *rgid* or *egid* argument is invalid or out-of-range.

24239 [EPERM] The process does not have appropriate privileges and a change other than
 24240 changing the real group ID to the saved set-group-ID, or changing the
 24241 effective group ID to the real group ID or the saved group ID, was requested.

24242 **EXAMPLES**

24243 None.

24244 **APPLICATION USAGE**

24245 If a set-group-ID process sets its effective group ID to its real group ID, it can still set its effective
 24246 group ID back to the saved set-group-ID.

24247 **FUTURE DIRECTIONS**

24248 None.

24249 **SEE ALSO**24250 *exec*, *getuid()*, *setreuid()*, *setuid()*, <unistd.h>.24251 **CHANGE HISTORY**

24252 First released in Issue 4, Version 2.

24253 **Issue 5**

24254 Moved from X/OPEN UNIX extension to BASE.

24255 The DESCRIPTION is updated to indicate that the saved set-group-ID can be set by any of the
 24256 *exec*()* functions, not just *execev()*.

24257 **NAME**

24258 setreuid — set real and effective user IDs

24259 **SYNOPSIS**

24260 EX #include <unistd.h>

24261 int setreuid(uid_t ruid, uid_t euid);

24262

24263 **DESCRIPTION**

24264 The *setreuid()* function sets the real and effective user IDs of the current process to the values specified by the *ruid* and *euid* arguments. If *ruid* or *euid* is -1 , the corresponding effective or real user ID of the current process is left unchanged.

24267 A process with appropriate privileges can set either ID to any value. An unprivileged process can only set the effective user ID if the *euid* argument is equal to either the real, effective, or saved user ID of the process.

24270 It is unspecified whether a process without appropriate privileges is permitted to change the real user ID to match the current real, effective or saved user ID of the process.

24272 **RETURN VALUE**

24273 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

24275 **ERRORS**24276 The *setreuid()* function will fail if:

24277 [EINVAL] The value of the *ruid* or *euid* argument is invalid or out-of-range.

24278 [EPERM] The current process does not have appropriate privileges, and either an attempt was made to change the effective user ID to a value other than the real user ID or the saved set-user-ID or an attempt was made to change the real user ID to a value not permitted by the implementation.

24282 **EXAMPLES**

24283 None.

24284 **APPLICATION USAGE**

24285 None.

24286 **FUTURE DIRECTIONS**

24287 None.

24288 **SEE ALSO**24289 *getuid()*, *setuid()*, <unistd.h>.24290 **CHANGE HISTORY**

24291 First released in Issue 4, Version 2.

24292 **Issue 5**

24293 Moved from X/OPEN UNIX extension to BASE.

24294 **NAME**

24295 setrlimit — control maximum resource consumption

24296 **SYNOPSIS**

24297 EX #include <sys/resource.h>

24298 int setrlimit(int resource, const struct rlimit *rlp);

24299

24300 **DESCRIPTION**24301 Refer to *getrlimit()*.24302 **CHANGE HISTORY**

24303 First released in Issue 4, Version 2.

24304 **Issue 5**

24305 Moved from X/OPEN UNIX extension to BASE.

24306 **NAME**

24307 setsid — create session and set process group ID

24308 **SYNOPSIS**

24309 OH #include <sys/types.h>

24310 #include <unistd.h>

24311 pid_t setsid(void);

24312 **DESCRIPTION**

24313 The *setsid()* function creates a new session, if the calling process is not a process group leader.
 24314 Upon return the calling process will be the session leader of this new session, will be the process
 24315 group leader of a new process group, and will have no controlling terminal. The process group
 24316 ID of the calling process will be set equal to the process ID of the calling process. The calling
 24317 process will be the only process in the new process group and the only process in the new
 24318 session.

24319 **RETURN VALUE**

24320 Upon successful completion, *setsid()* returns the value of the process group ID of the calling
 24321 process. Otherwise it returns (**pid_t**)−1 and sets *errno* to indicate the error.

24322 **ERRORS**24323 The *setsid()* function will fail if:

24324 [EPERM] The calling process is already a process group leader, or the process group ID
 24325 of a process other than the calling process matches the process ID of the
 24326 calling process.

24327 **EXAMPLES**

24328 None.

24329 **APPLICATION USAGE**

24330 None.

24331 **FUTURE DIRECTIONS**

24332 None.

24333 **SEE ALSO**24334 *getsid()*, *setpgid()*, *setpgrp()*, <sys/types.h>, <unistd.h>.24335 **CHANGE HISTORY**

24336 First released in Issue 3.

24337 Entry included for alignment with the POSIX.1-1988 standard.

24338 **Issue 4**

24339 The following changes are incorporated in this issue:

- 24340 • The <sys/types.h> header is now marked as optional (OH); this header need not be included
- 24341 on XSI-conformant systems.
- 24342 • The header <unistd.h> is added to the SYNOPSIS section.
- 24343 • The argument list is explicitly defined as **void**.

24344 **NAME**

24345 setstate — switch pseudorandom number generator state arrays

24346 **SYNOPSIS**

24347 EX #include <stdlib.h>

24348 char *setstate(const char *state);

24349

24350 **DESCRIPTION**24351 Refer to *initstate()*.24352 **CHANGE HISTORY**

24353 First released in Issue 4, Version 2.

24354 **Issue 5**

24355 Moved from X/OPEN UNIX extension to BASE.

24356 **NAME**

24357 setuid — set-user-ID

24358 **SYNOPSIS**

24359 OH #include <sys/types.h>

24360 #include <unistd.h>

24361 int setuid(uid_t uid);

24362 **DESCRIPTION**24363 FIPS If the process has appropriate privileges, *setuid()* sets the real user ID, effective user ID, and the
24364 saved set-user-ID to *uid*.24365 FIPS If the process does not have appropriate privileges, but *uid* is equal to the real user ID or the
24366 FIPS saved set-user-ID, *setuid()* sets the effective user ID to *uid*; the real user ID and saved set-user-ID
24367 remain unchanged.24368 **RETURN VALUE**24369 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate
24370 the error.24371 **ERRORS**24372 The *setuid()* function will fail and return -1 and set *errno* to the corresponding value if one or
24373 more of the following are true:24374 [EINVAL] The value of the *uid* argument is invalid and not supported by the
24375 implementation.24376 [EPERM] The process does not have appropriate privileges and *uid* does not match the
24377 FIPS real user ID or the saved set-user-ID.24378 **EXAMPLES**

24379 None.

24380 **APPLICATION USAGE**

24381 None.

24382 **FUTURE DIRECTIONS**

24383 None.

24384 **SEE ALSO**24385 *exec*, *geteuid()*, *getuid()*, *setgid()*, <sys/types.h>, <unistd.h>.24386 **CHANGE HISTORY**

24387 First released in Issue 1.

24388 Derived from Issue 1 of the SVID.

24389 **Issue 4**

24390 The following change is incorporated for alignment with the FIPS requirements:

- 24391
- All references to the saved set-user-ID are marked as extensions. This is because Issue 4
24392 defines this mechanism as mandatory, whereas the ISO POSIX-1 standard defines that it is
24393 only supported if {POSIX_SAVED_IDS} is set.

24394 Other changes are incorporated as follows:

- 24395
- The <sys/types.h> header is now marked as optional (OH); this header need not be included
24396 on XSI-conformant systems.

- 24397
- The header <unistd.h> is added to the SYNOPSIS section.

24398 **NAME**

24399 setutxent — reset user accounting database to first entry

24400 **SYNOPSIS**

24401 EX #include <utmpx.h>

24402 void setutxent(void);

24403

24404 **DESCRIPTION**24405 Refer to *endutxent()*.24406 **CHANGE HISTORY**

24407 First released in Issue 4, Version 2.

24408 **Issue 5**

24409 Moved from X/OPEN UNIX extension to BASE.

24410 **NAME**

24411 setvbuf — assign buffering to a stream

24412 **SYNOPSIS**

24413 #include <stdio.h>

24414 int setvbuf(FILE **stream*, char **buf*, int *type*, size_t *size*);24415 **DESCRIPTION**

24416 The *setvbuf()* function may be used after the stream pointed to by *stream* is associated with an
 24417 open file but before any other operation is performed on the stream. The argument *type*
 24418 determines how *stream* will be buffered, as follows: *_IOFBF* causes input/output to be fully
 24419 buffered; *_IOLBF* causes input/output to be line buffered; *_IONBF* causes input/output to be
 24420 unbuffered. If *buf* is not a null pointer, the array it points to may be used instead of a buffer
 24421 allocated by *setvbuf()*. The argument *size* specifies the size of the array. The contents of the
 24422 array at any time are indeterminate.

24423 For information about streams, see Section 2.4 on page 30.

24424 **RETURN VALUE**

24425 Upon successful completion, *setvbuf()* returns 0. Otherwise, it returns a non-zero value if an
 24426 invalid value is given for *type* or if the request cannot be honoured.

24427 **ERRORS**24428 The *setvbuf()* function may fail if:24429 EX [EBADF] The file descriptor underlying *stream* is not valid.24430 **EXAMPLES**

24431 None.

24432 **APPLICATION USAGE**

24433 A common source of error is allocating buffer space as an “automatic” variable in a code block,
 24434 and then failing to close the stream in the same block.

24435 With *setvbuf()*, allocating a buffer of *size* bytes does not necessarily imply that all of *size* bytes are
 24436 used for the buffer area.

24437 Applications should note that many implementations only provide line buffering on input from
 24438 terminal devices.

24439 **FUTURE DIRECTIONS**

24440 None.

24441 **SEE ALSO**24442 *fopen()*, *setbuf()*, <stdio.h>.24443 **CHANGE HISTORY**

24444 First released in Issue 1.

24445 Derived from Issue 1 of the SVID.

24446 **Issue 4**

24447 The following change is incorporated for alignment with the ISO C standard:

- 24448 • This function is no longer marked as an extension.

24449 Other changes are incorporated as follows:

- 24450 • The second paragraph of the DESCRIPTION is now in Section 2.4 on page 30.

- 24451 • The [EBADF] error is marked as an extension.
- 24452 • The APPLICATION USAGE section is expanded.

|

24453 NAME

24454 shm_open — open a shared memory object (**REALTIME**)

24455 SYNOPSIS

24456 RT `#include <sys/mman.h>`24457 `int shm_open(const char *name, int oflag, mode_t mode);`

24458

24459 DESCRIPTION

24460 The *shm_open()* function establishes a connection between a shared memory object and a file
 24461 descriptor. It creates an open file description that refers to the shared memory object and a file
 24462 descriptor that refers to that open file description. The file descriptor is used by other functions
 24463 to refer to that shared memory object. The *name* argument points to a string naming a shared
 24464 memory object. It is unspecified whether the name appears in the file system and is visible to
 24465 other functions that take pathnames as arguments. The *name* argument conforms to the
 24466 construction rules for a pathname. If *name* begins with the slash character, then processes calling
 24467 *shm_open()* with the same value of *name* refer to the same shared memory object, as long as that
 24468 name has not been removed. If *name* does not begin with the slash character, the effect is
 24469 implementation-dependent. The interpretation of slash characters other than the leading slash
 24470 character in *name* is implementation-dependent.

24471 If successful, *shm_open()* returns a file descriptor for the shared memory object that is the lowest
 24472 numbered file descriptor not currently open for that process. The open file description is new,
 24473 and therefore the file descriptor does not share it with any other processes. It is unspecified
 24474 whether the file offset is set. The FD_CLOEXEC file descriptor flag associated with the new file
 24475 descriptor is set.

24476 The file status flags and file access modes of the open file description are according to the value
 24477 of *oflag*. The *oflag* argument is the bitwise inclusive OR of the following flags defined in the
 24478 header `<fcntl.h>`. Applications specify exactly one of the first two values (access modes) below
 24479 in the value of *oflag*:

24480 O_RDONLY Open for read access only.

24481 O_RDWR Open for read or write access.

24482 Any combination of the remaining flags may be specified in the value of *oflag*:

24483 O_CREAT If the shared memory object exists, this flag has no effect, except as noted
 24484 under O_EXCL below. Otherwise the shared memory object is created; the
 24485 user ID of the shared memory object will be set to the effective user ID of the
 24486 process; the group ID of the shared memory object will be set to a system
 24487 default group ID or to the effective group ID of the process. The permission
 24488 bits of the shared memory object will be set to the value of the *mode* argument
 24489 except those set in the file mode creation mask of the process. When bits in
 24490 *mode* other than the file permission bits are set, the effect is unspecified. The
 24491 *mode* argument does not affect whether the shared memory object is opened
 24492 for reading, for writing, or for both. The shared memory object has a size of
 24493 zero.

24494 O_EXCL If O_EXCL and O_CREAT are set, *shm_open()* fails if the shared memory
 24495 object exists. The check for the existence of the shared memory object and the
 24496 creation of the object if it does not exist is atomic with respect to other
 24497 processes executing *shm_open()* naming the same shared memory object with
 24498 O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the
 24499 result is undefined.

24500	O_TRUNC	If the shared memory object exists, and it is successfully opened O_RDWR, the object will be truncated to zero length and the mode and owner will be unchanged by this function call. The result of using O_TRUNC with O_RDONLY is undefined.
24501		
24502		
24503		
24504		When a shared memory object is created, the state of the shared memory object, including all data associated with the shared memory object, persists until the shared memory object is unlinked and all other references are gone. It is unspecified whether the name and shared memory object state remain valid after a system reboot.
24505		
24506		
24507		
24508	RETURN VALUE	
24509		Upon successful completion, the <i>shm_open()</i> function returns a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, it returns -1 and sets <i>errno</i> to indicate the error.
24510		
24511		
24512	ERRORS	
24513		The <i>shm_open()</i> function will fail if:
24514	[EACCES]	The shared memory object exists and the permissions specified by <i>oflag</i> are denied, or the shared memory object does not exist and permission to create the shared memory object is denied, or O_TRUNC is specified and write permission is denied.
24515		
24516		
24517		
24518	[EEXIST]	O_CREAT and O_EXCL are set and the named shared memory object already exists.
24519		
24520	[EINTR]	The <i>shm_open()</i> operation was interrupted by a signal.
24521	[EINVAL]	The <i>shm_open()</i> operation is not supported for the given name.
24522	[EMFILE]	Too many file descriptors are currently in use by this process.
24523	[ENAMETOOLONG]	
24524		The length of the <i>name</i> string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while _POSIX_NO_TRUNC is in effect.
24525		
24526		
24527	[ENFILE]	Too many shared memory objects are currently open in the system.
24528	[ENOENT]	O_CREAT is not set and the named shared memory object does not exist.
24529	[ENOSPC]	There is insufficient space for the creation of the new shared memory object.
24530	[ENOSYS]	The function <i>shm_open()</i> is not supported by this implementation.
24531	EXAMPLES	
24532		None.
24533	APPLICATION USAGE	
24534		None.
24535	FUTURE DIRECTIONS	
24536		None.
24537	SEE ALSO	
24538		<i>close()</i> , <i>dup()</i> , <i>exec</i> , <i>fcntl()</i> , <i>mmap()</i> , <i>shmat()</i> , <i>shmctl()</i> , <i>shmdt()</i> , <i>shm_unlink()</i> , <i>umask()</i> , <fcntl.h>, <sys/mman.h>.
24539		

24540 CHANGE HISTORY

24541 First released in Issue 5.

24542 Included for alignment with the POSIX Realtime Extension.

24543 **NAME**24544 `shm_unlink` — remove a shared memory object (**REALTIME**)24545 **SYNOPSIS**24546 RT `#include <sys/mman.h>`24547 `int shm_unlink(const char * name);`

24548

24549 **DESCRIPTION**

24550 The `shm_unlink()` function removes the name of the shared memory object named by the string
 24551 pointed to by *name*. If one or more references to the shared memory object exist when the object
 24552 is unlinked, the name is removed before `shm_unlink()` returns, but the removal of the memory
 24553 object contents is postponed until all open and map references to the shared memory object have
 24554 been removed.

24555 **RETURN VALUE**

24556 Upon successful completion, a value of zero is returned. Otherwise, a value of `-1` is returned
 24557 and *errno* will be set to indicate the error. If `-1` is returned, the named shared memory object will
 24558 not be changed by this function call.

24559 **ERRORS**24560 The `shm_unlink()` function will fail if:

24561 [EACCES] Permission is denied to unlink the named shared memory object.

24562 [ENAMETOOLONG]

24563 The length of the *name* string exceeds `{NAME_MAX}` while
 24564 `_POSIX_NO_TRUNC` is in effect.

24565 [ENOENT] The named shared memory object does not exist.

24566 [ENOSYS] The function `shm_unlink()` is not supported by this implementation.24567 **EXAMPLES**

24568 None.

24569 **APPLICATION USAGE**

24570 None.

24571 **FUTURE DIRECTIONS**

24572 None.

24573 **SEE ALSO**24574 `close()`, `mmap()`, `munmap()`, `shmat()`, `shmctl()`, `shmdt()`, `shm_open()`, `<sys/mman.h>`.24575 **CHANGE HISTORY**

24576 First released in Issue 5.

24577 Included for alignment with the POSIX Realtime Extension.

24578 **NAME**

24579 shmat — shared memory attach operation

24580 **SYNOPSIS**

24581 EX #include <sys/shm.h>

24582 void *shmat(int *shmid*, const void **shmaddr*, int *shmflg*);

24583

24584 **DESCRIPTION**

24585 The *shmat()* function attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the address space of the calling process. The segment is attached at the address specified by one of the following criteria:

- 24588 • If *shmaddr* is a null pointer, the segment is attached at the first available address as selected by the system.
- 24590 • If *shmaddr* is not a null pointer and (*shmflg* & SHM_RND) is non-zero, the segment is attached at the address given by (*shmaddr* - ((*ptrdiff_t*)*shmaddr* % SHMLBA)). The character % is the C-language remainder operator.
- 24593 • If *shmaddr* is not a null pointer and (*shmflg* & SHM_RND) is 0, the segment is attached at the address given by *shmaddr*.
- 24595 • The segment is attached for reading if (*shmflg* & SHM_RDONLY) is non-zero and the calling process has read permission; otherwise, if it is 0 and the calling process has read and write permission, the segment is attached for reading and writing.

24598 **RETURN VALUE**

24599 Upon successful completion, *shmat()* increments the value of *shm_nattch* in the data structure associated with the shared memory ID of the attached shared memory segment and returns the segment's start address.

24602 Otherwise, the shared memory segment is not attached, *shmat()* returns -1 and *errno* is set to indicate the error.

24604 **ERRORS**24605 The *shmat()* function will fail if:

- | | | |
|---|----------|--|
| 24606
24607 | [EACCES] | Operation permission is denied to the calling process, see Section 2.6 on page 36. |
| 24608
24609
24610
24611
24612 | [EINVAL] | The value of <i>shmid</i> is not a valid shared memory identifier; the <i>shmaddr</i> is not a null pointer and the value of (<i>shmaddr</i> - ((<i>ptrdiff_t</i>) <i>shmaddr</i> % SHMLBA)) is an illegal address for attaching shared memory; or the <i>shmaddr</i> is not a null pointer, (<i>shmflg</i> & SHM_RND) is 0 and the value of <i>shmaddr</i> is an illegal address for attaching shared memory. |
| 24613
24614 | [EMFILE] | The number of shared memory segments attached to the calling process would exceed the system-imposed limit. |
| 24615
24616 | [ENOMEM] | The available data space is not large enough to accommodate the shared memory segment. |

24617 **EXAMPLES**

24618 None.

24619 **APPLICATION USAGE**

24620 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.

24621 Application developers who need to use IPC should design their applications so that modules

24622 using the IPC routines described in Section 2.6 on page 36 can be easily modified to use the
 24623 alternative interfaces.

24624 **FUTURE DIRECTIONS**

24625 None.

24626 **SEE ALSO**

24627 *exec*, *exit()*, *fork()*, *shmctl()*, *shmdt()*, *shmget()*, *shm_open()*, *shm_unlink()*, **<sys/shm.h>**, Section
 24628 2.6 on page 36.

24629 **CHANGE HISTORY**

24630 First released in Issue 2.

24631 Derived from Issue 2 of the SVID.

24632 **Issue 4**

24633 The following changes are incorporated in this issue:

- 24634 • The interface is no longer marked as OPTIONAL FUNCTIONALITY.
- 24635 • Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the SYNOPSIS
 24636 section.
- 24637 • The type of argument *shmaddr* is changed from **char *** to **const void***.
- 24638 • The [ENOSYS] error is removed from the ERRORS section.
- 24639 • The DESCRIPTION is clarified in several places.
- 24640 • A FUTURE DIRECTIONS section is added warning application developers about migration
 24641 to IEEE 1003.4 interfaces for interprocess communication.

24642 **Issue 5**

24643 Moved from SHARED MEMORY to BASE.

24644 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
 24645 DIRECTIONS to a new APPLICATION USAGE section.

24646 **NAME**

24647 shmctl — shared memory control operations

24648 **SYNOPSIS**24649 EX `#include <sys/shm.h>`24650 `int shmctl(int shmid, int cmd, struct shm_ds *buf);`

24651

24652 **DESCRIPTION**24653 The *shmctl()* function provides a variety of shared memory control operations as specified by
24654 *cmd*. The following values for *cmd* are available:24655 **IPC_STAT** Place the current value of each member of the **shm_ds** data structure
24656 associated with *shmid* into the structure pointed to by *buf*. The contents of the
24657 structure are defined in `<sys/shm.h>`.24658 **IPC_SET** Set the value of the following members of the **shm_ds** data structure
24659 associated with *shmid* to the corresponding value found in the structure
24660 pointed to by *buf*:24661 `shm_perm.uid`
24662 `shm_perm.gid`
24663 `shm_perm.mode` low-order nine bits24664 **IPC_SET** can only be executed by a process that has an effective user ID equal
24665 to either that of a process with appropriate privileges or to the value of
24666 **shm_perm.cuid** or **shm_perm.uid** in the **shm_ds** data structure associated
24667 with *shmid*.24668 **IPC_RMID** Remove the shared memory identifier specified by *shmid* from the system and
24669 destroy the shared memory segment and **shm_ds** data structure associated
24670 with it. **IPC_RMID** can only be executed by a process that has an effective
24671 user ID equal to either that of a process with appropriate privileges or to the
24672 value of **shm_perm.cuid** or **shm_perm.uid** in the **shm_ds** data structure
24673 associated with *shmid*.24674 **RETURN VALUE**24675 Upon successful completion, *shmctl()* returns 0. Otherwise, it returns -1 and *errno* will be set to
24676 indicate the error.24677 **ERRORS**24678 The *shmctl()* function will fail if:24679 **[EACCES]** The argument *cmd* is equal to **IPC_STAT** and the calling process does not have
24680 read permission, see Section 2.6 on page 36.24681 **[EINVAL]** The value of *shmid* is not a valid shared memory identifier, or the value of *cmd*
24682 is not a valid command.24683 **[EPERM]** The argument *cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user ID
24684 of the calling process is not equal to that of a process with appropriate
24685 privileges and it is not equal to the value of **shm_perm.cuid** or **shm_perm.uid**
24686 in the data structure associated with *shmid*.24687 The *shmctl()* function may fail if:24688 EX **[EOVERFLOW]** The *cmd* argument is **IPC_STAT** and the **gid** or **uid** value is too large to be
24689 stored in the structure pointed to by the *buf* argument.

24690 **EXAMPLES**

24691 None.

24692 **APPLICATION USAGE**

24693 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
 24694 Application developers who need to use IPC should design their applications so that modules
 24695 using the IPC routines described in Section 2.6 on page 36 can be easily modified to use the
 24696 alternative interfaces.

24697 **FUTURE DIRECTIONS**

24698 None.

24699 **SEE ALSO**24700 *shmat()*, *shmdt()*, *shmget()*, *shm_open()*, *shm_unlink()*, *<sys/shm.h>*, Section 2.6 on page 36.24701 **CHANGE HISTORY**

24702 First released in Issue 2.

24703 Derived from Issue 2 of the SVID.

24704 **Issue 4**

24705 The following changes are incorporated in this issue:

- 24706 • The interface is no longer marked as OPTIONAL FUNCTIONALITY.
- 24707 • Inclusion of the *<sys/types.h>* and *<sys/ipc.h>* headers is removed from the SYNOPSIS
24708 section.
- 24709 • The [ENOSYS] error is removed from the ERRORS section.
- 24710 • A FUTURE DIRECTIONS section is added warning application developers about migration
24711 to IEEE 1003.4 interfaces for interprocess communication.

24712 **Issue 4, Version 2**

24713 The ERRORS section is updated for X/OPEN UNIX conformance to include [EOVERFLOW] as
 24714 an optional error.

24715 **Issue 5**

24716 Moved from SHARED MEMORY to BASE.

24717 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
 24718 DIRECTIONS to a new APPLICATION USAGE section.

24719 **NAME**

24720 shmdt — shared memory detach operation

24721 **SYNOPSIS**24722 EX `#include <sys/shm.h>`24723 `int shmdt(const void *shmaddr);`

24724

24725 **DESCRIPTION**

24726 The *shmdt()* function detaches the shared memory segment located at the address specified by
 24727 *shmaddr*. from the address space of the calling process.

24728 **RETURN VALUE**

24729 Upon successful completion, *shmdt()* will decrement the value of *shm_nattch* in the data
 24730 structure associated with the shared memory ID of the attached shared memory segment and
 24731 return 0.

24732 Otherwise, the shared memory segment will not be detached, *shmdt()* will return -1 and *errno*
 24733 will be set to indicate the error.

24734 **ERRORS**24735 The *shmdt()* function will fail if:

24736 [EINVAL] The value of *shmaddr* is not the data segment start address of a shared
 24737 memory segment.

24738 **EXAMPLES**

24739 None.

24740 **APPLICATION USAGE**

24741 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
 24742 Application developers who need to use IPC should design their applications so that modules
 24743 using the IPC routines described in Section 2.6 on page 36 can be easily modified to use the
 24744 alternative interfaces.

24745 **FUTURE DIRECTIONS**

24746 None.

24747 **SEE ALSO**

24748 *exec*, *exit()*, *fork()*, *shmat()*, *shmctl()*, *shmget()*, *shm_open()*, *shm_unlink()*, <sys/shm.h>, Section
 24749 2.6 on page 36.

24750 **CHANGE HISTORY**

24751 First released in Issue 2.

24752 Derived from Issue 2 of the SVID.

24753 **Issue 4**

24754 The following changes are incorporated in this issue:

- 24755 • The interface is no longer marked as OPTIONAL FUNCTIONALITY.
- 24756 • Inclusion of the <sys/types.h> and <sys/ipc.h> headers is removed from the SYNOPSIS
 24757 section.

24758	<ul style="list-style-type: none">• The type of argument <i>shmaddr</i> is changed from char * to const void*.	
24759	<ul style="list-style-type: none">• The DESCRIPTION is clarified in several places.	
24760	<ul style="list-style-type: none">• The [ENOSYS] error is removed from the ERRORS section.	
24761	<ul style="list-style-type: none">• A FUTURE DIRECTIONS section is added warning application developers about migration	
24762	to IEEE 1003.4 interfaces for interprocess communication.	
24763	Issue 5	
24764	Moved from SHARED MEMORY to BASE.	
24765	The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE	
24766	DIRECTIONS to a new APPLICATION USAGE section.	

24767 **NAME**

24768 shmget — get shared memory segment

24769 **SYNOPSIS**

24770 EX #include <sys/shm.h>

24771 int shmget(key_t key, size_t size, int shmflg);

24772

24773 **DESCRIPTION**24774 The *shmget()* function returns the shared memory identifier associated with *key*.24775 A shared memory identifier, associated data structure and shared memory segment of at least
24776 *size* bytes, see <sys/shm.h>, are created for *key* if one of the following is true:

- 24777 • The argument *key* is equal to IPC_PRIVATE.
- 24778 • The argument *key* does not already have a shared memory identifier associated with it and
24779 (*shmflg* & IPC_CREAT) is non-zero.

24780 Upon creation, the data structure associated with the new shared memory identifier is initialised
24781 as follows:

- 24782 • The values of *shm_perm.cuid*, *shm_perm.uid*, *shm_perm.cgid* and *shm_perm.gid* are set equal to
24783 the effective user ID and effective group ID, respectively, of the calling process.
- 24784 • The low-order nine bits of *shm_perm.mode* are set equal to the low-order nine bits of *shmflg*.
24785 The value of *shm_segsz* is set equal to the value of *size*.
- 24786 • The values of *shm_lpid*, *shm_nattch*, *shm_atime* and *shm_dtime* are set equal to 0.
- 24787 • The value of *shm_ctime* is set equal to the current time.

24788 When the shared memory segment is created, it will be initialised with all zero values.

24789 **RETURN VALUE**24790 Upon successful completion, *shmget()* returns a non-negative integer, namely a shared memory
24791 identifier; otherwise, it returns -1 and *errno* will be set to indicate the error.24792 **ERRORS**24793 The *shmget()* function will fail if:

- | | | |
|----------------------------------|----------|---|
| 24794
24795
24796 | [EACCES] | A shared memory identifier exists for <i>key</i> but operation permission as specified by the low-order nine bits of <i>shmflg</i> would not be granted. See Section 2.6 on page 36. |
| 24797
24798 | [EEXIST] | A shared memory identifier exists for the argument <i>key</i> but (<i>shmflg</i> & IPC_CREAT) && (<i>shmflg</i> & IPC_EXCL) is non-zero. |
| 24799
24800
24801
24802 | [EINVAL] | The value of <i>size</i> is less than the system-imposed minimum or greater than the system-imposed maximum, or a shared memory identifier exists for the argument <i>key</i> but the size of the segment associated with it is less than <i>size</i> and <i>size</i> is not 0. |
| 24803
24804 | [ENOENT] | A shared memory identifier does not exist for the argument <i>key</i> and (<i>shmflg</i> & IPC_CREAT) is 0. |
| 24805
24806
24807 | [ENOMEM] | A shared memory identifier and associated shared memory segment are to be created but the amount of available physical memory is not sufficient to fill the request. |

24808 [ENOSPC] A shared memory identifier is to be created but the system-imposed limit on
 24809 the maximum number of allowed shared memory identifiers system-wide
 24810 would be exceeded.

24811 EXAMPLES

24812 None.

24813 APPLICATION USAGE

24814 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
 24815 Application developers who need to use IPC should design their applications so that modules
 24816 using the IPC routines described in Section 2.6 on page 36 can be easily modified to use the
 24817 alternative interfaces.

24818 FUTURE DIRECTIONS

24819 None.

24820 SEE ALSO

24821 *shmat()*, *shmctl()*, *shmdt()*, *shm_open()*, *shm_unlink()*, **<sys/shm.h>**, Section 2.6 on page 36.

24822 CHANGE HISTORY

24823 First released in Issue 2.

24824 Derived from Issue 2 of the SVID.

24825 Issue 4

24826 The following changes are incorporated in this issue:

- 24827 • The interface is no longer marked as OPTIONAL FUNCTIONALITY.
- 24828 • Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the SYNOPSIS
 24829 section.
- 24830 • The [ENOSYS] error is removed from the ERRORS section.
- 24831 • A FUTURE DIRECTIONS section is added warning application developers about migration
 24832 to IEEE 1003.4 interfaces for interprocess communication.

24833 Issue 5

24834 Moved from SHARED MEMORY to BASE.

24835 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
 24836 DIRECTIONS to a new APPLICATION USAGE section.

24837 NAME

24838 sigaction — examine and change signal action

24839 SYNOPSIS

24840 #include <signal.h>

```
24841 int sigaction(int sig, const struct sigaction *act,
24842             struct sigaction *oact);
```

24843 DESCRIPTION

24844 The *sigaction()* function allows the calling process to examine and/or specify the action to be
 24845 associated with a specific signal. The argument *sig* specifies the signal; acceptable values are
 24846 defined in <signal.h>.

24847 The structure **sigaction**, used to describe an action to be taken, is defined in the header
 24848 <signal.h> to include at least the following members:

24849

24850

Member Type	Member Name	Description
void(*) (int) sigset_t	sa_handler sa_mask	SIG_DFL, SIG_IGN or pointer to a function. Additional set of signals to be blocked during execution of signal-catching function.
int void(*) (int, siginfo_t *, void *)	sa_flags sa_sigaction	Special flags to affect behaviour of signal. Signal-catching function.

24857

24858 If the argument *act* is not a null pointer, it points to a structure specifying the action to be
 24859 associated with the specified signal. If the argument *oact* is not a null pointer, the action
 24860 previously associated with the signal is stored in the location pointed to by the argument *oact*. If
 24861 the argument *act* is a null pointer, signal handling is unchanged; thus, the call can be used to
 24862 enquire about the current handling of a given signal. The *sa_handler* field of the **sigaction**
 24863 structure identifies the action to be associated with the specified signal. If the *sa_handler* field
 24864 specifies a signal-catching function, the *sa_mask* field identifies a set of signals that will be added
 24865 to the process' signal mask before the signal-catching function is invoked. The SIGKILL and
 24866 SIGSTOP signals will not be added to the signal mask using this mechanism; this restriction will
 24867 be enforced by the system without causing an error to be indicated.

24868 If the SA_SIGINFO flag (see below) is cleared in the *sa_flags* field of the **sigaction** structure, the
 24869 *sa_handler* field identifies the action to be associated with the specified signal. If the
 24870 SA_SIGINFO flag is set in the *sa_flags* field, the *sa_sigaction* field specifies a signal-catching
 24871 function. If the SA_SIGINFO bit is cleared and the *sa_handler* field specifies a signal-catching
 24872 function, or if the SA_SIGINFO bit is set, the *sa_mask* field identifies a set of signals that will be
 24873 added to the signal mask of the thread before the signal-catching function is invoked.

24874 The *sa_flags* field can be used to modify the behaviour of the specified signal.

24875 The following flags, defined in the header <signal.h>, can be set in *sa_flags*:

24876 SA_NOCLDSTOP Do not generate SIGCHLD when children stop.

24877 EX SA_ONSTACK If set and an alternate signal stack has been declared with *sigaltstack()* or
 24878 *sigstack()*, the signal will be delivered to the calling process on that stack.
 24879 Otherwise, the signal will be delivered on the current stack.

24880 SA_RESETHAND If set, the disposition of the signal will be reset to SIG_DFL and the
 24881 SA_SIGINFO flag will be cleared on entry to the signal handler.

24882		Note: SIGILL and SIGTRAP cannot be automatically reset when
24883		delivered; the system silently enforces this restriction.
24884		Otherwise, the disposition of the signal will not be modified on entry to
24885		the signal handler.
24886		In addition, if this flag is set, <i>sigaction()</i> behaves as if the SA_NODEFER
24887		flag were also set.
24888	SA_RESTART	This flag affects the behaviour of interruptible functions; that is, those
24889		specified to fail with <i>errno</i> set to [EINTR]. If set, and a function specified
24890		as interruptible is interrupted by this signal, the function will restart and
24891		will not fail with [EINTR] unless otherwise specified. If the flag is not set,
24892		interruptible functions interrupted by this signal will fail with <i>errno</i> set to
24893		[EINTR].
24894	SA_SIGINFO	If cleared and the signal is caught, the signal-catching function will be
24895		entered as:
24896		<pre>void func(int signo);</pre>
24897		where <i>signo</i> is the only argument to the signal catching function. In this
24898		case the sa_handler member must be used to describe the signal catching
24899		function and the application must not modify the sa_sigaction member.
24900		If SA_SIGINFO is set and the signal is caught, the signal-catching
24901		function will be entered as:
24902		<pre>void func(int signo, siginfo_t *info, void *context);</pre>
24903		where two additional arguments are passed to the signal catching
24904		function. The second argument will point to an object of type siginfo_t
24905		explaining the reason why the signal was generated; the third argument
24906		can be cast to a pointer to an object of type ucontext_t to refer to the
24907		receiving process' context that was interrupted when the signal was
24908		delivered. In this case the sa_sigaction member must be used to describe
24909		the signal catching function and the application must not modify the
24910		sa_handler member.
24911		The si_signo member contains the system-generated signal number.
24912		The si_errno member may contain implementation-dependent additional
24913		error information; if non-zero, it contains an error number identifying the
24914		condition that caused the signal to be generated.
24915		The si_code member contains a code identifying the cause of the signal.
24916		If the value of si_code is less than or equal to 0, then the signal was
24917		generated by a process and si_pid and si_uid respectively indicate the
24918		process ID and the real user ID of the sender. The < signal.h > header
24919		description contains information about the signal specific contents of the
24920		elements of the siginfo_t type.
24921	SA_NOCLDWAIT	If set, and <i>sig</i> equals SIGCHLD, child processes of the calling processes
24922		will not be transformed into zombie processes when they terminate. If
24923		the calling process subsequently waits for its children, and the process
24924		has no unwaited for children that were transformed into zombie
24925		processes, it will block until all of its children terminate, and <i>wait()</i> ,
24926		<i>wait3()</i> , <i>waitid()</i> and <i>waitpid()</i> will fail and set <i>errno</i> to [ECHILD].
24927		Otherwise, terminating child processes will be transformed into zombie

24928		processes, unless SIGCHLD is set to SIG_IGN.
24929 EX	SA_NODEFER	If set and <i>sig</i> is caught, <i>sig</i> will not be added to the process' signal mask on entry to the signal handler unless it is included in sa_mask . Otherwise, <i>sig</i> will always be added to the process' signal mask on entry to the signal handler.
24930		
24931		
24932		
24933		If <i>sig</i> is SIGCHLD and the SA_NOCLDSTOP flag is not set in <i>sa_flags</i> , and the implementation supports the SIGCHLD signal, then a SIGCHLD signal will be generated for the calling process whenever any of its child processes stop. If <i>sig</i> is SIGCHLD and the SA_NOCLDSTOP flag is set in <i>sa_flags</i> , then the implementation will not generate a SIGCHLD signal in this way.
24934		
24935		
24936		
24937		When a signal is caught by a signal-catching function installed by <i>sigaction()</i> , a new signal mask is calculated and installed for the duration of the signal-catching function (or until a call to either <i>sigprocmask()</i> or <i>sigsuspend()</i> is made). This mask is formed by taking the union of the current signal mask and the value of the <i>sa_mask</i> 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.
24938		
24939		
24940 EX		
24941		
24942		
24943		Once an action is installed for a specific signal, it remains installed until another action is explicitly requested (by another call to <i>sigaction()</i>), until the SA_RESETHAND flag causes resetting of the handler, or until one of the <i>exec</i> functions is called.
24944 EX		
24945		
24946		If the previous action for <i>sig</i> had been established by <i>signal()</i> , the values of the fields returned in the structure pointed to by <i>oact</i> are unspecified, and in particular <i>oact->sa_handler</i> is not necessarily the same value passed to <i>signal()</i> . However, if a pointer to the same structure or a copy thereof is passed to a subsequent call to <i>sigaction()</i> via the <i>act</i> argument, handling of the signal will be as if the original call to <i>signal()</i> were repeated.
24947		
24948		
24949		
24950		
24951		If <i>sigaction()</i> fails, no new signal handler is installed.
24952		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 <i>errno</i> set to [EINVAL].
24953		
24954		If SA_SIGINFO is not set in <i>sa_flags</i> , then the disposition of subsequent occurrences of <i>sig</i> when it is already pending is implementation-dependent; the signal-catching function will be invoked with a single argument. If the implementation supports the Realtime Signals Extension option, and if SA_SIGINFO is set in <i>sa_flags</i> , then subsequent occurrences of <i>sig</i> generated by <i>sigqueue()</i> or as a result of any signal-generating function that supports the specification of an application-defined value (when <i>sig</i> is already pending) will be queued in FIFO order until delivered or accepted; the signal-catching function will be invoked with three arguments. The application specified value is passed to the signal-catching function as the <i>si_value</i> member of the siginfo_t structure.
24955		
24956 RT		
24957		
24958		
24959		
24960		
24961		
24962		
24963		Signal Generation and Delivery
24964		A signal is said to be <i>generated</i> 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, signals generated via the sigevent structure and terminal activity, as well as invocations of <i>kill()</i> and <i>sigqueue()</i> functions. In some circumstances, the same event generates signals for multiple processes.
24965		
24966 RT		
24967 RT		
24968		
24969		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
24970		
24971		
24972		
24973		

24974	process.
24975	Each process has an action to be taken in response to each signal defined by the system (see Signal Actions on page 811). A signal is said to be <i>delivered</i> to a process when the appropriate action for the process and signal is taken. A signal is said to be <i>accepted</i> by a process when the signal is selected and returned by one of the <i>sigwait()</i> functions.
24976	
24977	
24978	
24979	During the time between the generation of a signal and its delivery or acceptance, the signal is said to be <i>pending</i> . Ordinarily, this interval cannot be detected by an application. However, a signal can be <i>blocked</i> 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 will remain pending until it is unblocked, it is accepted when it is selected and returned by a call to the <i>sigwait()</i> function, or the action associated with it is set to ignore the signal. Signals generated for the process will be delivered to exactly one of those threads within the process which is in a call to a <i>sigwait()</i> function selecting that signal or has not blocked delivery of the signal. If there are no threads in a call to a <i>sigwait()</i> function selecting that signal, and if all threads within the process block delivery of the signal, the signal will remain pending on the process until a thread calls a <i>sigwait()</i> 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.
24980	
24981	
24982	
24983	
24984	
24985	
24986	
24987	
24988	
24989	
24990	Each thread has a <i>signal mask</i> that defines the set of signals currently blocked from delivery to it. The signal mask for a thread is initialised 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 <i>fork()</i> . The <i>sigaction()</i> , <i>sigprocmask()</i> and <i>sigsuspend()</i> functions control the manipulation of the signal mask.
24991	
24992	
24993	
24994	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-dependent as to whether the signal is delivered or accepted more than once in circumstances other than those in which queueing 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.
24995	
24996	
24997	
24998	
24999	
25000	
25001	When any stop signal (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) is generated for a process, any pending SIGCONT signals for that process will be discarded. Conversely, when SIGCONT is generated for a process, all pending stop signals for that process will be discarded. When SIGCONT is generated for a process that is stopped, the process will be continued, even if the SIGCONT signal is blocked or ignored. If SIGCONT is blocked and not ignored, it will remain pending until it is either unblocked or a stop signal is generated for the process.
25002	
25003 RT	
25004	
25005	
25006	
25007	An implementation will document any condition not specified by this document under which the implementation generates signals.
25008	
25009	Some signal-generating functions, such as high-resolution timer expiration, asynchronous I/O completion, interprocess message arrival, and the <i>sigqueue()</i> 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 <code><signal.h></code> and contains at least the following members:
25010	
25011	
25012	
25013	
25014	
25015 RT	Some signal-generating functions, such as high-resolution timer expiration, asynchronous I/O completion, interprocess message arrival, and the <i>sigqueue()</i> 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 <code><signal.h></code> and contains at least the following members:
25016	
25017	
25018	
25019	

25020
25021
25022
25023
25024
25025
25026

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

25027 RT
25028

The *sigev_notify* member specifies the notification mechanism to use when an asynchronous event occurs. This document defines the following values for the *sigev_notify* member:

25029
25030

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

25031
25032
25033
25034
25035
25036
25037

SIGEV_SIGNAL The signal specified in *sigev_signo* will 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 will be queued to the process and the value specified in *sigev_value* will 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.

25038

SIGEV_THREAD A notification function will be called to perform notification.

25039

An implementation may define additional notification mechanisms.

25040
25041
25042

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.

25043

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

25044
25045
25046
25047

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

25048
25049

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.

25050

If the Realtime Signals Extension option is supported:

25051
25052
25053
25054
25055
25056

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 will be marked pending and, if the SA_SIGINFO flag is set for that signal, the signal will 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.

25057
25058
25059
25060

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.

25061 When multiple unblocked signals, all in the range SIGRTMIN to SIGRTMAX, are pending,
 25062 the behaviour will be as if the implementation delivers the pending unblocked signal with
 25063 the lowest signal number within that range. No other ordering of signal delivery is specified.

25064 If, when a pending signal is delivered, there are additional signals queued to that signal
 25065 number, the signal remains pending. Otherwise, the pending indication is reset.

25066 Multi-threaded programs can use an alternate event notification mechanism:

25067 When a notification is processed, and the *sigev_notify* member of the **sigevent** structure has
 25068 the value SIGEV_THREAD, the function *sigev_notify_function* is called with parameter
 25069 *sigev_value*.

25070 The function will be executed in an environment as if it were the *start_routine* for a newly
 25071 created thread with thread attributes specified by *sigev_notify_attributes*. If
 25072 *sigev_notify_attributes* is NULL, the behaviour will be as if the thread were created with the
 25073 *detachstate* attribute set to PTHREAD_CREATE_DETACHED. Supplying an attributes
 25074 structure with a *detachstate* attribute of PTHREAD_CREATE_JOINABLE results in undefined
 25075 behaviour. The signal mask of this thread is implementation-dependent.

25076 Signal Actions

25077 There are three types of action that can be associated with a signal: SIG_DFL, SIG_IGN or a
 25078 *pointer to a function*. Initially, all signals will be set to SIG_DFL or SIG_IGN prior to entry of the
 25079 *main()* routine (see the *exec* functions). The actions prescribed by these values are as follows:

25080 SIG_DFL — signal-specific default action

25081 • The default actions for the signals defined in this specification are specified under
 25082 RT **<signal.h>**. If the Realtime Signals Extension option is supported, the default actions for
 25083 the realtime signals in the range SIGRTMIN to SIGRTMAX are to terminate the process
 25084 abnormally.

25085 • If the default action is to stop the process, the execution of that process is temporarily
 25086 suspended. When a process stops, a SIGCHLD signal will be generated for its parent
 25087 process, unless the parent process has set the SA_NOCLDSTOP flag. While a process is
 25088 stopped, any additional signals that are sent to the process will not be delivered until the
 25089 process is continued, except SIGKILL which always terminates the receiving process. A
 25090 process that is a member of an orphaned process group will not be allowed to stop in
 25091 response to the SIGTSTP, SIGTTIN or SIGTTOU signals. In cases where delivery of one
 25092 of these signals would stop such a process, the signal will be discarded.

25093 • Setting a signal action to SIG_DFL for a signal that is pending, and whose default action
 25094 is to ignore the signal (for example, SIGCHLD), will cause the pending signal to be
 25095 RT discarded, whether or not it is blocked. If the Realtime Signals Extension option is
 25096 supported, any queued values pending will be discarded and the resources used to
 25097 queue them will be released and made available to queue other signals.

25098 SIG_IGN — ignore signal

25099 • Delivery of the signal will have no effect on the process. The behaviour of a process is
 25100 RT undefined after it ignores a SIGFPE, SIGILL, SIGSEGV or SIGBUS signal that was not
 25101 RT generated by *kill()*, *sigqueue()* or *raise()*.

25102 • The system will not allow the action for the signals SIGKILL or SIGSTOP to be set to
 25103 SIG_IGN.

25104 • Setting a signal action to SIG_IGN for a signal that is pending will cause the pending
25105 signal to be discarded, whether or not it is blocked.

25106 • If a process sets the action for the SIGCHLD signal to SIG_IGN, the behaviour is
25107 EX unspecified, except as specified below.

25108 If the action for the SIGCHLD signal is set to SIG_IGN, child processes of the calling
25109 processes will not be transformed into zombie processes when they terminate. If the
25110 calling process subsequently waits for its children, and the process has no unwaited for
25111 children that were transformed into zombie processes, it will block until all of its
25112 children terminate, and *wait()*, *wait3()*, *waitid()* and *waitpid()* will fail and set *errno* to
25113 [ECHILD].

25114 RT If the Realtime Signals Extension option is supported, any queued values pending will
25115 be discarded and the resources used to queue them will be released and made available
25116 to queue other signals.

25117 *pointer to a function* — catch signal

25118 • On delivery of the signal, the receiving process is to execute the signal-catching function
25119 at the specified address. After returning from the signal-catching function, the receiving
25120 process will resume execution at the point at which it was interrupted.

25121 If the SA_SIGINFO flag for the signal is cleared, the signal-catching function will be
25122 entered as a C language function call as follows:

```
25123       void func(int signo);
```

25124 If the SA_SIGINFO flag for the signal is set, the signal-catching function will be entered
25125 as a C language function call as follows:

```
25126       void func(int signo, siginfo_t *info, void *context);
```

25127 where *func* is the specified signal-catching function, *signo* is the signal number of the
25128 signal being delivered, and *info* is a pointer to a **siginfo_t** structure defined in **<signal.h>**
25129 containing at least the following member(s):

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

25135 The *si_signo* member contains the signal number. This is the same as the *signo*
25136 parameter. The *si_code* member contains a code identifying the cause of the signal. The
25137 following values are defined for *si_code*:

25138 SI_USER The signal was sent by the *kill()* function. The implementation may
25139 set *si_code* to SI_USER if the signal was sent by the *raise()* or *abort()*
25140 functions or any similar functions provided as implementation
25141 extensions.

25142 RT SI_QUEUE The signal was sent by the *sigqueue()* function.

25143 SI_TIMER The signal was generated by the expiration of a timer set by
25144 *timer_settime()*.

25145 SI_ASYNCIO The signal was generated by the completion of an asynchronous I/O
25146 request.

25147 SI_MESGQ The signal was generated by the arrival of a message on an empty
25148 message queue.

25149 If the signal was not generated by one of the functions or events listed above, the *si_code*
25150 will be set to an implementation-dependent value that is not equal to any of the values
25151 defined above.

25152 RT If the Realtime Signals Extension is supported, and *si_code* is one of SI_QUEUE,
25153 SI_TIMER, SI_ASYNCIO, or SI_MESGQ, then *si_value* contains the application-specified
25154 signal value. Otherwise, the contents of *si_value* are undefined.

25155

- The behaviour of a process is undefined after it returns normally from a signal-catching
25156 EX function for a SIGBUS, SIGFPE, SIGILL or SIGSEGV signal that was not generated by
25157 RT *kill()*, *sigqueue()* or *raise()*.

25158

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

25159

- If a process establishes a signal-catching function for the SIGCHLD signal while it has a
25160 terminated child process for which it has not waited, it is unspecified whether a
25161 SIGCHLD signal is generated to indicate that child process.

25162

- When signal-catching functions are invoked asynchronously with process execution, the
25163 behaviour of some of the functions defined by this document is unspecified if they are
25164 called from a signal-catching function.

25165 The following table defines a set of interfaces that are either reentrant or not
25166 interruptible by signals and are async-signal safe. Therefore applications may invoke
25167 them, without restriction, from signal-catching functions:

25168

Base Interfaces

25169	<i>_exit()</i>	<i>fstat()</i>	<i>raise()</i>	<i>stat()</i>
25170	<i>access()</i>	<i>fsync()</i>	<i>read()</i>	<i>sysconf()</i>
25171	<i>alarm()</i>	<i>getegid()</i>	<i>rename()</i>	<i>tcdrain()</i>
25172	<i>cfgetispeed()</i>	<i>geteuid()</i>	<i>rmdir()</i>	<i>tcflow()</i>
25173	<i>cfgetospeed()</i>	<i>getgid()</i>	<i>setgid()</i>	<i>tcflush()</i>
25174	<i>cfsetispeed()</i>	<i>getgroups()</i>	<i>setpgid()</i>	<i>tcgetattr()</i>
25175	<i>cfsetospeed()</i>	<i>getpgrp()</i>	<i>setsid()</i>	<i>tcgetpgrp()</i>
25176	<i>chdir()</i>	<i>getpid()</i>	<i>setuid()</i>	<i>tcsendbreak()</i>
25177	<i>chmod()</i>	<i>getppid()</i>	<i>sigaction()</i>	<i>tcsetattr()</i>
25178	<i>chown()</i>	<i>getuid()</i>	<i>sigaddset()</i>	<i>tcsetpgrp()</i>
25179	<i>close()</i>	<i>kill()</i>	<i>sigdelset()</i>	<i>time()</i>
25180	<i>creat()</i>	<i>link()</i>	<i>sigemptyset()</i>	<i>times()</i>
25181	<i>dup()</i>	<i>lseek()</i>	<i>sigfillset()</i>	<i>umask()</i>
25182	<i>dup2()</i>	<i>mkdir()</i>	<i>sigismember()</i>	<i>uname()</i>
25183	<i>execle()</i>	<i>mkfifo()</i>	<i>signal()</i>	<i>unlink()</i>
25184	<i>execve()</i>	<i>open()</i>	<i>sigpending()</i>	<i>utime()</i>
25185	<i>fcntl()</i>	<i>pathconf()</i>	<i>sigprocmask()</i>	<i>wait()</i>
25186	<i>fork()</i>	<i>pause()</i>	<i>sigsuspend()</i>	<i>waitpid()</i>
25187	<i>fpathconf()</i>	<i>pipe()</i>	<i>sleep()</i>	<i>write()</i>

25188 **Realtime Interfaces**

25189 RT	<i>aio_error()</i>	<i>clock_gettime()</i>	<i>sigpause()</i>	<i>timer_getoverrun()</i>
25190	<i>aio_return()</i>	<i>fdatasync()</i>	<i>sigqueue()</i>	<i>timer_gettime()</i>
25191	<i>aio_suspend()</i>	<i>sem_post()</i>	<i>sigset()</i>	<i>timer_settime()</i>

25192 All functions not in the above table are considered to be unsafe with respect to signals.
 25193 In the presence of signals, all functions defined by this specification will behave as
 25194 defined when called from or interrupted by a signal-catching function, with a single
 25195 exception: when a signal interrupts an unsafe function and the signal-catching function
 25196 calls an unsafe function, the behaviour is undefined.

25197 When a signal is delivered to a thread, if the action of that signal specifies termination, stop, or
 25198 continue, the entire process will be terminated, stopped, or continued, respectively.

25199 **Signal Effects on Other Functions**

25200 Signals affect the behaviour of certain functions defined by this specification if delivered to a
 25201 process while it is executing such a function. If the action of the signal is to terminate the
 25202 process, the process will be terminated and the function will not return. If the action of the
 25203 signal is to stop the process, the process will stop until continued or terminated. Generation of a
 25204 SIGCONT signal for the process causes the process to be continued, and the original function
 25205 will continue at the point the process was stopped. If the action of the signal is to invoke a
 25206 signal-catching function, the signal-catching function will be invoked; in this case the original
 25207 function is said to be *interrupted* by the signal. If the signal-catching function executes a **return**
 25208 statement, the behaviour of the interrupted function will be as described individually for that
 25209 function. Signals that are ignored will not affect the behaviour of any function; signals that are
 25210 blocked will not affect the behaviour of any function until they are unblocked and then
 25211 delivered, except as specified for and the *sigwait()* functions.

25212 The result of the use of *sigaction()* and a *sigwait()* function concurrently within a process on the
 25213 same signal is unspecified.

25214 **RETURN VALUE**

25215 Upon successful completion, *sigaction()* returns 0. Otherwise -1 is returned, *errno* is set to
 25216 indicate the error and no new signal-catching function will be installed.

25217 **ERRORS**

25218 The *sigaction()* function will fail if:

25219 [EINVAL] The *sig* argument is not a valid signal number or an attempt is made to catch a
 25220 signal that cannot be caught or ignore a signal that cannot be ignored.

25221 The *sigaction()* function may fail if:

25222 [EINVAL] An attempt was made to set the action to SIG_DFL for a signal that cannot be
 25223 caught or ignored (or both).

25224 **EXAMPLES**

25225 None.

25226 **APPLICATION USAGE**

25227 The *sigaction()* function supersedes the *signal()* interface, and should be used in preference. In
 25228 particular, *sigaction()* and *signal()* should not be used in the same process to control the same
 25229 signal. The behaviour of reentrant interfaces, as defined in the description, is as specified by this
 25230 specification, regardless of invocation from a signal-catching function. This is the only intended
 25231 meaning of the statement that reentrant interfaces may be used in signal-catching functions

without restrictions. Applications must still consider all effects of such functions on such things as data structures, files and process state. In particular, application writers need to consider the restrictions on interactions when interrupting *sleep()* and interactions among multiple handles for a file description. The fact that any specific interface is listed as reentrant does not necessarily mean that invocation of that interface from a signal-catching function is recommended.

In order to prevent errors arising from interrupting non-reentrant function calls, applications should protect calls to these functions either by blocking the appropriate signals or through the use of some programmatic semaphore (see *semget()*, *sem_init()*, *sem_open()*, and so on). Note in particular that even the “safe” functions may modify *errno*; the signal-catching function, if not executing as an independent thread, may want to save and restore its value. Naturally, the same principles apply to the reentrancy of application routines and asynchronous data access. Note that *longjmp()* and *siglongjmp()* are not in the list of reentrant interfaces. This is because the code executing after *longjmp()* and *siglongjmp()* can call any unsafe functions with the same danger as calling those unsafe functions directly from the signal handler. Applications that use *longjmp()* and *siglongjmp()* from within signal handlers require rigorous protection in order to be portable. Many of the other functions that are excluded from the list are traditionally implemented using either *malloc()* or *free()* functions or the standard I/O library, both of which traditionally use data structures in a non-reentrant manner. Because any combination of different functions using a common data structure can cause reentrancy problems, this document does not define the behaviour when any unsafe function is called in a signal handler that interrupts an unsafe function.

If the signal occurs other than as the result of calling *abort()*, *kill()* or *raise()*, the behaviour is undefined if the signal handler calls any function in the standard library other than one of the functions listed in the table above or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type **volatile sig_atomic_t**. Furthermore, if such a call fails, the value of *errno* is indeterminate.

Usually, the signal is executed on the stack that was in effect before the signal was delivered. An alternate stack may be specified to receive a subset of the signals being caught.

When the signal handler returns, the receiving process will resume 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.

The ISO POSIX-1 standard 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 specification. 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.

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 revision of that standard.

25280 **SEE ALSO**

25281 *bsd_signal()*, *kill()*, *_longjmp()*, *longjmp()*, *raise()*, *semget()*, *sem_init()*, *sem_open()*, *sigaddset()*,
 25282 *sigaltstack()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*, *signal()*, *sigprocmask()*,
 25283 *sigsuspend()*, *wait()*, *wait3()*, *waitid()*, *waitpid()*, **<signal.h>**, **<ucontext.h>**.

25284 **CHANGE HISTORY**

25285 First released in Issue 3.

25286 Entry included for alignment with the POSIX.1-1988 standard.

25287 **Issue 4**

25288 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- 25289 • The type of argument *act* is changed from **struct sigaction *** to **const struct sigaction ***.
- 25290 • A statement is added to the DESCRIPTION indicating that the consequence of attempting to
 25291 set SIG_DFL for a signal that cannot be caught or ignored is unspecified. The [EINVAL]
 25292 error, describing one possible reaction to this condition, is added to the ERRORS section.

25293 Other changes are incorporated as follows:

- 25294 • The *raise()* and *signal()* functions are added to the list of interfaces that are either reentrant or
 25295 not interruptible by signals; *fpathconf()* is also added to this list and marked as an extension;
 25296 *ustat()* is removed from the list, as this function is withdrawn from the interface definition. It
 25297 is no longer specified whether *abort()*, *exit()* and *longjmp()* also fall into this category of
 25298 functions.
- 25299 • The APPLICATION USAGE section is added. Most of this text is moved from the
 25300 DESCRIPTION in Issue 3.
- 25301 • The FUTURE DIRECTIONS section is added.

25302 **Issue 4, Version 2**

25303 The following changes are incorporated for X/OPEN UNIX conformance:

- 25304 • The DESCRIPTION describes **sa_sigaction**, the member of the **sigaction** structure that is the
 25305 signal-catching function.
- 25306 • The DESCRIPTION describes the SA_ONSTACK, SA_RESETHAND, SA_RESTART,
 25307 SA_SIGINFO, SA_NOCLDWAIT and SA_NODEFER settings of *sa_flags*. The text describes
 25308 the implications of the use of SA_SIGINFO for the number of arguments passed to the
 25309 signal-catching function. The text also describes the effects of the SA_NODEFER and
 25310 SA_RESETHAND flags on the delivery of a signal and on the permanence of an installed
 25311 action.
- 25312 • The DESCRIPTION specifies the effect if the action for the SIGCHLD signal is set to
 25313 SIG_IGN.
- 25314 • In the DESCRIPTION, additional text describes the effect if the action is a pointer to a
 25315 function. A new bullet covers the case where SA_SIGINFO is set. SIGBUS is given as an
 25316 additional signal for which the behaviour of a process is undefined following a normal return
 25317 from the signal-catching function.
- 25318 • The APPLICATION USAGE section is updated to describe use of an alternate signal stack;
 25319 resumption of the process receiving the signal; coding for compatibility with POSIX.4-1993;
 25320 and implementation of signal-handling functions in BSD.

25321 **Issue 5**

25322 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and POSIX
 25323 Threads Extension.

25324	In the DESCRIPTION, the second argument to <i>func</i> when SA_SIGINFO is set is no longer	
25325	permitted to be NULL, and the description of permitted siginfo_t contents is expanded by	
25326	reference to < signal.h >.	
25327	Because the X/OPEN UNIX Extension functionality is now folded into the BASE, the	
25328	[ENOTSUP] error is deleted.	

25329 **NAME**

25330 sigaddset — add a signal to a signal set

25331 **SYNOPSIS**

25332 #include <signal.h>

25333 int sigaddset(sigset_t *set, int signo);

25334 **DESCRIPTION**25335 The *sigaddset()* function adds the individual signal specified by the *signo* to the signal set pointed
25336 to by *set*.25337 Applications must call either *sigemptyset()* or *sigfillset()* at least once for each object of type
25338 **sigset_t** prior to any other use of that object. If such an object is not initialised in this way, but is
25339 nonetheless supplied as an argument to any of *sigaction()*, *sigaddset()*, *sigdelset()*, *sigismember()*,
25340 *sigpending()* or *sigprocmask()*, the results are undefined.25341 **RETURN VALUE**25342 Upon successful completion, *sigaddset()* returns 0. Otherwise, it returns -1 and sets *errno* to
25343 indicate the error.25344 **ERRORS**25345 The *sigaddset()* function may fail if:25346 [EINVAL] The value of the *signo* argument is an invalid or unsupported signal number.25347 **EXAMPLES**

25348 None.

25349 **APPLICATION USAGE**

25350 None.

25351 **FUTURE DIRECTIONS**

25352 None.

25353 **SEE ALSO**25354 *sigaction()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*, *sigpending()*, *sigprocmask()*,
25355 *sigsuspend()*, <signal.h>.25356 **CHANGE HISTORY**

25357 First released in Issue 3.

25358 Entry included for alignment with the POSIX.1-1988 standard.

25359 **Issue 4**

25360 The following change is incorporated in this issue:

- 25361
- The word “will” is replaced by the word “may” in the ERRORS section.

25362 **Issue 5**25363 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
25364 previous issues.

25365 **NAME**

25366 sigaltstack — set and/or get signal alternate stack context.

25367 **SYNOPSIS**25368 EX `#include <signal.h>`25369 `int sigaltstack(const stack_t *ss, stack_t *oss);`

25370

25371 **DESCRIPTION**

25372 The *sigaltstack()* function allows a process to define and examine the state of an alternate stack
 25373 for signal handlers. Signals that have been explicitly declared to execute on the alternate stack
 25374 will be delivered on the alternate stack.

25375 If *ss* is not a null pointer, it points to a **stack_t** structure that specifies the alternate signal stack
 25376 that will take effect upon return from *sigaltstack()*. The **ss_flags** member specifies the new stack
 25377 state. If it is set to **SS_DISABLE**, the stack is disabled and **ss_sp** and **ss_size** are ignored.
 25378 Otherwise the stack will be enabled, and the **ss_sp** and **ss_size** members specify the new
 25379 address and size of the stack.

25380 The range of addresses starting at **ss_sp**, up to but not including **ss_sp + ss_size**, is available to
 25381 the implementation for use as the stack. This interface makes no assumptions regarding which
 25382 end is the stack base and in which direction the stack grows as items are pushed.

25383 If *oss* is not a null pointer, on successful completion it will point to a **stack_t** structure that
 25384 specifies the alternate signal stack that was in effect prior to the call to *sigaltstack()*. The **ss_sp**
 25385 and **ss_size** members specify the address and size of that stack. The **ss_flags** member specifies
 25386 the stack's state, and may contain one of the following values:

25387 **SS_ONSTACK** The process is currently executing on the alternate signal stack. Attempts to
 25388 modify the alternate signal stack while the process is executing on it fails.
 25389 This flag must not be modified by processes.

25390 **SS_DISABLE** The alternate signal stack is currently disabled.

25391 The value **SIGSTKSZ** is a system default specifying the number of bytes that would be used to
 25392 cover the usual case when manually allocating an alternate stack area. The value
 25393 **MINSIGSTKSZ** is defined to be the minimum stack size for a signal handler. In computing an
 25394 alternate stack size, a program should add that amount to its stack requirements to allow for the
 25395 system implementation overhead. The constants **SS_ONSTACK**, **SS_DISABLE**, **SIGSTKSZ**, and
 25396 **MINSIGSTKSZ** are defined in **<signal.h>**.

25397 After a successful call to one of the *exec* functions, there are no alternate signal stacks in the new
 25398 process image.

25399 In some implementations, a signal (whether or not indicated to execute on the alternate stack)
 25400 will always execute on the alternate stack if it is delivered while another signal is being caught
 25401 using the alternate stack.

25402 Use of this function by library threads that are not bound to kernel-scheduled entities results in
 25403 undefined behaviour.

25404 **RETURN VALUE**

25405 Upon successful completion, *sigaltstack()* returns 0. Otherwise, it returns **-1** and sets *errno* to
 25406 indicate the error.

25407 **ERRORS**

25408 The *sigaltstack()* function will fail if:

- 25409 [EINVAL] The *ss* argument is not a null pointer, and the **ss_flags** member pointed to by *ss* contains flags other than **SS_DISABLE**.
- 25410
- 25411 [ENOMEM] The size of the alternate stack area is less than **MINSIGSTKSZ**.
- 25412 [EPERM] An attempt was made to modify an active stack.

25413 **EXAMPLES**

25414 None.

25415 **APPLICATION USAGE**

25416 The following code fragment illustrates a method for allocating memory for an alternate stack:

```
25417       if ((sigstk.ss_sp = malloc(SIGSTKSZ)) == NULL)
25418           /* error return */
25419       sigstk.ss_size = SIGSTKSZ;
25420       sigstk.ss_flags = 0;
25421       if (sigaltstack(&sigstk, (stack_t *)0) < 0)
25422           perror("sigaltstack");
```

25423 On some implementations, stack space is automatically extended as needed. On those
 25424 implementations, automatic extension is typically not available for an alternate stack. If the
 25425 stack overflows, the behaviour is undefined.

25426 **FUTURE DIRECTIONS**

25427 None.

25428 **SEE ALSO**

25429 *sigaction()*, *sigsetjmp()*, <signal.h>.

25430 **CHANGE HISTORY**

25431 First released in Issue 4, Version 2.

25432 **Issue 5**

25433 Moved from X/OPEN UNIX extension to BASE.

25434 The last sentence of the DESCRIPTION was included as an APPLICATION USAGE note in
 25435 previous issues.

25436 **NAME**

25437 sigdelset — delete a signal from a signal set

25438 **SYNOPSIS**

25439 #include <signal.h>

25440 int sigdelset(sigset_t *set, int signo);

25441 **DESCRIPTION**25442 The *sigdelset()* function deletes the individual signal specified by *signo* from the signal set pointed to by *set*.25444 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 initialised in this way, but is nonetheless supplied as an argument to any of *sigaction()*, *sigaddset()*, *sigdelset()*, *sigismember()*, *sigpending()* or *sigprocmask()*, the results are undefined.25448 **RETURN VALUE**25449 Upon successful completion, *sigdelset()* returns 0. Otherwise, it returns -1 and sets *errno* to indicate the error.25451 **ERRORS**25452 The *sigdelset()* function may fail if:25453 [EINVAL] The *signo* argument is not a valid signal number, or is an unsupported signal number.25455 **EXAMPLES**

25456 None.

25457 **APPLICATION USAGE**

25458 None.

25459 **FUTURE DIRECTIONS**

25460 None.

25461 **SEE ALSO**25462 *sigaction()*, *sigaddset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*, *sigpending()*, *sigprocmask()*, *sigsuspend()*, <signal.h>.25464 **CHANGE HISTORY**

25465 First released in Issue 3.

25466 Entry included for alignment with the POSIX.1-1988 standard.

25467 **Issue 4**

25468 The following change is incorporated in this issue:

- 25469
- The word “will” is replaced by the word “may” in the ERRORS section.

25470 **Issue 5**

25471 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

25473 **NAME**

25474 sigemptyset — initialise and empty a signal set

25475 **SYNOPSIS**

25476 #include <signal.h>

25477 int sigemptyset(sigset_t *set);

25478 **DESCRIPTION**

25479 The *sigemptyset()* function initialises the signal set pointed to by *set*, such that all signals defined
25480 in this document are excluded.

25481 **RETURN VALUE**

25482 Upon successful completion, *sigemptyset()* returns 0. Otherwise, it returns -1 and sets *errno* to
25483 indicate the error.

25484 **ERRORS**

25485 No errors are defined.

25486 **EXAMPLES**

25487 None.

25488 **APPLICATION USAGE**

25489 None.

25490 **FUTURE DIRECTIONS**

25491 None.

25492 **SEE ALSO**

25493 *sigaction()*, *sigaddset()*, *sigdelset()*, *sigfillset()*, *sigismember()*, *sigpending()*, *sigprocmask()*,
25494 *sigsuspend()*, <signal.h>.

25495 **CHANGE HISTORY**

25496 First released in Issue 3.

25497 Entry included for alignment with the POSIX.1-1988 standard.

25498 **NAME**

25499 sigfillset — initialise and fill a signal set

25500 **SYNOPSIS**

25501 #include <signal.h>

25502 int sigfillset(sigset_t *set);

25503 **DESCRIPTION**25504 The *sigfillset()* function initialises the signal set pointed to by *set*, such that all signals defined in
25505 this document are included.25506 **RETURN VALUE**25507 Upon successful completion, *sigfillset()* returns 0. Otherwise, it returns -1 and sets *errno* to
25508 indicate the error.25509 **ERRORS**

25510 No errors are defined.

25511 **EXAMPLES**

25512 None.

25513 **APPLICATION USAGE**

25514 None.

25515 **FUTURE DIRECTIONS**

25516 None.

25517 **SEE ALSO**25518 *sigaction()*, *sigaddset()*, *sigdelset()*, *sigemptyset()*, *sigismember()*, *sigpending()*, *sigprocmask()*,
25519 *sigsuspend()*, <signal.h>.25520 **CHANGE HISTORY**

25521 First released in Issue 3.

25522 Entry included for alignment with the POSIX.1-1988 standard.

25523 **NAME**

25524 sighold, sigignore — add a signal to the signal mask or set a signal disposition to be ignored

25525 **SYNOPSIS**

25526 EX `#include <signal.h>`

25527 `int sighold(int sig);`

25528 `int sigignore(int sig);`

25529

25530 **DESCRIPTION**

25531 Refer to *signal()*.

25532 **CHANGE HISTORY**

25533 First released in Issue 4, Version 2.

25534 **Issue 5**

25535 Moved from X/OPEN UNIX extension to BASE.

25536 **NAME**

25537 siginterrupt — allow signals to interrupt functions

25538 **SYNOPSIS**25539 EX `#include <signal.h>`25540 `int siginterrupt(int sig, int flag);`

25541

25542 **DESCRIPTION**

25543 The *siginterrupt()* function is used to change the restart behaviour when a function is interrupted
 25544 by the specified signal. The function *siginterrupt(sig, flag)* has an effect as if implemented as:

```

25545     siginterrupt(int sig, int flag) {
25546         int ret;
25547         struct sigaction act;

25548         (void) sigaction(sig, NULL, &act);
25549         if (flag)
25550             act.sa_flags &= ~SA_RESTART;
25551         else
25552             act.sa_flags |= SA_RESTART;
25553         ret = sigaction(sig, &act, NULL);
25554         return ret;
25555     }
```

25556 **RETURN VALUE**

25557 Upon successful completion, *siginterrupt()* returns 0. Otherwise -1 is returned and *errno* is set to
 25558 indicate the error.

25559 **ERRORS**25560 The *siginterrupt()* function will fail if:

25561 [EINVAL] The *sig* argument is not a valid signal number.

25562 **EXAMPLES**

25563 None.

25564 **APPLICATION USAGE**

25565 The *siginterrupt()* function supports programs written to historical system interfaces. A portable
 25566 application, when being written or rewritten, should use *sigaction()* with the SA_RESTART flag
 25567 instead of *siginterrupt()*.

25568 **FUTURE DIRECTIONS**

25569 None.

25570 **SEE ALSO**25571 *sigaction()*, <signal.h>.25572 **CHANGE HISTORY**

25573 First released in Issue 4, Version 2.

25574 **Issue 5**

25575 Moved from X/OPEN UNIX extension to BASE.

25576 **NAME**

25577 sigismember — test for a signal in a signal set

25578 **SYNOPSIS**

25579 #include <signal.h>

25580 int sigismember(const sigset_t *set, int signo);

25581 **DESCRIPTION**25582 The *sigismember()* function tests whether the signal specified by *signo* is a member of the set
25583 pointed to by *set*.25584 Applications should call either *sigemptyset()* or *sigfillset()* at least once for each object of type
25585 **sigset_t** prior to any other use of that object. If such an object is not initialised in this way, but is
25586 nonetheless supplied as an argument to any of *sigaction()*, *sigaddset()*, *sigdelset()*, *sigismember()*,
25587 *sigpending()* or *sigprocmask()*, the results are undefined.25588 **RETURN VALUE**25589 Upon successful completion, *sigismember()* returns 1 if the specified signal is a member of the
25590 specified set, or 0 if it is not. Otherwise, it returns -1 and sets *errno* to indicate the error.25591 **ERRORS**25592 The *sigismember()* function may fail if:25593 [EINVAL] The *signo* argument is not a valid signal number, or is an unsupported signal
25594 number.25595 **EXAMPLES**

25596 None.

25597 **APPLICATION USAGE**

25598 None.

25599 **FUTURE DIRECTIONS**

25600 None.

25601 **SEE ALSO**25602 *sigaction()*, *sigaddset()*, *sigdelset()*, *sigfillset()*, *sigemptyset()*, *sigpending()*, *sigprocmask()*,
25603 *sigsuspend()*, <**signal.h**>.25604 **CHANGE HISTORY**

25605 First released in Issue 3.

25606 Entry included for alignment with the POSIX.1-1988 standard.

25607 **Issue 4**

25608 The following changes are incorporated for alignment with the ISO C standard:

- 25609
- 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.
- 25610

25611 **Issue 5**25612 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
25613 previous issues.

25614 **NAME**

25615 siglongjmp — non-local goto with signal handling

25616 **SYNOPSIS**

25617 #include <setjmp.h>

25618 void siglongjmp(sigjmp_buf env, int val);

25619 **DESCRIPTION**

25620 The *siglongjmp()* function restores the environment saved by the most recent invocation of
 25621 *sigsetjmp()* in the same thread, with the corresponding *sigjmp_buf* argument. If there is no such
 25622 invocation, or if the function containing the invocation of *sigsetjmp()* has terminated execution in
 25623 the interim, the behaviour is undefined.

25624 All accessible objects have values as of the time *sigsetjmp()* was called, except that the values of
 25625 objects of automatic storage duration which are local to the function containing the invocation of
 25626 the corresponding *sigsetjmp()* which do not have volatile-qualified type and which are changed
 25627 between the *sigsetjmp()* invocation and *siglongjmp()* call are indeterminate.

25628 As it bypasses the usual function call and return mechanisms, *siglongjmp()* will execute correctly
 25629 in contexts of interrupts, signals and any of their associated functions. However, if *siglongjmp()*
 25630 is invoked from a nested signal handler (that is, from a function invoked as a result of a signal
 25631 raised during the handling of another signal), the behaviour is undefined.

25632 The *siglongjmp()* function will restore the saved signal mask if and only if the *env* argument was
 25633 initialised by a call to *sigsetjmp()* with a non-zero *savemask* argument.

25634 The effect of a call to *siglongjmp()* where initialisation of the *jmp_buf* structure was not
 25635 performed in the calling thread is undefined.

25636 **RETURN VALUE**

25637 After *siglongjmp()* is completed, program execution continues as if the corresponding invocation
 25638 of *sigsetjmp()* had just returned the value specified by *val*. The *siglongjmp()* function cannot
 25639 cause *sigsetjmp()* to return 0; if *val* is 0, *sigsetjmp()* returns the value 1.

25640 **ERRORS**

25641 No errors are defined.

25642 **EXAMPLES**

25643 None.

25644 **APPLICATION USAGE**

25645 The distinction between *setjmp()* or *longjmp()* and *sigsetjmp()* or *siglongjmp()* is only significant
 25646 for programs which use *sigaction()*, *sigprocmask()* or *sigsuspend()*.

25647 **FUTURE DIRECTIONS**

25648 None.

25649 **SEE ALSO**25650 *longjmp()*, *setjmp()*, *sigprocmask()*, *sigsetjmp()*, *sigsuspend()*, <setjmp.h>.25651 **CHANGE HISTORY**

25652 First released in Issue 3.

25653 Entry included for alignment with the ISO POSIX-1 standard.

25654 **Issue 4**

25655 The following changes are incorporated in this issue:

- 25656 • The APPLICATION USAGE section is amended.

25657 • An ERRORS section is added.

25658 **Issue 5**

25659 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

25660 **NAME**

25661 signal, sigset, sighold, sigrelse, sigignore, sigpause — signal management

25662 **SYNOPSIS**

25663 #include <signal.h>

25664 void (*signal(int *sig*, void (**func*)(int)))(int);25665 EX int sighold(int *sig*);25666 int sigignore(int *sig*);25667 int sigpause(int *sig*);25668 int sigrelse(int *sig*);25669 void (*sigset(int *sig*, void (**disp*)(int)))(int);

25670

25671 **DESCRIPTION**

25672 Use of any of these functions is unspecified in a multi-threaded process.

25673 The *signal()* function chooses one of three ways in which receipt of the signal number *sig* is to be
 25674 subsequently handled. If the value of *func* is SIG_DFL, default handling for that signal will
 25675 occur. If the value of *func* is SIG_IGN, the signal will be ignored. Otherwise, *func* must point to
 25676 a function to be called when that signal occurs. Such a function is called a *signal handler*.

25677 When a signal occurs, if *func* points to a function, first the equivalent of a:25678 signal(*sig*, SIG_DFL);

25679 is executed or an implementation-dependent blocking of the signal is performed. (If the value of
 25680 *sig* is SIGILL, whether the reset to SIG_DFL occurs is implementation-dependent.) Next the
 25681 equivalent of:

25682 (*func)(*sig*);

25683 is executed. The *func* function may terminate by executing a **return** statement or by calling
 25684 *abort()*, *exit()*, or *longjmp()*. If *func* executes a **return** statement and the value of *sig* was SIGFPE
 25685 or any other implementation-dependent value corresponding to a computational exception, the
 25686 behaviour is undefined. Otherwise, the program will resume execution at the point it was
 25687 interrupted.

25688 If the signal occurs other than as the result of calling *abort()*, *kill()* or *raise()*, the behaviour is
 25689 undefined if the signal handler calls any function in the standard library other than one of the
 25690 functions listed on the *sigaction()* page or refers to any object with static storage duration other
 25691 than by assigning a value to a static storage duration variable of type **volatile sig_atomic_t**.
 25692 Furthermore, if such a call fails, the value of *errno* is indeterminate.

25693 At program startup, the equivalent of:

25694 signal(*sig*, SIG_IGN);

25695 is executed for some signals, and the equivalent of:

25696 signal(*sig*, SIG_DFL);25697 is executed for all other signals (see *exec*).

25698 EX The *sigset()*, *sighold()*, *sigignore()*, *sigpause()* and *sigrelse()* functions provide simplified signal
 25699 management.

25700 The *sigset()* function is used to modify signal dispositions. The *sig* argument specifies the signal,
 25701 which may be any signal except SIGKILL and SIGSTOP. The *disp* argument specifies the signal's
 25702 disposition, which may be SIG_DFL, SIG_IGN or the address of a signal handler. If *sigset()* is
 25703 used, and *disp* is the address of a signal handler, the system will add *sig* to the calling process'

25704 signal mask before executing the signal handler; when the signal handler returns, the system will
 25705 restore the calling process' signal mask to its state prior the delivery of the signal. In addition, if
 25706 *sigset()* is used, and *disp* is equal to SIG_HOLD, *sig* will be added to the calling process' signal
 25707 mask and *sig*'s disposition will remain unchanged. If *sigset()* is used, and *disp* is not equal to
 25708 SIG_HOLD, *sig* will be removed from the calling process' signal mask.

25709 The *sighold()* function adds *sig* to the calling process' signal mask.

25710 The *sigrelse()* function removes *sig* from the calling process' signal mask.

25711 The *sigignore()* function sets the disposition of *sig* to SIG_IGN.

25712 The *sigpause()* function removes *sig* from the calling process' signal mask and suspends the
 25713 calling process until a signal is received. The *sigpause()* function restores the process' signal
 25714 mask to its original state before returning.

25715 If the action for the SIGCHLD signal is set to SIG_IGN, child processes of the calling processes
 25716 will not be transformed into zombie processes when they terminate. If the calling process
 25717 subsequently waits for its children, and the process has no unwaited for children that were
 25718 transformed into zombie processes, it will block until all of its children terminate, and *wait()*,
 25719 *wait3()*, *waitid()* and *waitpid()* will fail and set *errno* to [ECHILD].

25720 RETURN VALUE

25721 If the request can be honoured, *signal()* returns the value of *func* for the most recent call to
 25722 *signal()* for the specified signal *sig*. Otherwise, SIG_ERR is returned and a positive value is
 25723 stored in *errno*.

25724 EX Upon successful completion, *sigset()* returns SIG_HOLD if the signal had been blocked and the
 25725 signal's previous disposition if it had not been blocked. Otherwise, SIG_ERR is returned and
 25726 *errno* is set to indicate the error.

25727 The *sigpause()* function suspends execution of the thread until a signal is received, whereupon it
 25728 returns -1 and sets *errno* to [EINTR].

25729 For all other functions, upon successful completion, 0 is returned. Otherwise, -1 is returned and
 25730 *errno* is set to indicate the error.

25731 ERRORS

25732 The *signal()* function will fail if:

25733 [EINVAL] The *sig* argument is not a valid signal number or an attempt is made to catch a
 25734 signal that cannot be caught or ignore a signal that cannot be ignored.

25735 The *signal()* function may fail if:

25736 [EINVAL] An attempt was made to set the action to SIG_DFL for a signal that cannot be
 25737 caught or ignored (or both).

25738 EX The *sigset()*, *sighold()*, *sigrelse()*, *sigignore()* and *sigpause()* functions will fail if:

25739 [EINVAL] The *sig* argument is an illegal signal number.

25740 The *sigset()*, and *sigignore()* functions will fail if:

25741 [EINVAL] An attempt is made to catch a signal that cannot be caught, or to ignore a
 25742 signal that cannot be ignored.

25743 EXAMPLES

25744 None.

25745 **APPLICATION USAGE**

25746 The *sigaction()* function provides a more comprehensive and reliable mechanism for controlling
 25747 signals; new applications should use *sigaction()* rather than *signal()*.

25748 The *sighold()* function, in conjunction with *sigrelse()* or *sigpause()*, may be used to establish
 25749 critical regions of code that require the delivery of a signal to be temporarily deferred.

25750 The *sigsuspend()* function should be used in preference to *sigpause()* for broader portability.

25751 **FUTURE DIRECTIONS**

25752 None.

25753 **SEE ALSO**

25754 *exec*, *pause()*, *sigaction()*, *sigsuspend()*, *waitid()*, <signal.h>.

25755 **CHANGE HISTORY**

25756 First released in Issue 1.

25757 Derived from Issue 1 of the SVID.

25758 **Issue 4**

25759 The following changes are incorporated for alignment with the ISO C standard:

- 25760 • The function is no longer marked as an extension.
- 25761 • The argument **int** is added to the definition of *func* in the SYNOPSIS section.
- 25762 • In Issue 3, this interface cross-referred to *sigaction()*. This issue provides a complete
 25763 description of the function as defined in ISO C standard.

25764 Another change is incorporated as follows:

- 25765 • The APPLICATION USAGE section is added.

25766 **Issue 4, Version 2**

25767 The following changes are incorporated for X/OPEN UNIX conformance:

- 25768 • The *sighold()*, *sigignore()*, *sigpause()*, *sigrelse()* and *sigset()* functions are added to the
 25769 SYNOPSIS.
- 25770 • The DESCRIPTION is updated to describe semantics of the above interfaces.
- 25771 • Additional text is added to the RETURN VALUE section to describe possible returns from
 25772 the *sigset()* function specifically, and all of the above functions in general.
- 25773 • The ERRORS section is restructured to describe possible error returns from each of the above
 25774 functions individually.
- 25775 • The APPLICATION USAGE section is updated to describe certain programming
 25776 considerations associated with the X/OPEN UNIX functions.

25777 **Issue 5**

25778 The DESCRIPTION is updated to indicate that the *sigpause()* function restores the process'
 25779 signal mask to its original state before returning.

25780 The RETURN VALUE section is updated to indicate that the *sigpause()* function suspends
 25781 execution of the process until a signal is received, whereupon it returns -1 and sets *errno* to
 25782 EINTR.

25783 NAME

25784 signgam — storage for sign of lgamma()

25785 SYNOPSIS

25786 EX `#include <math.h>`

25787 `extern int signgam;`

25788

25789 DESCRIPTION

25790 Refer to *lgamma()*.

25791 CHANGE HISTORY

25792 First released in Issue 1.

25793 Derived from Issue 1 of the SVID.

25794 Issue 4

25795 The following change is incorporated in this issue:

- 25796 • The `<math.h>` header is added to the SYNOPSIS section.

25797 **NAME**

25798 sigpause — remove a signal from the signal mask and suspend the thread

25799 **SYNOPSIS**

25800 EX #include <signal.h>

25801 int sigpause(int *sig*);

25802

25803 **DESCRIPTION**

25804 Refer to *signal()*.

25805 **CHANGE HISTORY**

25806 First released in Issue 4, Version 2.

25807 **Issue 5**

25808 Moved from X/OPEN UNIX extension to BASE.

25809 NAME

25810 sigpending — examine pending signals

25811 SYNOPSIS

25812 #include <signal.h>

25813 int sigpending(sigset_t *set);

25814 DESCRIPTION

25815 The *sigpending()* function stores, in the location referenced by the *set* argument, the set of signals
25816 that are blocked from delivery to the calling thread and that are pending on the process or the
25817 calling thread.

25818 RETURN VALUE

25819 Upon successful completion, *sigpending()* returns 0. Otherwise -1 is returned and *errno* is set to
25820 indicate the error.

25821 ERRORS

25822 No errors are defined.

25823 EXAMPLES

25824 None.

25825 APPLICATION USAGE

25826 None.

25827 FUTURE DIRECTIONS

25828 None.

25829 SEE ALSO

25830 *sigaddset()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*, *sigprocmask()*, <signal.h>.

25831 CHANGE HISTORY

25832 First released in Issue 3.

25833 Issue 5

25834 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

25835 **NAME**

25836 sigprocmask, pthread_sigmask — examine and change blocked signals

25837 **SYNOPSIS**

25838 #include <signal.h>

25839 int sigprocmask(int *how*, const sigset_t **set*, sigset_t **oset*);25840 int pthread_sigmask(int *how*, const sigset_t **set*, sigset_t **oset*);25841 **DESCRIPTION**25842 In a single-threaded process, the *sigprocmask()* function allows the calling process to examine or
25843 change (or both) the signal mask of the calling thread.25844 If the argument *set* is not a null pointer, it points to a set of signals to be used to change the
25845 currently blocked set.25846 The argument *how* indicates the way in which the set is changed, and consists of one of the
25847 following values:25848 SIG_BLOCK The resulting set will be the union of the current set and the signal set pointed
25849 to by *set*.25850 SIG_SETMASK The resulting set will be the signal set pointed to by *set*.25851 SIG_UNBLOCK The resulting set will be the intersection of the current set and the complement
25852 of the signal set pointed to by *set*.25853 If the argument *oset* is not a null pointer, the previous mask is stored in the location pointed to
25854 by *oset*. If *set* is a null pointer, the value of the argument *how* is not significant and the process'
25855 signal mask is unchanged; thus the call can be used to enquire about currently blocked signals.25856 If there are any pending unblocked signals after the call to *sigprocmask()*, at least one of those
25857 signals will be delivered before the call to *sigprocmask()* returns.25858 It is not possible to block those signals which cannot be ignored. This is enforced by the system
25859 without causing an error to be indicated.25860 If any of the SIGFPE, SIGILL, SIGSEGV or SIGBUS signals are generated while they are blocked,
25861 the result is undefined, unless the signal was generated by a function capable of sending a signal
25862 to a specific process or thread.25863 If *sigprocmask()* fails, the thread's signal mask is not changed.25864 The use of the *sigprocmask()* function is unspecified in a multi-threaded process.25865 The *pthread_sigmask()* function is used to examine or change (or both) the calling thread's signal
25866 mask, regardless of the number of threads in the process. The effect is the same as described for
25867 *sigprocmask()*, without the restriction that the call be made in a single-threaded process.25868 **RETURN VALUE**25869 Upon successful completion, *sigprocmask()* returns 0. Otherwise -1 is returned, *errno* is set to
25870 indicate the error and the process' signal mask will be unchanged.25871 Upon successful completion *pthread_sigmask()* returns 0; otherwise it returns the corresponding
25872 error number.25873 **ERRORS**25874 The *sigprocmask()* and *pthread_sigmask()* functions will fail if:25875 [EINVAL] The value of the *how* argument is not equal to one of the defined values.

25876	The <i>pthread_sigmask()</i> function will not return an error code of [EINTR].	
25877	EXAMPLES	
25878	None.	
25879	APPLICATION USAGE	
25880	None.	
25881	FUTURE DIRECTIONS	
25882	None.	
25883	SEE ALSO	
25884	<i>sigaction()</i> , <i>sigaddset()</i> , <i>sigdelset()</i> , <i>sigemptyset()</i> , <i>sigfillset()</i> , <i>sigismember()</i> , <i>sigpending()</i> , <i>squeue()</i> ,	
25885	<i>sigsuspend()</i> , <signal.h>.	
25886	CHANGE HISTORY	
25887	First released in Issue 3.	
25888	Entry included for alignment with the POSIX.1-1988 standard.	
25889	Issue 4	
25890	The following change is incorporated for alignment with the ISO POSIX-1 standard:	
25891	<ul style="list-style-type: none"> The type of the arguments <i>set</i> and <i>oset</i> are changed from sigset_t* to const sigset_t*. 	
25892	Another change is incorporated as follows:	
25893	<ul style="list-style-type: none"> The DESCRIPTION is changed to indicate that signals can also be generated by <i>raise()</i>. 	
25894	Issue 5	
25895	The DESCRIPTION is updated for alignment with the POSIX Threads Extension.	

25896 **NAME**25897 sigqueue — queue a signal to a process (**REALTIME**)25898 **SYNOPSIS**

25899 RT #include <sys/types.h>

25900 #include <signal.h>

25901 int sigqueue(pid_t pid, int signo, const union sigval value);

25902

25903 **DESCRIPTION**

25904 The *sigqueue()* function causes the signal specified by *signo* to be sent with the value specified by
 25905 *value* to the process specified by *pid*. If *signo* is zero (the null signal), error checking is performed
 25906 but no signal is actually sent. The null signal can be used to check the validity of *pid*.

25907 The conditions required for a process to have permission to queue a signal to another process
 25908 are the same as for the *kill()* function.

25909 The *sigqueue()* function returns immediately. If SA_SIGINFO is set for *signo* and if the resources
 25910 were available to queue the signal, the signal is queued and sent to the receiving process. If
 25911 SA_SIGINFO is not set for *signo*, then *signo* is sent at least once to the receiving process; it is
 25912 unspecified whether *value* will be sent to the receiving process as a result of this call.

25913 If the value of *pid* causes *signo* to be generated for the sending process, and if *signo* is not blocked
 25914 for the calling thread and if no other thread has *signo* unblocked or is waiting in a *sigwait()*
 25915 function for *signo*, either *signo* or at least the pending, unblocked signal will be delivered to the
 25916 calling thread before the *sigqueue()* function returns. Should any of multiple pending signals in
 25917 the range SIGRTMIN to SIGRTMAX be selected for delivery, it will be the lowest numbered one.
 25918 The selection order between realtime and non-realtime signals, or between multiple pending
 25919 non-realtime signals, is unspecified.

25920 **RETURN VALUE**

25921 Upon successful completion, the specified signal will have been queued, and the *sigqueue()*
 25922 function returns a value of zero. Otherwise, the function returns a value of -1 and sets *errno* to
 25923 indicate the error.

25924 **ERRORS**25925 The *sigqueue()* function will fail if:

25926 [EAGAIN] No resources available to queue the signal. The process has already queued
 25927 SIGQUEUE_MAX signals that are still pending at the receiver(s), or a system-
 25928 wide resource limit has been exceeded.

25929 [EINVAL] The value of the *signo* argument is an invalid or unsupported signal number.

25930 [ENOSYS] The function *sigqueue()* is not supported by this implementation.

25931 [EPERM] The process does not have the appropriate privilege to send the signal to the
 25932 receiving process.

25933 [ESRCH] The process *pid* does not exist.

25934 **EXAMPLES**

25935 None.

25936 **APPLICATION USAGE**

25937 None.

25938 **FUTURE DIRECTIONS**

25939 None.

25940 **SEE ALSO**

25941 `<signal.h>`.

25942 **CHANGE HISTORY**

25943 First released in Issue 5.

25944 Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

25945 **NAME**

25946 sigrelse, sigset — remove a signal from signal mask or modify signal disposition

25947 **SYNOPSIS**25948 EX `#include <signal.h>`25949 `int sigrelse(int sig);`25950 `void (*sigset(int sig, void (*disp)(int)))(int);`

25951

25952 **DESCRIPTION**25953 Refer to *signal()*.25954 **CHANGE HISTORY**

25955 First released in Issue 4, Version 2.

25956 **Issue 5**

25957 Moved from X/OPEN UNIX extension to BASE.

25958 **NAME**

25959 sigsetjmp — set jump point for a non-local goto

25960 **SYNOPSIS**

25961 #include <setjmp.h>

25962 int sigsetjmp(sigjmp_buf env, int savemask);

25963 **DESCRIPTION**

25964 A call to *sigsetjmp()* saves the calling environment in its *env* argument for later use by
 25965 *siglongjmp()*. It is unspecified whether *sigsetjmp()* is a macro or a function. If a macro definition
 25966 is suppressed in order to access an actual function, or a program defines an external identifier
 25967 with the name *sigsetjmp* the behaviour is undefined.

25968 If the value of the *savemask* argument is not 0, *sigsetjmp()* will also save the current signal mask
 25969 of the calling thread as part of the calling environment.

25970 All accessible objects have values as of the time *siglongjmp()* was called, except that the values of
 25971 objects of automatic storage duration which are local to the function containing the invocation of
 25972 the corresponding *sigsetjmp()* which do not have volatile-qualified type and which are changed
 25973 between the *sigsetjmp()* invocation and *siglongjmp()* call are indeterminate.

25974 An invocation of *sigsetjmp()* must appear in one of the following contexts only:

- 25975 • the entire controlling expression of a selection or iteration statement
- 25976 • one operand of a relational or equality operator with the other operand an integral constant
 25977 expression, with the resulting expression being the entire controlling expression of a
 25978 selection or iteration statement
- 25979 • the operand of a unary (!) operator with the resulting expression being the entire controlling
 25980 expression of a selection or iteration
- 25981 • the entire expression of an expression statement (possibly cast to **void**).

25982 **RETURN VALUE**

25983 If the return is from a successful direct invocation, *sigsetjmp()* returns 0. If the return is from a
 25984 call to *siglongjmp()*, *sigsetjmp()* returns a non-zero value.

25985 **ERRORS**

25986 No errors are defined.

25987 **EXAMPLES**

25988 None.

25989 **APPLICATION USAGE**

25990 The distinction between *setjmp()/longjmp()* and *sigsetjmp()/siglongjmp()* is only significant for
 25991 programs which use *sigaction()*, *sigprocmask()* or *sigsuspend()*.

25992 **FUTURE DIRECTIONS**

25993 None.

25994 **SEE ALSO**25995 *siglongjmp()*, *signal()*, *sigprocmask()*, *sigsuspend()*, <setjmp.h>.25996 **CHANGE HISTORY**

25997 First released in Issue 3.

25998 Entry included for alignment with the POSIX.1-1988 standard.

25999 **Issue 4**

26000 The following changes are incorporated in this issue:

- 26001 • The DESCRIPTION states that *sigsetjmp()* is a macro or a function. Issue 3 states that it is a
26002 macro. Warnings are also added about the suppression of a *sigsetjmp()* macro definition.
- 26003 • A statement is added to the DESCRIPTION about the accessibility of objects after a
26004 *siglongjmp()* call.
- 26005 • Text is added to the DESCRIPTION describing the contexts in which calls to *sigsetjmp()* are
26006 valid.

26007 **Issue 5**

26008 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

26009 **NAME**26010 sigstack — set and/or get alternate signal stack context (**LEGACY**)26011 **SYNOPSIS**26012 EX `#include <signal.h>`26013 `int sigstack(struct sigstack *ss, struct sigstack *oss);`

26014

26015 **DESCRIPTION**26016 The *sigstack()* function allows the calling process to indicate to the system an area of its address
26017 space to be used for processing signals received by the process.26018 If the *ss* argument is not a null pointer, it must point to a **sigstack** structure. The length of the
26019 application-supplied stack must be at least SIGSTKSZ bytes. If the alternate signal stack
26020 overflows, the resulting behaviour is undefined. (See APPLICATION USAGE below.)26021 • The value of the **ss_onstack** member indicates whether the process wants the system to use
26022 an alternate signal stack when delivering signals.26023 • The value of the **ss_sp** member indicates the desired location of the alternate signal stack
26024 area in the process' address space.26025 • If the *ss* argument is a null pointer, the current alternate signal stack context is not changed.26026 If the *oss* argument is not a null pointer, it points to a **sigstack** structure in which the current
26027 alternate signal stack context is placed. The value stored in the **ss_onstack** member of *oss* will be
26028 non-zero if the process is currently executing on the alternate signal stack. If the *oss* argument is
26029 a null pointer, the current alternate signal stack context is not returned.26030 When a signal's action indicates its handler should execute on the alternate signal stack
26031 (specified by calling *sigaction()*), the implementation checks to see if the process is currently
26032 executing on that stack. If the process is not currently executing on the alternate signal stack, the
26033 system arranges a switch to the alternate signal stack for the duration of the signal handler's
26034 execution.26035 After a successful call to one of the *exec* functions, there are no alternate signal stacks in the new
26036 process image.

26037 This interface need not be reentrant.

26038 **RETURN VALUE**26039 Upon successful completion, *sigstack()* returns 0. Otherwise, it returns -1 and sets *errno* to
26040 indicate the error.26041 **ERRORS**26042 The *sigstack()* function will fail if:

26043 [EPERM] An attempt was made to modify an active stack.

26044 **EXAMPLES**

26045 None.

26046 **APPLICATION USAGE**26047 A portable application, when being written or rewritten, should use *sigaltstack()* instead of
26048 *sigstack()*.26049 On some implementations, stack space is automatically extended as needed. On those
26050 implementations, automatic extension is typically not available for an alternate stack. If a signal
26051 stack overflows, the resulting behaviour of the process is undefined.

26052 The direction of stack growth is not indicated in the historical definition of **struct sigstack**. The
 26053 only way to portably establish a stack pointer is for the application to determine stack growth
 26054 direction, or to allocate a block of storage and set the stack pointer to the middle. The
 26055 implementation may assume that the size of the signal stack is SIGSTKSZ as found in
 26056 **<signal.h>**. An implementation that would like to specify a signal stack size other than
 26057 SIGSTKSZ should use *sigaltstack()*.

26058 Programs should not use *longjmp()* to leave a signal handler that is running on a stack
 26059 established with *sigstack()*. Doing so may disable future use of the signal stack. For abnormal
 26060 exit from a signal handler, *siglongjmp()*, *setcontext()* or *swapcontext()* may be used. These
 26061 functions fully support switching from one stack to another.

26062 The *sigstack()* function requires the application to have knowledge of the underlying system's
 26063 stack architecture. For this reason, *sigaltstack()* is recommended over this function.

26064 FUTURE DIRECTIONS

26065 None.

26066 SEE ALSO

26067 *exec*, *fork()*, *_longjmp()*, *longjmp()*, *setjmp()*, *sigaltstack()*, *siglongjmp()*, *sigsetjmp()*, **<signal.h>**.

26068 CHANGE HISTORY

26069 First released in Issue 4, Version 2.

26070 Issue 5

26071 Marked LEGACY.

26072 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

26073 **NAME**

26074 sigsuspend — wait for a signal

26075 **SYNOPSIS**

26076 #include <signal.h>

26077 int sigsuspend(const sigset_t *sigmask);

26078 **DESCRIPTION**

26079 The *sigsuspend()* function replaces the current signal mask of the calling thread with the set of
 26080 signals pointed to by *sigmask* and then suspends the thread until delivery of a signal whose
 26081 action is either to execute a signal-catching function or to terminate the process. This will not
 26082 cause any other signals that may have been pending on the process to become pending on the
 26083 thread.

26084 If the action is to terminate the process then *sigsuspend()* will never return. If the action is to
 26085 execute a signal-catching function, then *sigsuspend()* will return after the signal-catching
 26086 function returns, with the signal mask restored to the set that existed prior to the *sigsuspend()*
 26087 call.

26088 It is not possible to block signals that cannot be ignored. This is enforced by the system without
 26089 causing an error to be indicated.

26090 **RETURN VALUE**

26091 Since *sigsuspend()* suspends process execution indefinitely, there is no successful completion
 26092 return value. If a return occurs, -1 is returned and *errno* is set to indicate the error.

26093 **ERRORS**26094 The *sigsuspend()* function will fail if:

26095 [EINTR] A signal is caught by the calling process and control is returned from the
 26096 signal-catching function.

26097 **EXAMPLES**

26098 None.

26099 **APPLICATION USAGE**

26100 An interpretation request has been filed with IEEE PASC concerning whether *sigsuspend()*
 26101 suspends process execution or suspends thread execution. The wording here matches the
 26102 description of this interface specified by the ISO POSIX-1 standard.

26103 **FUTURE DIRECTIONS**

26104 None.

26105 **SEE ALSO**26106 *pause()*, *sigaction()*, *sigaddset()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, <signal.h>.26107 **CHANGE HISTORY**

26108 First released in Issue 3.

26109 Entry included for alignment with the POSIX.1-1988 standard.

26110 **Issue 4**

26111 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 26112 • The type of the argument *sigmask* is changed from **sigset_t*** to type **const sigset_t***.

26113 Another change is incorporated as follows:

- 26114 • The term “signal handler” is changed to “signal-catching function”.

26115 **Issue 5**

26116 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

|

26117 **NAME**

26118 sigwait — wait for queued signals

26119 **SYNOPSIS**

26120 #include <signal.h>

26121 int sigwait(const sigset_t *set, int *sig);

26122 **DESCRIPTION**

26123 The *sigwait()* function selects a pending signal from *set*, atomically clears it from the system's set
 26124 of pending signals, and returns that signal number in the location referenced by *sig*. If prior to
 26125 the call to *sigwait()* there are multiple pending instances of a single signal number, it is
 26126 implementation-dependent whether upon successful return there are any remaining pending
 26127 **RT** signals for that signal number. If the implementation supports queued signals and there are
 26128 multiple signals queued for the signal number selected, the first such queued signal causes a
 26129 return from *sigwait()* and the remainder remain queued. If no signal in *set* is pending at the time
 26130 of the call, the thread is suspended until one or more becomes pending. The signals defined by
 26131 *set* will be blocked at the time of the call to *sigwait()*; otherwise the behaviour is undefined.
 26132 The effect of *sigwait()* on the signal actions for the signals in *set* is unspecified.

26133 If more than one thread is using *sigwait()* to wait for the same signal, no more than one of these
 26134 threads will return from *sigwait()* with the signal number. Which thread returns from *sigwait()*
 26135 if more than a single thread is waiting is unspecified.

26136 Should any of the multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it
 26137 shall be the lowest numbered one. The selection order between realtime and non-realtime
 26138 signals, or between multiple pending non-realtime signals, is unspecified.

26139 **RETURN VALUE**

26140 Upon successful completion, *sigwait()* stores the signal number of the received signal at the
 26141 location referenced by *sig* and returns zero. Otherwise, an error number is returned to indicate
 26142 the error.

26143 **ERRORS**26144 The *sigwait()* function may fail if:26145 [EINVAL] The *set* argument contains an invalid or unsupported signal number.26146 **EXAMPLES**

26147 None.

26148 **APPLICATION USAGE**

26149 None.

26150 **FUTURE DIRECTIONS**

26151 None.

26152 **SEE ALSO**

26153 *pause()*, *pthread_sigmask()*, *sigaction()*, <signal.h>, *sigpending()*, *sigsuspend()*, *sigwaitinfo()*,
 26154 <time.h>.

26155 **CHANGE HISTORY**

26156 First released in Issue 5.

26157 Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

26158 **NAME**26159 sigwaitinfo, sigtimedwait — wait for queued signals (**REALTIME**)26160 **SYNOPSIS**26161 RT

```
#include <signal.h>
```

```
26162 int sigwaitinfo(const sigset_t *set, siginfo *info);
26163 int sigtimedwait(const sigset_t *set, siginfo_t *info,
26164                 const struct timespec *timeout);
26165
```

26166 **DESCRIPTION**

26167 The function *sigwaitinfo()* selects the pending signal from the set specified by *set*. Should any of
 26168 multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it will be the lowest
 26169 numbered one. The selection order between realtime and non-realtime signals, or between
 26170 multiple pending non-realtime signals, is unspecified. If no signal in *set* is pending at the time of
 26171 the call, the calling thread is suspended until one or more signals in *set* become pending or until
 26172 it is interrupted by an unblocked, caught signal.

26173 The function *sigwaitinfo()* behaves the same as the *sigwait()* function if the *info* argument is
 26174 NULL. If the *info* argument is non-NULL, the *sigwaitinfo()* function behaves the same as
 26175 *sigwait()* except that the selected signal number is stored in the *si_signo* member, and the cause
 26176 of the signal is stored in the *si_code* member. If any value is queued to the selected signal, the
 26177 first such queued value is dequeued and, if the *info* argument is non-NULL, the value is stored in
 26178 the *si_value* member of *info*. The system resource used to queue the signal will be released and
 26179 made available to queue other signals. If no value is queued, the content of the *si_value* member
 26180 is undefined. If no further signals are queued for the selected signal, the pending indication for
 26181 that signal will be reset.

26182 The function *sigtimedwait()* behaves the same as *sigwaitinfo()* except that if none of the signals
 26183 specified by *set* are pending, *sigtimedwait()* waits for the time interval specified in the **timespec**
 26184 structure referenced by *timeout*. If the **timespec** structure pointed to by *timeout* is zero-valued
 26185 and if none of the signals specified by *set* are pending, then *sigtimedwait()* returns immediately
 26186 with an error. If *timeout* is the NULL pointer, the behaviour is unspecified.

26187 **RETURN VALUE**

26188 Upon successful completion (that is, one of the signals specified by *set* is pending or is
 26189 generated) *sigwaitinfo()* and *sigtimedwait()* will return the selected signal number. Otherwise,
 26190 the function returns a value of -1 and sets *errno* to indicate the error.

26191 **ERRORS**26192 The *sigwaitinfo()* and *sigtimedwait()* functions will fail if:

26193 [ENOSYS] The functions *sigwaitinfo()* and *sigtimedwait()* are not supported by this
 26194 implementation.

26195 The *sigtimedwait()* function will also fail if:

26196 [EAGAIN] No signal specified by *set* was generated within the specified timeout period.

26197 The *sigwaitinfo()* and *sigtimedwait()* functions may fail if:

26198 [EINTR] The wait was interrupted by an unblocked, caught signal. It will be
 26199 documented in system documentation whether this error will cause these
 26200 functions to fail.

26201 The *sigtimedwait()* function may also fail if:

26202 [EINVAL] The *timeout* argument specified a *tv_nsec* value less than zero or greater than

26203 or equal to 1000 million.

26204 An implementation only checks for this error if no signal is pending in *set* and it is necessary to

26205 wait.

26206 **EXAMPLES**

26207 None.

26208 **APPLICATION USAGE**

26209 None.

26210 **FUTURE DIRECTIONS**

26211 None.

26212 **SEE ALSO**

26213 *pause()*, *pthread_sigmask()*, *sigaction()*, **<signal.h>**, *sigpending()*, *sigsuspend()*, *sigwait()*, **<time.h>**.

26214 **CHANGE HISTORY**

26215 First released in Issue 5.

26216 Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

26217 **NAME**

26218 sin — sine function

26219 **SYNOPSIS**

26220 #include <math.h>

26221 double sin(double x);

26222 **DESCRIPTION**26223 The *sin()* function computes the sine of its argument *x*, measured in radians.26224 An application wishing to check for error situations should set *errno* to 0 before calling *sin()*. If
26225 *errno* is non-zero on return, or the return value is NaN, an error has occurred.26226 The *sin()* function may lose accuracy when its argument is far from 0.0 .26227 **RETURN VALUE**26228 Upon successful completion, *sin()* returns the sine of *x*.26229 EX If *x* is NaN, NaN is returned and *errno* may be set to [EDOM].26230 EX If *x* is $\pm\text{Inf}$, either 0.0 is returned and *errno* is set to [EDOM], or NaN is returned and *errno* may be
26231 set to [EDOM].26232 If the correct result would cause underflow, 0.0 is returned and *errno* may be set to [ERANGE].26233 **ERRORS**26234 The *sin()* function may fail if:26235 EX [EDOM] The value of *x* is NaN, or *x* is $\pm\text{Inf}$.

26236 [ERANGE] The result underflows.

26237 EX No other errors will occur.

26238 **EXAMPLES**

26239 None.

26240 **APPLICATION USAGE**

26241 None.

26242 **FUTURE DIRECTIONS**

26243 None.

26244 **SEE ALSO**26245 *asin()*, *isnan()*, <math.h>.26246 **CHANGE HISTORY**

26247 First released in Issue 1.

26248 Derived from Issue 1 of the SVID.

26249 **Issue 4**

26250 The following changes are incorporated in this issue:

- 26251 • Removed references to *matherr()*.
- 26252 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
- 26253 the ISO C standard and to rationalise error handling in the mathematics functions.
- 26254 • The return value specified for [EDOM] is marked as an extension.

26255 **Issue 5**

26256 The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes
26257 in previous issues.

26258 **NAME**

26259 sinh — hyperbolic sine function

26260 **SYNOPSIS**

26261 #include <math.h>

26262 double sinh(double x);

26263 **DESCRIPTION**26264 The *sinh()* function computes the hyperbolic sine of *x*.

26265 An application wishing to check for error situations should set *errno* to 0 before calling *sinh()*. If
 26266 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

26267 **RETURN VALUE**26268 Upon successful completion, *sinh()* returns the hyperbolic sine of *x*.26269 If the result would cause an overflow, $\pm\text{HUGE_VAL}$ is returned and *errno* is set to [ERANGE].26270 If the result would cause underflow, 0.0 is returned and *errno* may be set to [ERANGE].26271 EX If *x* is NaN, NaN is returned and *errno* may be set to [EDOM].26272 **ERRORS**26273 The *sinh()* function will fail if:

26274 [ERANGE] The result would cause overflow.

26275 The *sinh()* function may fail if:26276 EX [EDOM] The value of *x* is NaN.

26277 [ERANGE] The result would cause underflow.

26278 EX No other errors will occur.

26279 **EXAMPLES**

26280 None.

26281 **APPLICATION USAGE**

26282 None.

26283 **FUTURE DIRECTIONS**

26284 None.

26285 **SEE ALSO**26286 *asinh()*, *cosh()*, *isnan()*, *tanh()*, <math.h>.26287 **CHANGE HISTORY**

26288 First released in Issue 1.

26289 Derived from Issue 1 of the SVID.

26290 **Issue 4**

26291 The following changes are incorporated in this issue:

- 26292 • Removed references to *matherr()*.
- 26293 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
 26294 the ISO C standard and to rationalise error handling in the mathematics functions.
- 26295 • The return value specified for [EDOM] is marked as an extension.

26296 **Issue 5**

26297 The DESCRIPTION is updated to indicate how an application should check for an error. This
26298 text was previously published in the APPLICATION USAGE section.

26299 **NAME**

26300 sleep — suspend execution for an interval of time

26301 **SYNOPSIS**

26302 #include <unistd.h>

26303 unsigned int sleep(unsigned int *seconds*);26304 **DESCRIPTION**

26305 The *sleep()* function will cause the calling thread to be suspended from execution until either the
 26306 number of real-time seconds specified by the argument *seconds* has elapsed or a signal is
 26307 delivered to the calling thread and its action is to invoke a signal-catching function or to
 26308 terminate the process. The suspension time may be longer than requested due to the scheduling
 26309 of other activity by the system.

26310 If a SIGALRM signal is generated for the calling process during execution of *sleep()* and if the
 26311 SIGALRM signal is being ignored or blocked from delivery, it is unspecified whether *sleep()*
 26312 returns when the SIGALRM signal is scheduled. If the signal is being blocked, it is also
 26313 unspecified whether it remains pending after *sleep()* returns or it is discarded.

26314 If a SIGALRM signal is generated for the calling process during execution of *sleep()*, except as a
 26315 result of a prior call to *alarm()*, and if the SIGALRM signal is not being ignored or blocked from
 26316 delivery, it is unspecified whether that signal has any effect other than causing *sleep()* to return.

26317 If a signal-catching function interrupts *sleep()* and examines or changes either the time a
 26318 SIGALRM is scheduled to be generated, the action associated with the SIGALRM signal, or
 26319 whether the SIGALRM signal is blocked from delivery, the results are unspecified.

26320 If a signal-catching function interrupts *sleep()* and calls *siglongjmp()* or *longjmp()* to restore an
 26321 environment saved prior to the *sleep()* call, the action associated with the SIGALRM signal and
 26322 the time at which a SIGALRM signal is scheduled to be generated are unspecified. It is also
 26323 unspecified whether the SIGALRM signal is blocked, unless the process' signal mask is restored
 26324 as part of the environment.

26325 EX Interactions between *sleep()* and any of *setitimer()*, *ualarm()* or *usleep()* are unspecified.

26326 **RETURN VALUE**

26327 If *sleep()* returns because the requested time has elapsed, the value returned will be 0. If *sleep()*
 26328 returns because of premature arousal due to delivery of a signal, the return value will be the
 26329 “unslept” amount (the requested time minus the time actually slept) in seconds.

26330 **ERRORS**

26331 No errors are defined.

26332 **EXAMPLES**

26333 None.

26334 **APPLICATION USAGE**

26335 None.

26336 **FUTURE DIRECTIONS**

26337 None.

26338 **SEE ALSO**26339 *alarm()*, *getitimer()*, *nanosleep()*, *pause()*, *sigaction()*, *sigsetjmp()*, *ualarm()*, *usleep()*, <unistd.h>.26340 **CHANGE HISTORY**

26341 First released in Issue 1.

26342 Derived from Issue 1 of the SVID.

26343 **Issue 4**

26344 The following change is incorporated in this issue:

- 26345 • The <**unistd.h**> header is added to the SYNOPSIS section.

26346 **Issue 4, Version 2**

26347 The DESCRIPTION is updated to indicate possible interactions with the *setitimer()*, *ualarm()* and
26348 *usleep()* functions.

26349 **Issue 5**

26350 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

26351 **NAME**

26352 sprintf, snprintf — print formatted output

26353 **SYNOPSIS**

26354 #include <stdio.h>

26355 EX `int snprintf(char *s, size_t n, const char *format, /* args */ ...);`26356 `int sprintf(char *s, const char *format, ...);`26357 **DESCRIPTION**26358 Refer to *fprintf()*.26359 **CHANGE HISTORY**

26360 First released in Issue 1.

26361 Derived from Issue 1 of the SVID.

26362 **Issue 4**

26363 The following change is incorporated for alignment with the ISO C standard:

- 26364 • The type of argument
- format*
- is changed from
- char ***
- to
- const char ***
- .

26365 Another change is incorporated as follows:

- 26366 • The detail for this function is now in
- fprintf()*
- instead of
- printf()*
- .

26367 **Issue 5**26368 The *snprintf()* function is new in Issue 5.

26369 **NAME**

26370 sqrt — square root function

26371 **SYNOPSIS**

26372 #include <math.h>

26373 double sqrt(double x);

26374 **DESCRIPTION**26375 The *sqrt()* function computes the square root of x , \sqrt{x} .

26376 An application wishing to check for error situations should set *errno* to 0 before calling *sqrt()*. If
 26377 *errno* is non-zero on return, or the return value is NaN, an error has occurred.

26378 **RETURN VALUE**26379 Upon successful completion, *sqrt()* returns the square root of x .26380 EX If x is NaN, NaN is returned and *errno* may be set to [EDOM].26381 EX If x is negative, 0.0 or NaN is returned and *errno* is set to [EDOM].26382 **ERRORS**26383 The *sqrt()* function will fail if:26384 [EDOM] The value of x is negative.26385 The *sqrt()* function may fail if:26386 EX [EDOM] The value of x is NaN.

26387 EX No other errors will occur.

26388 **EXAMPLES**

26389 None.

26390 **APPLICATION USAGE**

26391 None.

26392 **FUTURE DIRECTIONS**

26393 None.

26394 **SEE ALSO**26395 *isnan()*, <math.h>, <stdio.h>.26396 **CHANGE HISTORY**

26397 First released in Issue 1.

26398 Derived from Issue 1 of the SVID.

26399 **Issue 4**

26400 The following changes are incorporated in this issue:

- 26401 • Removed references to *matherr()*.
- 26402 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
 26403 the ISO C standard and to rationalise error handling in the mathematics functions.
- 26404 • The return value specified for [EDOM] is marked as an extension.

26405 **Issue 5**

26406 The DESCRIPTION is updated to indicate how an application should check for an error. This
 26407 text was previously published in the APPLICATION USAGE section.

26408 **NAME**

26409 srand — seed simple pseudo-random number generator

26410 **SYNOPSIS**

26411 #include <stdlib.h>

26412 void srand(unsigned int *seed*);

26413 **DESCRIPTION**

26414 Refer to *rand()*.

26415 **CHANGE HISTORY**

26416 First released in Issue 1.

26417 Derived from Issue 1 of the SVID.

26418 **Issue 4**

26419 The following change is incorporated for alignment with the ISO C standard:

- 26420
 - The argument *seed* is explicitly defined as **unsigned int**.

26421 **NAME**

26422 srand48 — seed uniformly distributed double-precision pseudo-random number generator

26423 **SYNOPSIS**

26424 EX #include <stdlib.h>

26425 void srand48(long int *seedval*);

26426

26427 **DESCRIPTION**26428 Refer to *drand48()*.26429 **CHANGE HISTORY**

26430 First released in Issue 1.

26431 Derived from Issue 1 of the SVID.

26432 **Issue 4**

26433 The following change is incorporated in this issue:

- 26434
- The header <stdlib.h> is added to the SYNOPSIS section.

26435 **NAME**

26436 srandom — seed pseudorandom number generator

26437 **SYNOPSIS**

26438 EX #include <stdlib.h>

26439 void srandom(unsigned int *seed*);

26440

26441 **DESCRIPTION**26442 Refer to *initstate()*.26443 **CHANGE HISTORY**

26444 First released in Issue 4, Version 2.

26445 **Issue 5**

26446 Moved from X/OPEN UNIX extension to BASE.

26447 NAME

26448 scanf — convert formatted input

26449 SYNOPSIS

26450 #include <stdio.h>

26451 int sscanf(const char *s, const char *format, ...);

26452 DESCRIPTION

26453 Refer to *fscanf()*.

26454 CHANGE HISTORY

26455 First released in Issue 1.

26456 Derived from Issue 1 of the SVID.

26457 Issue 4

26458 The following change is incorporated for alignment with the ISO C standard:

- 26459 • The type of arguments *s* and *format* is changed from **char *** to **const char ***.

26460 Another change is incorporated as follows:

- 26461 • The detail for this function is now in *fscanf()* instead of *scanf()*.

26462 **NAME**

26463 stat — get file status

26464 **SYNOPSIS**

26465 OH #include <sys/types.h>

26466 #include <sys/stat.h>

26467 int stat(const char *path, struct stat *buf);

26468 **DESCRIPTION**

26469 The *stat()* function obtains information about the named file and writes it to the area pointed to
 26470 by the *buf* argument. The *path* argument points to a pathname naming a file. Read, write or
 26471 execute permission of the named file is not required, but all directories listed in the pathname
 26472 leading to the file must be searchable. An implementation that provides additional or alternate
 26473 file access control mechanisms may, under implementation-dependent conditions, cause *stat()*
 26474 to fail. In particular, the system may deny the existence of the file specified by *path*.

26475 The *buf* argument is a pointer to a *stat* structure, as defined in the header <sys/stat.h>, into which
 26476 information is placed concerning the file.

26477 The *stat()* function updates any time-related fields (as described in the definition of **File Times**
 26478 **Update** in the **XBD** specification), before writing into the *stat* structure.

26479 The structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime* and *st_mtime* will
 26480 have meaningful values for all file types defined in this document. The value of the member
 26481 *st_nlink* will be set to the number of links to the file.

26482 **RETURN VALUE**

26483 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate
 26484 the error.

26485 **ERRORS**26486 The *stat()* function will fail if:

26487 [EACCES] Search permission is denied for a component of the path prefix.

26488 EX [EIO] An error occurred while reading from the file system.

26489 EX [ELOOP] Too many symbolic links were encountered in resolving *path*.

26490 FIPS [ENAMETOOLONG]

26491 The length of the *path* argument exceeds {PATH_MAX} or a pathname
 26492 component is longer than {NAME_MAX}.

26493 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

26494 [ENOTDIR] A component of the path prefix is not a directory.

26495 EX [E_OVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file
 26496 serial number cannot be represented correctly in the structure pointed to by
 26497 *buf*.

26498 EX The *stat()* function may fail if:

26499 EX [ENAMETOOLONG]

26500 Pathname resolution of a symbolic link produced an intermediate result
 26501 whose length exceeds {PATH_MAX}.

26502 [E_OVERFLOW] A value to be stored would overflow one of the members of the *stat* structure.

26503 **EXAMPLES**

26504 None.

26505 **APPLICATION USAGE**

26506 None.

26507 **FUTURE DIRECTIONS**

26508 None.

26509 **SEE ALSO**26510 *fstat()*, *lstat()*, <sys/stat.h>, <sys/types.h>.26511 **CHANGE HISTORY**

26512 First released in Issue 1.

26513 Derived from Issue 1 of the SVID.

26514 **Issue 4**

26515 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- 26516 • The type of argument *path* is changed from **char *** to **const char ***.
- 26517 • In the DESCRIPTION (a) statements indicating the purpose of this interface and a paragraph
- 26518 defining the contents of **stat** structure members are added, and (b) the words “extended
- 26519 security controls” are replaced by “additional or alternate file access control mechanisms”.

26520 The following change is incorporated for alignment with the FIPS requirements:

- 26521 • In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a
- 26522 pathname component is larger than {NAME_MAX} is now defined as mandatory and marked
- 26523 as an extension.

26524 Another change is incorporated as follows:

- 26525 • The <sys/types.h> header is now marked as optional (OH); this header need not be included
- 26526 on XSI-conformant systems.

26527 **Issue 4, Version 2**

26528 The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- 26529 • In the mandatory section, [EIO] is added to indicate that a physical I/O error has occurred,
- 26530 and [ELOOP] to indicate that too many symbolic links were encountered during pathname
- 26531 resolution.
- 26532 • In the optional section, a second [ENAMETOOLONG] condition is defined that may report
- 26533 excessive length of an intermediate result of pathname resolution of a symbolic link.
- 26534 • In the optional section, [EOVERFLOW] is added to indicate that a value to be stored in a
- 26535 member of the **stat** structure would cause overflow.

26536 **Issue 5**

26537 Large File Summit extensions added.

26538 **NAME**

26539 statvfs — get file system information

26540 **SYNOPSIS**

26541 EX #include <sys/statvfs.h>

26542 int statvfs(const char *path, struct statvfs *buf);

26543

26544 **DESCRIPTION**26545 Refer to *fstatvfs()*.26546 **CHANGE HISTORY**

26547 First released in Issue 4, Version 2.

26548 **Issue 5**

26549 Moved from X/OPEN UNIX extension to BASE.

26550 **NAME**

26551 stderr, stdin, stdout — standard I/O streams

26552 **SYNOPSIS**

26553 #include <stdio.h>

26554 extern FILE **stderr*, **stdin*, **stdout*;

26555 **DESCRIPTION**

26556 A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type
 26557 **FILE**. The *fopen()* function creates certain descriptive data for a stream and returns a pointer to
 26558 designate the stream in all further transactions. Normally, there are three open streams with
 26559 constant pointers declared in the <**stdio.h**> header and associated with the standard open files.

26560 At program startup, three streams are predefined and need not be opened explicitly: *standard*
 26561 *input* (for reading conventional input), *standard output* (for writing conventional output) and
 26562 *standard error* (for writing diagnostic output). When opened, the standard error stream is not
 26563 fully buffered; the standard input and standard output streams are fully buffered if and only if
 26564 the stream can be determined not to refer to an interactive device.

26565 The following symbolic values in <**unistd.h**> define the file descriptors that will be associated
 26566 with the C-language *stdin*, *stdout* and *stderr* when the application is started:

26567 STDIN_FILENO Standard input value, *stdin*. Its value is 0.

26568 STDOUT_FILENO Standard output value, *stdout*. Its value is 1.

26569 STDERR_FILENO Standard error value, *stderr*. Its value is 2.

26570 **RETURN VALUE**

26571 None.

26572 **ERRORS**

26573 No errors are defined.

26574 **EXAMPLES**

26575 None.

26576 **APPLICATION USAGE**

26577 None.

26578 **FUTURE DIRECTIONS**

26579 None.

26580 **SEE ALSO**

26581 *fclose()*, *feof()*, *ferror()*, *fileno()*, *fopen()*, *fread()*, *fseek()*, *getc()*, *gets()*, *popen()*, *printf()*, *putc()*,
 26582 *puts()*, *read()*, *scanf()*, *setbuf()*, *setvbuf()*, *tmpfile()*, *ungetc()*, *vprintf()*, <**stdio.h**>, <**unistd.h**>.

26583 **CHANGE HISTORY**

26584 First released in Issue 1.

26585 **NAME**26586 step — pattern match with regular expressions (**LEGACY**)26587 **SYNOPSIS**

26588 EX #include <regex.h>

26589 int step(const char *string, const char *expbuf);

26590

26591 **DESCRIPTION**26592 Refer to *regex()*.26593 **CHANGE HISTORY**

26594 First released in Issue 2.

26595 Derived from Issue 2 of the SVID.

26596 **Issue 4**

26597 The following changes are incorporated in this issue:

- 26598 • The <**regex.h**> header is added to the SYNOPSIS section.
- 26599 • The type of arguments *string* and *expbuf* are changed from **char *** to **const char ***.
- 26600 • The interface is marked TO BE WITHDRAWN, because improved functionality is now
- 26601 provided by interfaces introduced for alignment with the ISO POSIX-2 standard.

26602 **Issue 5**

26603 Marked LEGACY.

26604 **NAME**

26605 strcasecmp, strncasecmp — case-insensitive string comparisons

26606 **SYNOPSIS**26607 EX `#include <strings.h>`26608 `int strcasecmp(const char *s1, const char *s2);`26609 `int strncasecmp(const char *s1, const char *s2, size_t n);`

26610

26611 **DESCRIPTION**

26612 The *strcasecmp()* function compares, while ignoring differences in case, the string pointed to by
26613 *s1* to the string pointed to by *s2*. The *strncasecmp()* function compares, while ignoring
26614 differences in case, not more than *n* bytes from the string pointed to by *s1* to the string pointed to
26615 by *s2*.

26616 In the POSIX locale, *strcasecmp()* and *strncasecmp()* do upper to lower conversions, then a byte
26617 comparison. The results are unspecified in other locales.

26618 **RETURN VALUE**

26619 Upon completion, *strcasecmp()* returns an integer greater than, equal to or less than 0, if the
26620 string pointed to by *s1* is, ignoring case, greater than, equal to or less than the string pointed to
26621 by *s2* respectively.

26622 Upon successful completion, *strncasecmp()* returns an integer greater than, equal to or less than
26623 0, if the possibly null-terminated array pointed to by *s1* is, ignoring case, greater than, equal to or
26624 less than the possibly null-terminated array pointed to by *s2* respectively.

26625 **ERRORS**

26626 No errors are defined.

26627 **EXAMPLES**

26628 None.

26629 **APPLICATION USAGE**

26630 None.

26631 **FUTURE DIRECTIONS**

26632 None.

26633 **SEE ALSO**26634 `<strings.h>`.26635 **CHANGE HISTORY**

26636 First released in Issue 4, Version 2.

26637 **Issue 5**

26638 Moved from X/OPEN UNIX extension to BASE.

26639 **NAME**

26640 strcat — concatenate two strings

26641 **SYNOPSIS**

26642 #include <string.h>

26643 char *strcat(char *s1, const char *s2);

26644 **DESCRIPTION**

26645 The *strcat()* function appends a copy of the string pointed to by *s2* (including the terminating null byte) to the end of the string pointed to by *s1*. The initial byte of *s2* overwrites the null byte at the end of *s1*. If copying takes place between objects that overlap, the behaviour is undefined.

26648 **RETURN VALUE**26649 The *strcat()* function returns *s1*; no return value is reserved to indicate an error.26650 **ERRORS**

26651 No errors are defined.

26652 **EXAMPLES**

26653 None.

26654 **APPLICATION USAGE**

26655 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.

26657 **FUTURE DIRECTIONS**

26658 None.

26659 **SEE ALSO**26660 *strncat()*, <string.h>.26661 **CHANGE HISTORY**

26662 First released in Issue 1.

26663 Derived from Issue 1 of the SVID.

26664 **Issue 4**

26665 The following change is incorporated for alignment with the ISO C standard:

- 26666 • The type of argument *s2* is changed from **char *** to **const char ***.

26667 Other changes are incorporated as follows:

- 26668 • The DESCRIPTION is changed to make it clear that the function manipulates bytes rather than (possibly multi-byte) characters.

26670 NAME

26671 strchr — string scanning operation

26672 SYNOPSIS

26673 #include <string.h>

26674 char *strchr(const char *s, int c);

26675 DESCRIPTION

26676 The *strchr()* function locates the first occurrence of *c* (converted to an **unsigned char**) in the
26677 string pointed to by *s*. The terminating null byte is considered to be part of the string.

26678 RETURN VALUE

26679 Upon completion, *strchr()* returns a pointer to the byte, or a null pointer if the byte was not
26680 found.

26681 ERRORS

26682 No errors are defined.

26683 EXAMPLES

26684 None.

26685 APPLICATION USAGE

26686 None.

26687 FUTURE DIRECTIONS

26688 None.

26689 SEE ALSO

26690 *strrchr()*, <string.h>.

26691 CHANGE HISTORY

26692 First released in Issue 1.

26693 Derived from Issue 1 of the SVID.

26694 Issue 4

26695 The following change is incorporated for alignment with the ISO C standard:

- 26696 • The type of argument *s* is changed from **char *** to **const char ***.

26697 Other changes are incorporated as follows:

- 26698 • The DESCRIPTION and RETURN VALUE sections are changed to make it clear that the
26699 function manipulates bytes rather than (possibly multi-byte) characters.
- 26700 • The APPLICATION USAGE section is removed.

26701 **NAME**

26702 strcmp — compare two strings

26703 **SYNOPSIS**

26704 #include <string.h>

26705 int strcmp(const char *s1, const char *s2);

26706 **DESCRIPTION**26707 The *strcmp()* function compares the string pointed to by *s1* to the string pointed to by *s2*.

26708 The sign of a non-zero return value is determined by the sign of the difference between the
 26709 values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the strings
 26710 being compared.

26711 **RETURN VALUE**

26712 Upon completion, *strcmp()* returns an integer greater than, equal to or less than 0, if the string
 26713 pointed to by *s1* is greater than, equal to or less than the string pointed to by *s2* respectively.

26714 **ERRORS**

26715 No errors are defined.

26716 **EXAMPLES**

26717 None.

26718 **APPLICATION USAGE**

26719 None.

26720 **FUTURE DIRECTIONS**

26721 None.

26722 **SEE ALSO**26723 *strncmp()*, <string.h>.26724 **CHANGE HISTORY**

26725 First released in Issue 1.

26726 Derived from Issue 1 of the SVID.

26727 **Issue 4**

26728 The following change is incorporated for alignment with the ISO C standard:

- 26729 • The type of arguments *s1* and *s2* is changed from **char *** to **const char ***.

26730 Another change is incorporated as follows:

- 26731 • The DESCRIPTION is changed to make it clear that *strcmp()* compares bytes rather than
 26732 (possibly multi-byte) characters.

26733 **NAME**

26734 strcoll — string comparison using collating information

26735 **SYNOPSIS**

26736 #include <string.h>

26737 int strcoll(const char *s1, const char *s2);

26738 **DESCRIPTION**26739 The *strcoll()* function compares 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.26741 The *strcoll()* function will not change the setting of **errno** if successful.26742 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*.26744 **RETURN VALUE**26745 Upon successful completion, *strcoll()* returns 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.26749 **ERRORS**26750 The *strcoll()* function may fail if:

26751 EX	[EINVAL]	The <i>s1</i> or <i>s2</i> arguments contain characters outside the domain of the collating sequence.
26752		

26753 **EXAMPLES**

26754 None.

26755 **APPLICATION USAGE**26756 The *strxfrm()* and *strcmp()* functions should be used for sorting large lists.26757 **FUTURE DIRECTIONS**

26758 None.

26759 **SEE ALSO**26760 *strcmp()*, *strxfrm()*, <string.h>.26761 **CHANGE HISTORY**

26762 First released in Issue 3.

26763 **Issue 4**

26764 The following changes are incorporated for alignment with the ISO C standard:

- 26765 • The function is no longer marked as an extension.
- 26766 • The type of arguments *s1* and *s2* are changed from **char *** to **const char ***.

26767 Other changes are incorporated as follows:

- 26768 • A paragraph describing how the sign of the return value should be determined is removed from the DESCRIPTION.
- 26770 • The [EINVAL] error is marked as an extension.

26771 **Issue 5**26772 The DESCRIPTION is updated to indicate that **errno** will not be changed if the function is successful.

26774 **NAME**

26775 strcpy — copy a string

26776 **SYNOPSIS**

26777 #include <string.h>

26778 char *strcpy(char *s1, const char *s2);

26779 **DESCRIPTION**

26780 The *strcpy()* function copies the string pointed to by *s2* (including the terminating null byte) into
 26781 the array pointed to by *s1*. If copying takes place between objects that overlap, the behaviour is
 26782 undefined.

26783 **RETURN VALUE**26784 The *strcpy()* function returns *s1*; no return value is reserved to indicate an error.26785 **ERRORS**

26786 No errors are defined.

26787 **EXAMPLES**

26788 None.

26789 **APPLICATION USAGE**

26790 Character movement is performed differently in different implementations. Thus overlapping
 26791 moves may yield surprises.

26792 This issue is aligned with the ISO C standard; this does not affect compatibility with XPG3
 26793 applications. Reliable error detection by this function was never guaranteed.

26794 **FUTURE DIRECTIONS**

26795 None.

26796 **SEE ALSO**

26797 strncpy(), <string.h>.

26798 **CHANGE HISTORY**

26799 First released in Issue 1.

26800 Derived from Issue 1 of the SVID.

26801 **Issue 4**

26802 The following change is incorporated for alignment with the ISO C standard:

- 26803 • The type of argument *s2* is changed from **char *** to **const char ***.

26804 Other changes are incorporated as follows:

- 26805 • The **DESCRIPTION** is changed to make it clear that the function manipulates bytes rather
 26806 than (possibly multi-byte) characters.

26807 **NAME**

26808 strcspn — get length of a complementary substring

26809 **SYNOPSIS**

26810 #include <string.h>

26811 size_t strcspn(const char *s1, const char *s2);

26812 **DESCRIPTION**

26813 The *strcspn()* function computes the length of the maximum initial segment of the string pointed
26814 to by *s1* which consists entirely of bytes *not* from the string pointed to by *s2*.

26815 **RETURN VALUE**

26816 The *strcspn()* function returns the length of *s1*; no return value is reserved to indicate an error.

26817 **ERRORS**

26818 No errors are defined.

26819 **EXAMPLES**

26820 None.

26821 **APPLICATION USAGE**

26822 None.

26823 **FUTURE DIRECTIONS**

26824 None.

26825 **SEE ALSO**

26826 *strspn()*, <string.h>.

26827 **CHANGE HISTORY**

26828 First released in Issue 1.

26829 Derived from Issue 1 of the SVID.

26830 **Issue 4**

26831 The following change is incorporated for alignment with the ISO C standard:

- 26832 • The type of arguments *s1* and *s2* is changed from **char *** to **const char ***.

26833 Another change is incorporated as follows:

- 26834 • The **DESCRIPTION** is changed to make it clear that the function manipulates bytes rather
26835 than (possibly multi-byte) characters.

26836 **Issue 5**

26837 The **RETURN VALUE** section is updated to indicated that *strcspn()* returns the length of *s1*, and
26838 not *s1* itself as was previously stated.

26839 **NAME**

26840 strdup — duplicate a string

26841 **SYNOPSIS**

26842 EX #include <string.h>

26843 char *strdup(const char *s1);

26844

26845 **DESCRIPTION**

26846 The *strdup()* function returns a pointer to a new string, which is a duplicate of the string pointed
26847 to by *s1*. The returned pointer can be passed to *free()*. A null pointer is returned if the new
26848 string cannot be created.

26849 **RETURN VALUE**

26850 The *strdup()* function returns a pointer to a new string on success. Otherwise it returns a null
26851 pointer and sets *errno* to indicate the error.

26852 **ERRORS**26853 The *strdup()* function may fail if:

26854 [ENOMEM] Storage space available is insufficient.

26855 **EXAMPLES**

26856 None.

26857 **APPLICATION USAGE**

26858 None.

26859 **FUTURE DIRECTIONS**

26860 None.

26861 **SEE ALSO**26862 *malloc()*, *free()*, <string.h>.26863 **CHANGE HISTORY**

26864 First released in Issue 4, Version 2.

26865 **Issue 5**

26866 Moved from X/OPEN UNIX extension to BASE.

26867 **NAME**

26868 strerror — get error message string

26869 **SYNOPSIS**

26870 #include <string.h>

26871 char *strerror(int *errnum*);26872 **DESCRIPTION**

26873 The *strerror()* function maps the error number in *errnum* to a locale-dependent error message string and returns a pointer thereto. The string pointed to must not be modified by the program, but may be overwritten by a subsequent call to *strerror()* or *perror()*.

26876 EX The contents of the error message strings returned by *strerror()* should be determined by the setting of the LC_MESSAGES category in the current locale.

26878 The implementation will behave as if no function defined in this specification calls *strerror()*.

26879 The *strerror()* function will not change the setting of **errno** if successful.

26880 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 *strerror()*, then check *errno*.

26882 This interface need not be reentrant.

26883 **RETURN VALUE**

26884 EX Upon successful completion, *strerror()* returns a pointer to the generated message string. On error *errno* may be set, but no return value is reserved to indicate an error.

26886 **ERRORS**

26887 The *strerror()* function may fail if:

26888 EX [EINVAL] The value of *errnum* is not a valid error number.

26889 **EXAMPLES**

26890 None.

26891 **APPLICATION USAGE**

26892 None.

26893 **FUTURE DIRECTIONS**

26894 None.

26895 **SEE ALSO**

26896 *perror()*, <string.h>.

26897 **CHANGE HISTORY**

26898 First released in Issue 3.

26899 **Issue 4**

26900 The following change is incorporated for alignment with the ISO C standard:

- 26901 • The function is no longer marked as an extension.

26902 Other changes are incorporated as follows:

- 26903 • In the DESCRIPTION (a) the term “language-dependent” is replaced by “locale-dependent”,
26904 and (b) a statement about the use of the LC_MESSAGES category for determining the
26905 language of error messages is added and marked as an extension.
- 26906 • The fact that *strerror()* can return a null pointer on failure and set *errno* is marked as an
26907 extension.
- 26908 • The [EINVAL] error is marked as an extension.
- 26909 • The FUTURE DIRECTIONS section is removed.

26910 **Issue 5**

26911 The DESCRIPTION is updated to indicate that **errno** will not be changed if the function is
26912 successful.

26913 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

26914 **NAME**

26915 strfmon — convert monetary value to a string

26916 **SYNOPSIS**26917 EX

```
#include <monetary.h>
```

26918

```
ssize_t strfmon(char *s, size_t maxsize, const char *format, ...);
```

26919

26920 **DESCRIPTION**26921 The *strfmon()* function places characters into the array pointed to by *s* as controlled by the string
26922 pointed to by *format*. No more than *maxsize* bytes are placed into the array.26923 The format is a character string that contains two types of objects: plain characters, which are
26924 simply copied to the output stream, and conversion specifications, each of which results in the
26925 fetching of zero or more arguments which are converted and formatted. The results are
26926 undefined if there are insufficient arguments for the format. If the format is exhausted while
26927 arguments remain, the excess arguments are simply ignored.

26928 A conversion specification consists of the following sequence:

- 26929 • a % character
- 26930 • optional flags
- 26931 • optional field width
- 26932 • optional left precision
- 26933 • optional right precision
- 26934 • a required conversion character that determines the conversion to be performed.

26935 **Flags**

26936 One or more of the following optional flags can be specified to control the conversion:

26937 **=f** An = followed by a single character *f* which is used as the numeric fill character. The
26938 fill character must be representable in a single byte in order to work with precision and
26939 width counts. The default numeric fill character is the space character. This flag does
26940 not affect field width filling which always uses the space character. This flag is ignored
26941 unless a left precision (see below) is specified.

26942 **^** Do not format the currency amount with grouping characters. The default is to insert
26943 the grouping characters if defined for the current locale.

26944 **+ or (** Specify the style of representing positive and negative currency amounts. Only one of
26945 + or (may be specified. If + is specified, the locale's equivalent of + and – are used (for
26946 example, in the U.S.A.: the empty string if positive and – if negative). If (is specified,
26947 negative amounts are enclosed within parentheses. If neither flag is specified, the +
26948 style is used.

26949 **!** Suppress the currency symbol from the output conversion.

26950 **–** Specify the alignment. If this flag is present all fields are left-justified (padded to the
26951 right) rather than right-justified.

26952 **Field Width**

26953 **w** A decimal digit string *w* specifying a minimum field width in bytes in which the result
 26954 of the conversion is right-justified (or left-justified if the flag `-` is specified). The default
 26955 is 0.

26956 **Left Precision**

26957 **#n** A `#` followed by a decimal digit string *n* specifying a maximum number of digits
 26958 expected to be formatted to the left of the radix character. This option can be used to
 26959 keep the formatted output from multiple calls to the *strfmon()* aligned in the same
 26960 columns. It can also be used to fill unused positions with a special character as in
 26961 `$***123.45`. This option causes an amount to be formatted as if it has the number of
 26962 digits specified by *n*. If more than *n* digit positions are required, this conversion
 26963 specification is ignored. Digit positions in excess of those actually required are filled
 26964 with the numeric fill character (see the `=f` flag above).

26965 If grouping has not been suppressed with the `^` flag, and it is defined for the current
 26966 locale, grouping separators are inserted before the fill characters (if any) are added.
 26967 Grouping separators are not applied to fill characters even if the fill character is a digit.

26968 To ensure alignment, any characters appearing before or after the number in the
 26969 formatted output such as currency or sign symbols are padded as necessary with space
 26970 characters to make their positive and negative formats an equal length.

26971 **Right Precision**

26972 **.p** A period followed by a decimal digit string *p* specifying the number of digits after the
 26973 radix character. If the value of the right precision *p* is 0, no radix character appears. If a
 26974 right precision is not included, a default specified by the current locale is used. The
 26975 amount being formatted is rounded to the specified number of digits prior to
 26976 formatting.

26977 **Conversion Characters**

26978 The conversion characters and their meanings are:

26979 **i** The **double** argument is formatted according to the locale's international currency
 26980 format (for example, in the U.S.A.: `USD 1,234.56`).

26981 **n** The **double** argument is formatted according to the locale's national currency format
 26982 (for example, in the U.S.A.: `$1,234.56`).

26983 **%** Convert to a `%`; no argument is converted. The entire conversion specification must be
 26984 `%%`.

26985 **Locale Information**

26986 The `LC_MONETARY` category of the program's locale affects the behaviour of this function
 26987 including the monetary radix character (which may be different from the numeric radix
 26988 character affected by the `LC_NUMERIC` category), the grouping separator, the currency symbols
 26989 and formats. The international currency symbol should be conformant with the ISO 4217:1987
 26990 standard.

26991 If the value of *maxsize* is greater than `{SSIZE_MAX}`, the result is implementation-dependent.

26992 **RETURN VALUE**

26993 If the total number of resulting bytes including the terminating null byte is not more than
 26994 *maxsize*, *strfmon()* returns the number of bytes placed into the array pointed to by *s*, not
 26995 including the terminating null byte. Otherwise, -1 is returned, the contents of the array are
 26996 indeterminate, and *errno* is set to indicate the error.

26997 **ERRORS**

26998 The *strfmon()* function will fail if:

26999 [E2BIG] Conversion stopped due to lack of space in the buffer.

27000 **EXAMPLES**

27001 Given a locale for the U.S.A. and the values 123.45, -123.45 and 3456.781:

Conversion Specification	Output	Comments
%n	\$123.45 -\$123.45 \$3,456.78	default formatting
%11n	\$123.45 -\$123.45 \$3,456.78	right align within an 11 character field
%#5n	\$ 123.45 -\$ 123.45 \$ 3,456.78	aligned columns for values up to 99,999
%=*#5n	\$***123.45 -\$***123.45 \$*3,456.78	specify a fill character
%=0#5n	\$000123.45 -\$000123.45 \$03,456.78	fill characters do not use grouping even if the fill character is a digit
%^#5n	\$ 123.45 -\$ 123.45 \$ 3456.78	disable the grouping separator
%^#5.0n	\$ 123 -\$ 123 \$ 3457	round off to whole units
%^#5.4n	\$ 123.4500 -\$ 123.4500 \$ 3456.7810	increase the precision
%(#5n	123.45 (\$ 123.45) \$ 3,456.78	use an alternative pos/neg style
%	(#5n (123.45) 3,456.78	123.45

27035 **APPLICATION USAGE**

27036 None.

27037 **FUTURE DIRECTIONS**

27038 Lower-case conversion characters are reserved for future standards use and upper-case for
 27039 implementation-dependent use.

27040 **SEE ALSO**27041 *localeconv()*, <monetary.h>.27042 **CHANGE HISTORY**

27043 First released in Issue 4.

27044 **Issue 5**

27045 Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

27046 A sentence is added to the DESCRIPTION warning about values of *maxsize* that are greater than
27047 {SSIZE_MAX}.

27048 **NAME**

27049 strftime — convert date and time to a string

27050 **SYNOPSIS**

27051 #include <time.h>

```
27052       size_t strftime(char *s, size_t maxsize, const char *format,
27053                       const struct tm *timptr);
```

27054 **DESCRIPTION**

27055 The *strftime()* function places bytes into the array pointed to by *s* as controlled by the string
 27056 pointed to by *format*. The *format* string consists of zero or more conversion specifications and
 27057 ordinary characters. A conversion specification consists of a % character and a terminating
 27058 conversion character that determines the conversion specification's behaviour. All ordinary
 27059 characters (including the terminating null byte) are copied unchanged into the array. If copying
 27060 takes place between objects that overlap, the behaviour is undefined. No more than *maxsize*
 27061 bytes are placed into the array. Each conversion specification is replaced by appropriate
 27062 characters as described in the following list. The appropriate characters are determined by the
 27063 program's locale and by the values contained in the structure pointed to by *timptr*.

27064 Local timezone information is used as though *strftime()* called *tzset()*.

27065	%a	is replaced by the locale's abbreviated weekday name.
27066	%A	is replaced by the locale's full weekday name.
27067	%b	is replaced by the locale's abbreviated month name.
27068	%B	is replaced by the locale's full month name.
27069	%c	is replaced by the locale's appropriate date and time representation.
27070 EX	%C	is replaced by the century number (the year divided by 100 and truncated to an integer)
27071		as a decimal number [00-99].
27072	%d	is replaced by the day of the month as a decimal number [01,31].
27073 EX	%D	same as %m/%d/%y.
27074	%e	is replaced by the day of the month as a decimal number [1,31]; a single digit is
27075		preceded by a space.
27076	%h	same as %b.
27077	%H	is replaced by the hour (24-hour clock) as a decimal number [00,23].
27078	%I	is replaced by the hour (12-hour clock) as a decimal number [01,12].
27079	%j	is replaced by the day of the year as a decimal number [001,366].
27080	%m	is replaced by the month as a decimal number [01,12].
27081	%M	is replaced by the minute as a decimal number [00,59].
27082 EX	%n	is replaced by a newline character.
27083	%p	is replaced by the locale's equivalent of either a.m. or p.m.
27084 EX	%r	is replaced by the time in a.m. and p.m. notation; in the POSIX locale this is equivalent
27085		to %I:%M:%S %p.
27086	%R	is replaced by the time in 24 hour notation (%H:%M).
27087	%S	is replaced by the second as a decimal number [00,61].
27088 EX	%t	is replaced by a tab character.
27089	%T	is replaced by the time (%H:%M:%S).
27090	%u	is replaced by the weekday as a decimal number [1,7], with 1 representing Monday.
27091	%U	is replaced by the week number of the year (Sunday as the first day of the week) as a
27092		decimal number [00,53].
27093	%V	is replaced by the week number of the year (Monday as the first day of the week) as a
27094		decimal number [01,53]. If the week containing 1 January has four or more days in the
27095		new year, then it is considered week 1. Otherwise, it is week 53 of the previous year,
27096		and the next week is week 1.
27097	%w	is replaced by the weekday as a decimal number [0,6], with 0 representing Sunday.

27098	%W	is 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.
27099		
27100		
27101	%x	is replaced by the locale's appropriate date representation.
27102	%X	is replaced by the locale's appropriate time representation.
27103	%y	is replaced by the year without century as a decimal number [00,99].
27104	%Y	is replaced by the year with century as a decimal number.
27105	%Z	is replaced by the timezone name or abbreviation, or by no bytes if no timezone information exists.
27106		
27107	%%	is replaced by %.

27108 If a conversion specification does not correspond to any of the above, the behaviour is
 27109 undefined.

27110 **Modified Conversion Specifiers**

27111 EX Some conversion specifiers can be modified by the E or O modifier characters to indicate that an
 27112 alternative format or specification should be used rather than the one normally used by the
 27113 unmodified conversion specifier. If the alternative format or specification does not exist for the
 27114 current locale, (see ERA in the **XBD** specification, **Section 5.3.5**) the behaviour will be as if the
 27115 unmodified conversion specification were used.

27116	%Ec	is replaced by the locale's alternative appropriate date and time representation.
27117	%EC	is replaced by the name of the base year (period) in the locale's alternative
27118		representation.
27119	%Ex	is replaced by the locale's alternative date representation.
27120	%EX	is replaced by the locale's alternative time representation.
27121	%Ey	is replaced by the offset from %EC (year only) in the locale's alternative representation.
27122	%EY	is replaced by the full alternative year representation.
27123	%Od	is replaced by the day of the month, using the locale's alternative numeric symbols,
27124		filled as needed with leading zeros if there is any alternative symbol for zero, otherwise
27125		with leading spaces.
27126	%Oe	is replaced by the day of month, using the locale's alternative numeric symbols, filled as
27127		needed with leading spaces.
27128	%OH	is replaced by the hour (24-hour clock) using the locale's alternative numeric symbols.
27129	%OI	is replaced by the hour (12-hour clock) using the locale's alternative numeric symbols.
27130	%Om	is replaced by the month using the locale's alternative numeric symbols.
27131	%OM	is replaced by the minutes using the locale's alternative numeric symbols.
27132	%OS	is replaced by the seconds using the locale's alternative numeric symbols.
27133	%Ou	is replaced by the weekday as a number in the locale's alternative representation
27134		(Monday=1).
27135	%OU	is replaced by the week number of the year (Sunday as the first day of the week, rules
27136		corresponding to %U) using the locale's alternative numeric symbols.
27137	%OV	is replaced by the week number of the year (Monday as the first day of the week, rules
27138		corresponding to %V) using the locale's alternative numeric symbols.
27139	%Ow	is replaced by the number of the weekday (Sunday=0) using the locale's alternative
27140		numeric symbols.
27141	%OW	is replaced by the week number of the year (Monday as the first day of the week) using
27142		the locale's alternative numeric symbols.
27143	%Oy	is replaced by the year (offset from %C) using the locale's alternative numeric symbols.

27145 **RETURN VALUE**

27146 If the total number of resulting bytes including the terminating null byte is not more than

27147 *maxsize*, *strftime()* returns the number of bytes placed into the array pointed to by *s*, not
 27148 including the terminating null byte. Otherwise, 0 is returned and the contents of the array are
 27149 indeterminate.

27150 ERRORS

27151 No errors are defined.

27152 EXAMPLES

27153 None.

27154 APPLICATION USAGE

27155 The range of values for %S is [00,61] rather than [00,59] to allow for the occasional leap second
 27156 and even more infrequent double leap second.

27157 Some of the conversion specifications marked EX are duplicates of others. They are included for
 27158 compatibility with *nl_cxtime()* and *nl_ascxtime()*, which were published in Issue 2.

27159 Applications should use %Y (4-digit years) in preference to %y (2-digit years).

27160 FUTURE DIRECTIONS

27161 None.

27162 SEE ALSO

27163 *asctime()*, *clock()*, *ctime()*, *difftime()*, *gmtime()*, *localtime()*, *mktime()*, *strptime()*, *time()*, *utime()*,
 27164 **<time.h>**.

27165 CHANGE HISTORY

27166 First released in Issue 3.

27167 Issue 4

27168 The following changes are incorporated for alignment with the ISO C standard:

- 27169 • The type of argument *format* is changed from **char *** to **const char ***, and the type of argument
 27170 *timptr* is changed from **struct tm*** to **const struct tm***.
- 27171 • In the description of the %Z conversion specification, the words “or abbreviation” are added
 27172 to indicate that *strftime()* does not necessarily return a full timezone name.

27173 Other changes are incorporated as follows:

- 27174 • The DESCRIPTION is expanded to describe modified conversion specifiers.
- 27175 • %C, %e, %R, %u and %V are added to the list of valid conversion specifications.
- 27176 • The DESCRIPTION and RETURN VALUE sections are changed to make it clear when the
 27177 function uses byte values rather than (possibly multi-byte) character values.

27178 Issue 5

27179 The description of %OV is changed to be consistent with %V and defines Monday as the first
 27180 day of the week.

27181 The description of %Oy is clarified.

27182 **NAME**

27183 strlen — get string length

27184 **SYNOPSIS**

27185 #include <string.h>

27186 size_t strlen(const char *s);

27187 **DESCRIPTION**

27188 The *strlen()* function computes the number of bytes in the string to which *s* points, not including
 27189 the terminating null byte.

27190 **RETURN VALUE**27191 The *strlen()* function returns the length of *s*; no return value is reserved to indicate an error.27192 **ERRORS**

27193 No errors are defined.

27194 **EXAMPLES**

27195 None.

27196 **APPLICATION USAGE**

27197 None.

27198 **FUTURE DIRECTIONS**

27199 None.

27200 **SEE ALSO**

27201 <string.h>.

27202 **CHANGE HISTORY**

27203 First released in Issue 1.

27204 Derived from Issue 1 of the SVID.

27205 **Issue 4**

27206 The following change is incorporated for alignment with the ISO C standard:

- 27207 • The type of argument *s* is changed from **char *** to **const char ***.

27208 Another change is incorporated as follows:

- 27209 • The **DESCRIPTION** is changed to make it clear that the function works in units of bytes
 27210 rather than (possibly multi-byte) characters.

27211 **Issue 5**

27212 The **RETURN VALUE** section is updated to indicate that *strlen()* returns the length of *s*, and not
 27213 *s* itself as was previously stated.

27214 NAME

27215 strncasecmp — case-insensitive string comparison

27216 SYNOPSIS

27217 EX #include <strings.h>

27218 int strncasecmp(const char *s1, const char *s2, size_t n);

27219

27220 DESCRIPTION

27221 Refer to *strcasecmp()*.

27222 CHANGE HISTORY

27223 First released in Issue 4, Version 2.

27224 Issue 5

27225 Moved from X/OPEN UNIX extension to BASE.

27226 **NAME**

27227 strncat — concatenate part of two strings

27228 **SYNOPSIS**

27229 #include <string.h>

27230 char *strncat(char *s1, const char *s2, size_t n);

27231 **DESCRIPTION**

27232 The *strncat()* function appends not more than *n* bytes (a null byte and bytes that follow it are not
 27233 appended) from the array pointed to by *s2* to the end of the string pointed to by *s1*. The initial
 27234 byte of *s2* overwrites the null byte at the end of *s1*. A terminating null byte is always appended
 27235 to the result. If copying takes place between objects that overlap, the behaviour is undefined.

27236 **RETURN VALUE**27237 The *strncat()* function returns *s1*; no return value is reserved to indicate an error.27238 **ERRORS**

27239 No errors are defined.

27240 **EXAMPLES**

27241 None.

27242 **APPLICATION USAGE**

27243 None.

27244 **FUTURE DIRECTIONS**

27245 None.

27246 **SEE ALSO**27247 *strcat()*, <string.h>.27248 **CHANGE HISTORY**

27249 First released in Issue 1.

27250 Derived from Issue 1 of the SVID.

27251 **Issue 4**

27252 The following change is incorporated for alignment with the ISO C standard:

- 27253 • The type of argument *s2* is changed from **char *** to **const char ***.

27254 Another change is incorporated as follows:

- 27255 • The DESCRIPTION is changed to make it clear that the function manipulates bytes rather
 27256 than (possibly multi-byte) characters.

27257 **NAME**

27258 strncmp — compare part of two strings

27259 **SYNOPSIS**

27260 #include <string.h>

27261 int strncmp(const char *s1, const char *s2, size_t n);

27262 **DESCRIPTION**27263 The *strncmp()* function compares not more than *n* bytes (bytes that follow a null byte are not
27264 compared) from the array pointed to by *s1* to the array pointed to by *s2*.27265 The sign of a non-zero return value is determined by the sign of the difference between the
27266 values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the strings
27267 being compared.27268 **RETURN VALUE**27269 Upon successful completion, *strncmp()* returns an integer greater than, equal to or less than 0, if
27270 the possibly null-terminated array pointed to by *s1* is greater than, equal to or less than the
27271 possibly null-terminated array pointed to by *s2* respectively.27272 **ERRORS**

27273 No errors are defined.

27274 **EXAMPLES**

27275 None.

27276 **APPLICATION USAGE**

27277 None.

27278 **FUTURE DIRECTIONS**

27279 None.

27280 **SEE ALSO**27281 *strcmp()*, <string.h>.27282 **CHANGE HISTORY**

27283 First released in Issue 1.

27284 Derived from Issue 1 of the SVID.

27285 **Issue 4**

27286 The following change is incorporated for alignment with the ISO C standard:

- 27287
- The type of arguments *s1* and *s2* are changed from **char *** to **const char ***.

27288 Another change is incorporated as follows:

- 27289
- The DESCRIPTION is changed to make it clear that the function manipulates bytes rather
27290 than (possibly multi-byte) characters.

27291 **NAME**

27292 strncpy — copy part of a string

27293 **SYNOPSIS**

27294 #include <string.h>

27295 char *strncpy(char *s1, const char *s2, size_t n);

27296 **DESCRIPTION**

27297 The *strncpy()* function copies not more than *n* bytes (bytes that follow a null byte are not copied) from the array pointed to by *s2* to the array pointed to by *s1*. If copying takes place between objects that overlap, the behaviour is undefined.

27300 If the array pointed to by *s2* is a string that is shorter than *n* bytes, null bytes are appended to the copy in the array pointed to by *s1*, until *n* bytes in all are written.

27302 **RETURN VALUE**27303 The *strncpy()* function returns *s1*; no return value is reserved to indicate an error.27304 **ERRORS**

27305 No errors are defined.

27306 **EXAMPLES**

27307 None.

27308 **APPLICATION USAGE**

27309 Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

27311 If there is no null byte in the first *n* bytes of the array pointed to by *s2*, the result will not be null-terminated.

27313 **FUTURE DIRECTIONS**

27314 None.

27315 **SEE ALSO**27316 *strcpy()*, <string.h>.27317 **CHANGE HISTORY**

27318 First released in Issue 1.

27319 Derived from Issue 1 of the SVID.

27320 **Issue 4**

27321 The following change is incorporated for alignment with the ISO C standard:

- 27322 • The type of argument *s2* is changed from **char *** to **const char ***.

27323 Another change is incorporated as follows:

- 27324 • The DESCRIPTION is changed to make it clear that the function manipulates bytes rather than (possibly multi-byte) characters.

27326 **NAME**

27327 strpbrk — scan string for byte

27328 **SYNOPSIS**

27329 #include <string.h>

27330 char *strpbrk(const char *s1, const char *s2);

27331 **DESCRIPTION**

27332 The *strpbrk()* function locates the first occurrence in the string pointed to by *s1* of any byte from
27333 the string pointed to by *s2*.

27334 **RETURN VALUE**

27335 Upon successful completion, *strpbrk()* returns a pointer to the byte or a null pointer if no byte
27336 from *s2* occurs in *s1*.

27337 **ERRORS**

27338 No errors are defined.

27339 **EXAMPLES**

27340 None.

27341 **APPLICATION USAGE**

27342 None.

27343 **FUTURE DIRECTIONS**

27344 None.

27345 **SEE ALSO**27346 *strchr()*, *strrchr()*, <string.h>.27347 **CHANGE HISTORY**

27348 First released in Issue 1.

27349 Derived from Issue 1 of the SVID.

27350 **Issue 4**

27351 The following change is incorporated for alignment with the ISO C standard:

- 27352 • The type of arguments *s1* and *s2* is changed from **char *** to **const char ***.

27353 Another change is incorporated as follows:

- 27354 • The DESCRIPTION and RETURN VALUE sections are changed to make it clear that the
27355 function works in units of bytes rather than (possibly multi-byte) characters.

27356 **NAME**

27357 strptime — date and time conversion

27358 **SYNOPSIS**

27359 EX #include <time.h>

27360 char *strptime(const char *buf, const char *format, struct tm *tm);

27361

27362 **DESCRIPTION**

27363 The *strptime()* function converts the character string pointed to by *buf* to values which are stored
 27364 in the **tm** structure pointed to by *tm*, using the format specified by *format*.

27365 The *format* is composed of zero or more directives. Each directive is composed of one of the
 27366 following: one or more white-space characters (as specified by *isspace()*); an ordinary character
 27367 (neither % nor a white-space character); or a conversion specification. Each conversion
 27368 specification is composed of a % character followed by a conversion character which specifies
 27369 the replacement required. There must be white-space or other non-alphanumeric characters
 27370 between any two conversion specifications. The following conversion specifications are
 27371 supported:

27372	%a	is the day of week, using the locale's weekday names; either the abbreviated or full name may be specified.
27373		
27374	%A	is the same as %a.
27375	%b	is the month, using the locale's month names; either the abbreviated or full name may be specified.
27376		
27377	%B	is the same as %b.
27378	%c	is replaced by the locale's appropriate date and time representation.
27379	%C	is the century number [0,99]; leading zeros are permitted but not required.
27380	%d	is the day of month [1,31]; leading zeros are permitted but not required.
27381	%D	is the date as %m/%d/%y.
27382	%e	is the same as %d.
27383	%h	is the same as %b.
27384	%H	is the hour (24-hour clock) [0,23]; leading zeros are permitted but not required.
27385	%I	is the hour (12-hour clock) [1,12]; leading zeros are permitted but not required.
27386	%j	is the day number of the year [1,366]; leading zeros are permitted but not required.
27387	%m	is the month number [1,12]; leading zeros are permitted but not required.
27388	%M	is the minute [0-59]; leading zeros are permitted but not required.
27389	%n	is any white space.
27390	%p	is the locale's equivalent of a.m or p.m.
27391	%r	is the time as %I:%M:%S %p.
27392	%R	is the time as %H:%M.
27393	%S	is the seconds [0,61]; leading zeros are permitted but not required.
27394	%t	is any white space.
27395	%T	is the time as %H:%M:%S.
27396	%U	is 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.
27397		
27398	%w	is the weekday as a decimal number [0,6], with 0 representing Sunday; leading zeros are permitted but not required.
27399		
27400	%W	is the 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.
27401		
27402	%x	is the date, using the locale's date format.
27403	%X	is the time, using the locale's time format.

27404	%y	is the year within century. When a century is not otherwise specified, values in the
27405		range 69-99 refer to years in the twentieth century (1969 to 1999 inclusive); values in the
27406		range 00-68 refer to years in the twenty-first century (2000 to 2068 inclusive). Leading
27407		zeros are permitted but not required.
27408	%Y	is the year, including the century (for example, 1988).
27409	%%	is replaced by %.

27410 Modified Directives

27411 Some directives can be modified by the E and O modifier characters to indicate that an
 27412 alternative format or specification should be used rather than the one normally used by the
 27413 unmodified directive. If the alternative format or specification does not exist in the current
 27414 locale, the behaviour will be as if the unmodified directive were used.

27415	%Ec	is the locale's alternative appropriate date and time representation.
27416	%EC	is the name of the base year (period) in the locale's alternative representation.
27417	%Ex	is the locale's alternative date representation.
27418	%EX	is the locale's alternative time representation.
27419	%Ey	is the offset from %EC (year only) in the locale's alternative representation.
27420	%EY	is the full alternative year representation.
27421	%Od	is the day of the month using the locale's alternative numeric symbols; leading zeros
27422		are permitted but not required.
27423	%Oe	is the same as %Od.
27424	%OH	is the hour (24-hour clock) using the locale's alternative numeric symbols.
27425	%OI	is the hour (12-hour clock) using the locale's alternative numeric symbols.
27426	%Om	is the month using the locale's alternative numeric symbols.
27427	%OM	is the minutes using the locale's alternative numeric symbols.
27428	%OS	is the seconds using the locale's alternative numeric symbols.
27429	%OU	is the week number of the year (Sunday as the first day of the week) using the locale's
27430		alternative numeric symbols.
27431	%Ow	is the number of the weekday (Sunday=0) using the locale's alternative numeric
27432		symbols.
27433	%OW	is the week number of the year (Monday as the first day of the week) using the locale's
27434		alternative numeric symbols.
27435	%Oy	is the year (offset from %C) using the locale's alternative numeric symbols.

27436 A directive composed of white-space characters is executed by scanning input up to the first
 27437 character that is not white-space (which remains unscanned), or until no more characters can be
 27438 scanned.

27439 A directive that is an ordinary character is executed by scanning the next character from the
 27440 buffer. If the character scanned from the buffer differs from the one comprising the directive, the
 27441 directive fails, and the differing and subsequent characters remain unscanned.

27442 A series of directives composed of %n, %t, white-space characters or any combination is
 27443 executed by scanning up to the first character that is not white space (which remains
 27444 unscanned), or until no more characters can be scanned.

27445 Any other conversion specification is executed by scanning characters until a character matching
 27446 the next directive is scanned, or until no more characters can be scanned. These characters,
 27447 except the one matching the next directive, are then compared to the locale values associated
 27448 with the conversion specifier. If a match is found, values for the appropriate **tm** structure
 27449 members are set to values corresponding to the locale information. Case is ignored when
 27450 matching items in *buf* such as month or weekday names. If no match is found, *strptime()* fails
 27451 and no more characters are scanned.

27452 **RETURN VALUE**

27453 Upon successful completion, *strptime()* returns a pointer to the character following the last
27454 character parsed. Otherwise, a null pointer is returned.

27455 **ERRORS**

27456 No errors are defined.

27457 **EXAMPLES**

27458 None.

27459 **APPLICATION USAGE**

27460 Several “same as” formats, and the special processing of white-space characters are provided in
27461 order to ease the use of identical *format* strings for *strftime()* and *strptime()*.

27462 Applications should use %Y (4-digit years) in preference to %y (2-digit years).

27463 **FUTURE DIRECTIONS**

27464 This function is expected to be mandatory in the next issue of this specification.

27465 **SEE ALSO**

27466 *scanf()*, *strftime()*, *time()*, <time.h>.

27467 **CHANGE HISTORY**

27468 First released in Issue 4.

27469 **Issue 5**

27470 Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

27471 The exact meaning of the %y and %Oy specifiers are clarified in the DESCRIPTION.

27472 **NAME**

27473 strchr — string scanning operation

27474 **SYNOPSIS**

27475 #include <string.h>

27476 char *strchr(const char *s, int c);

27477 **DESCRIPTION**

27478 The *strchr()* function locates the last occurrence of *c* (converted to a **char**) in the string pointed
27479 to by *s*. The terminating null byte is considered to be part of the string.

27480 **RETURN VALUE**

27481 Upon successful completion, *strchr()* returns a pointer to the byte or a null pointer if *c* does not
27482 occur in the string.

27483 **ERRORS**

27484 No errors are defined.

27485 **EXAMPLES**

27486 None.

27487 **APPLICATION USAGE**

27488 None.

27489 **FUTURE DIRECTIONS**

27490 None.

27491 **SEE ALSO**27492 *strchr()*, <string.h>.27493 **CHANGE HISTORY**

27494 First released in Issue 1.

27495 Derived from Issue 1 of the SVID.

27496 **Issue 4**

27497 The following change is incorporated for alignment with the ISO C standard:

- 27498 • The type of argument *s* is changed from **char *** to **const char ***.

27499 Another change is incorporated as follows:

- 27500 • The DESCRIPTION and RETURN VALUE sections are changed to make it clear that the
27501 function works in units of bytes rather than (possibly multi-byte) characters.

27502 **NAME**

27503 strspn — get length of a substring

27504 **SYNOPSIS**

27505 #include <string.h>

27506 size_t strspn(const char *s1, const char *s2);

27507 **DESCRIPTION**

27508 The *strspn()* function computes 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*.

27510 **RETURN VALUE**27511 The *strspn()* function returns the length of *s1*; no return value is reserved to indicate an error.27512 **ERRORS**

27513 No errors are defined.

27514 **EXAMPLES**

27515 None.

27516 **APPLICATION USAGE**

27517 None.

27518 **FUTURE DIRECTIONS**

27519 None.

27520 **SEE ALSO**27521 *strcspn()*, <string.h>.27522 **CHANGE HISTORY**

27523 First released in Issue 1.

27524 Derived from Issue 1 of the SVID.

27525 **Issue 4**

27526 The following change is incorporated for alignment with the ISO C standard:

- 27527 • The type of arguments *s1* and *s2* are changed from **char *** to **const char ***.

27528 Another change is incorporated as follows:

- 27529 • The **DESCRIPTION** is changed to make it clear that the function works in units of bytes rather than (possibly multi-byte) characters.

27531 **Issue 5**

27532 The **RETURN VALUE** section is updated to indicate that *strspn()* returns the length of *s*, and not *s* itself as was previously stated.

27534 **NAME**

27535 strstr — find a substring

27536 **SYNOPSIS**

27537 #include <string.h>

27538 char *strstr(const char *s1, const char *s2);

27539 **DESCRIPTION**

27540 The *strstr()* function locates the first occurrence in the string pointed to by *s1* of the sequence of
27541 bytes (excluding the terminating null byte) in the string pointed to by *s2*.

27542 **RETURN VALUE**

27543 Upon successful completion, *strstr()* returns a pointer to the located string or a null pointer if the
27544 string is not found.

27545 If *s2* points to a string with zero length, the function returns *s1*.

27546 **ERRORS**

27547 No errors are defined.

27548 **EXAMPLES**

27549 None.

27550 **APPLICATION USAGE**

27551 None.

27552 **FUTURE DIRECTIONS**

27553 None.

27554 **SEE ALSO**

27555 *strchr()*, <string.h>.

27556 **CHANGE HISTORY**

27557 First released in Issue 3.

27558 Entry included for alignment with the ANSI C standard.

27559 **Issue 4**

27560 The following change is incorporated for alignment with the ISO C standard:

- 27561 • The type of arguments *s1* and *s2* are changed from **char *** to **const char ***.

27562 Another change is incorporated as follows:

- 27563 • The DESCRIPTION is changed to make it clear that the function works in units of bytes
27564 rather than (possibly multi-byte) characters.

27565 **NAME**27566 `strtod` — convert string to a double-precision number27567 **SYNOPSIS**27568 `#include <stdlib.h>`27569 `double strtod(const char *str, char **endptr);`27570 **DESCRIPTION**

27571 The `strtod()` function converts the initial portion of the string pointed to by *str* to type **double**
 27572 representation. First it decomposes the input string into three parts: an initial, possibly empty,
 27573 sequence of white-space characters (as specified by `isspace()`); a subject sequence interpreted as a
 27574 floating-point constant; and a final string of one or more unrecognised characters, including the
 27575 terminating null byte of the input string. Then it attempts to convert the subject sequence to a
 27576 floating-point number, and returns the result.

27577 The expected form of the subject sequence is an optional + or – sign, then a non-empty sequence
 27578 of digits optionally containing a radix character, then an optional exponent part. An exponent
 27579 part consists of *e* or *E*, followed by an optional sign, followed by one or more decimal digits.
 27580 The subject sequence is defined as the longest initial subsequence of the input string, starting
 27581 with the first non-white-space character, that is of the expected form. The subject sequence is
 27582 empty if the input string is empty or consists entirely of white-space characters, or if the first
 27583 character that is not white space is other than a sign, a digit or a radix character.

27584 If the subject sequence has the expected form, the sequence starting with the first digit or the
 27585 radix character (whichever occurs first) is interpreted as a floating constant of the C language,
 27586 except that the radix character is used in place of a period, and that if neither an exponent part
 27587 nor a radix character appears, a radix character is assumed to follow the last digit in the string.
 27588 If the subject sequence begins with a minus sign, the value resulting from the conversion is
 27589 negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that
 27590 *endptr* is not a null pointer.

27591 The radix character is defined in the program's locale (category `LC_NUMERIC`). In the POSIX
 27592 locale, or in a locale where the radix character is not defined, the radix character defaults to a
 27593 period (`.`).

27594 In other than the POSIX locale, other implementation-dependent subject sequence forms may be
 27595 accepted.

27596 If the subject sequence is empty or does not have the expected form, no conversion is performed;
 27597 the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null
 27598 pointer.

27599 The `strtod()` function will not change the setting of **errno** if successful.

27600 Because 0 is returned on error and is also a valid return on success, an application wishing to
 27601 check for error situations should set *errno* to 0, then call `strtod()`, then check *errno*.

27602 **RETURN VALUE**

27603 Upon successful completion, `strtod()` returns the converted value. If no conversion could be
 27604 performed, 0 is returned, and *errno* may be set to `[EINVAL]`.

27605 If the correct value is outside the range of representable values, `±HUGE_VAL` is returned
 27606 (according to the sign of the value), and *errno* is set to `[ERANGE]`.

27607 If the correct value would cause an underflow, 0 is returned and *errno* is set to `[ERANGE]`.

27608 **ERRORS**27609 The *strtod()* function will fail if:

27610 [ERANGE] The value to be returned would cause overflow or underflow.

27611 The *strtod()* function may fail if:

27612 EX [EINVAL] No conversion could be performed.

27613 **EXAMPLES**

27614 None.

27615 **APPLICATION USAGE**

27616 None.

27617 **FUTURE DIRECTIONS**

27618 None.

27619 **SEE ALSO**27620 *isspace()*, *localeconv()*, *scanf()*, *setlocale()*, *strtol()*, *<stdlib.h>*, the XBD specification, **Chapter 5**,
27621 **Locale**.27622 **CHANGE HISTORY**

27623 First released in Issue 1.

27624 Derived from Issue 1 of the SVID.

27625 **Issue 4**

27626 The following changes are incorporated for alignment with the ISO C standard:

- 27627 • The function is no longer marked as an extension.
- 27628 • The type of argument *str* is changed from **char *** to **const char ***.
- 27629 • The name of the second argument is changed from *ptr* to *endptr*.
- 27630 • The precise conditions under which the [ERANGE] error can be set have been defined in the
- 27631 RETURN VALUE section.

27632 Other changes are incorporated as follows:

- 27633 • The DESCRIPTION is changed to make it clear when the function manipulates bytes and
- 27634 when it manipulates characters.
- 27635 • The [EINVAL] error is added to the ERRORS section and marked as an extension.

27636 **Issue 5**27637 The DESCRIPTION is updated to indicate that **errno** will not be changed if the function is
27638 successful.

27639 **NAME**27640 *strtok*, *strtok_r* — split string into tokens27641 **SYNOPSIS**

27642 #include <string.h>

27643 char **strtok*(char **s1*, const char **s2*);27644 char **strtok_r*(char **s*, const char **sep*, char ***lasts*);27645 **DESCRIPTION**

27646 A sequence of calls to *strtok*() breaks the string pointed to by *s1* into a sequence of tokens, each
 27647 of which is delimited by a byte from the string pointed to by *s2*. The first call in the sequence has
 27648 *s1* as its first argument, and is followed by calls with a null pointer as their first argument. The
 27649 separator string pointed to by *s2* may be different from call to call.

27650 The first call in the sequence searches the string pointed to by *s1* for the first byte that is *not*
 27651 contained in the current separator string pointed to by *s2*. If no such byte is found, then there
 27652 are no tokens in the string pointed to by *s1* and *strtok*() returns a null pointer. If such a byte is
 27653 found, it is the start of the first token.

27654 The *strtok*() function then searches from there for a byte that *is* contained in the current
 27655 separator string. If no such byte is found, the current token extends to the end of the string
 27656 pointed to by *s1*, and subsequent searches for a token will return a null pointer. If such a byte is
 27657 found, it is overwritten by a null byte, which terminates the current token. The *strtok*() function
 27658 saves a pointer to the following byte, from which the next search for a token will start.

27659 Each subsequent call, with a null pointer as the value of the first argument, starts searching from
 27660 the saved pointer and behaves as described above.

27661 The implementation will behave as if no function defined in this document calls *strtok*().

27662 The *strtok*() interface need not be reentrant.

27663 The function *strtok_r*() considers the null-terminated string *s* as a sequence of zero or more text
 27664 tokens separated by spans of one or more characters from the separator string *sep*. The
 27665 argument *lasts* points to a user-provided pointer which points to stored information necessary
 27666 for *strtok_r*() to continue scanning the same string.

27667 In the first call to *strtok_r*(), *s* points to a null-terminated string, *sep* to a null-terminated string of
 27668 separator characters and the value pointed to by *lasts* is ignored. The function *strtok_r*() returns
 27669 a pointer to the first character of the first token, writes a null character into *s* immediately
 27670 following the returned token, and updates the pointer to which *lasts* points.

27671 In subsequent calls, *s* is a NULL pointer and *lasts* will be unchanged from the previous call so
 27672 that subsequent calls will move through the string *s*, returning successive tokens until no tokens
 27673 remain. The separator string *sep* may be different from call to call. When no token remains in *s*,
 27674 a NULL pointer is returned.

27675 **RETURN VALUE**

27676 Upon successful completion, *strtok*() returns a pointer to the first byte of a token. Otherwise, if
 27677 there is no token, *strtok*() returns a null pointer.

27678 The function *strtok_r*() returns a pointer to the token found, or a NULL pointer when no token is
 27679 found.

27680 **ERRORS**

27681 No errors are defined.

27682 **EXAMPLES**

27683 None.

27684 **APPLICATION USAGE**

27685 None.

27686 **FUTURE DIRECTIONS**

27687 None.

27688 **SEE ALSO**

27689 <string.h>.

27690 **CHANGE HISTORY**

27691 First released in Issue 1.

27692 Derived from Issue 1 of the SVID.

27693 **Issue 4**

27694 The following changes are incorporated for alignment with the ISO C standard:

- 27695 • The function is no longer marked as an extension.
- 27696 • The type of argument *s2* is changed from **char *** to **const char ***.

27697 Another change is incorporated as follows:

- 27698 • The DESCRIPTION is changed to make it clear that the function manipulates bytes rather
- 27699 than (possibly multi-byte) characters.

27700 **Issue 5**27701 The *strtok_r()* function is included for alignment with the POSIX Threads Extension.27702 A note indicating that the *strtok()* interface need not be reentrant is added to the DESCRIPTION.

27703 **NAME**27704 **strtol** — convert string to a long integer27705 **SYNOPSIS**

27706 #include <stdlib.h>

27707 long int strtol(const char **str*, char ***endptr*, int *base*);27708 **DESCRIPTION**

27709 The *strtol()* function converts the initial portion of the string pointed to by *str* to a type **long int** representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by *isspace()*); a subject sequence interpreted as an integer represented in some radix determined by the value of *base*; and a final string of one or more unrecognised characters, including the terminating null byte of the input string. Then it attempts to convert the subject sequence to an integer, and returns the result.

27715 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.

27721 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.

27727 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.

27731 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.

27737 In other than the POSIX locale, additional implementation-dependent subject sequence forms may be accepted.

27739 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.

27742 The *strtol()* function will not change the setting of **errno** if successful.

27743 Because 0, LONG_MIN and LONG_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*.

27746 **RETURN VALUE**

27747 Upon successful completion *strtol()* returns the converted value, if any. If no conversion could

27748 EX

27749	be performed, 0 is returned and <i>errno</i> may be set to [EINVAL].	
27750	If the correct value is outside the range of representable values, LONG_MAX or LONG_MIN is	
27751	returned (according to the sign of the value), and <i>errno</i> is set to [ERANGE].	
27752	ERRORS	
27753	The <i>strtol()</i> function will fail if:	
27754	[ERANGE] The value to be returned is not representable.	
27755	The <i>strtol()</i> function may fail if:	
27756 EX	[EINVAL] The value of <i>base</i> is not supported.	
27757	EXAMPLES	
27758	None.	
27759	APPLICATION USAGE	
27760	None.	
27761	FUTURE DIRECTIONS	
27762	None.	
27763	SEE ALSO	
27764	<i>isalpha()</i> , <i>scanf()</i> , <i>strtod()</i> , <stdlib.h>.	
27765	CHANGE HISTORY	
27766	First released in Issue 1.	
27767	Derived from Issue 1 of the SVID.	
27768	Issue 4	
27769	The following changes are incorporated for alignment with the ISO C standard:	
27770	• The function is no longer marked as an extension.	
27771	• The type of argument <i>str</i> is changed from char * to const char * .	
27772	• The name of the second argument is changed from <i>ptr</i> to <i>endptr</i> .	
27773	• The DESCRIPTION is changed to indicate permitted forms of the subject sequence when <i>base</i>	
27774	is 0.	
27775	• The RETURN VALUE section is changed to indicate that LONG_MAX or LONG_MIN will	
27776	be returned if the converted value is too large or too small.	
27777	Other changes are incorporated as follows:	
27778	• The DESCRIPTION is changed to make it clear when the function manipulates bytes and	
27779	when it manipulates characters.	
27780	• In the RETURN VALUE section, text indicating that <i>errno</i> will be set when 0 is returned is	
27781	marked as an extension.	
27782	• The ERRORS section is updated in line with the RETURN VALUE section.	
27783	Issue 5	
27784	The DESCRIPTION is updated to indicate that errno will not be changed if the function is	
27785	successful.	

27786 **NAME**

27787 strtoul — convert string to an unsigned long

27788 **SYNOPSIS**

27789 #include <stdlib.h>

27790 unsigned long int strtoul(const char *str, char **endptr, int base);

27791 **DESCRIPTION**

27792 The *strtoul()* function converts the initial portion of the string pointed to by *str* to a type
 27793 **unsigned long int** representation. First it decomposes the input string into three parts: an initial,
 27794 possibly empty, sequence of white-space characters (as specified by *isspace()*); a subject
 27795 sequence interpreted as an integer represented in some radix determined by the value of *base*;
 27796 and a final string of one or more unrecognised characters, including the terminating null byte of
 27797 the input string. Then it attempts to convert the subject sequence to an unsigned integer, and
 27798 returns the result.

27799 If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant,
 27800 octal constant or hexadecimal constant, any of which may be preceded by a + or – sign. A
 27801 decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An
 27802 octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 to 7 only.
 27803 A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal
 27804 digits and letters a (or A) to f (or F) with values 10 to 15 respectively.

27805 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
 27806 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
 27807 by a + or – sign. The letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35;
 27808 only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is
 27809 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following
 27810 the sign if present.

27811 The subject sequence is defined as the longest initial subsequence of the input string, starting
 27812 with the first non-white-space character, that is of the expected form. The subject sequence
 27813 contains no characters if the input string is empty or consists entirely of white-space characters,
 27814 or if the first non-white-space character is other than a sign or a permissible letter or digit.

27815 If the subject sequence has the expected form and the value of *base* is 0, the sequence of
 27816 characters starting with the first digit is interpreted as an integer constant. If the subject
 27817 sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base
 27818 for conversion, ascribing to each letter its value as given above. If the subject sequence begins
 27819 with a minus sign, the value resulting from the conversion is negated. A pointer to the final
 27820 string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

27821 In other than the POSIX locale, additional implementation-dependent subject sequence forms
 27822 may be accepted.

27823 If the subject sequence is empty or does not have the expected form, no conversion is performed;
 27824 the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null
 27825 pointer.

27826 The *strtoul()* function will not change the setting of **errno** if successful.

27827 Because 0 and ULONG_MAX are returned on error and are also valid returns on success, an
 27828 application wishing to check for error situations should set *errno* to 0, then call *strtoul()*, then
 27829 check *errno*.

27830 **RETURN VALUE**27831 Upon successful completion *strtoul()* returns the converted value, if any. If no conversion could

27832 EX be performed, 0 is returned and *errno* may be set to [EINVAL]. If the correct value is outside the
 27833 range of representable values, ULONG_MAX is returned and *errno* is set to [ERANGE].

27834 **ERRORS**

27835 The *strtoul()* function will fail if:

27836 EX [EINVAL] The value of *base* is not supported.

27837 [ERANGE] The value to be returned is not representable.

27838 The *strtoul()* function may fail if:

27839 EX [EINVAL] No conversion could be performed.

27840 **EXAMPLES**

27841 None.

27842 **APPLICATION USAGE**

27843 Unlike *strtod()* and *strtol()*, *strtoul()* must always return a non-negative number; so, using the
 27844 return value of *strtoul()* for out-of-range numbers with *strtoul()* could cause more severe
 27845 problems than just loss of precision if those numbers can ever be negative.

27846 **FUTURE DIRECTIONS**

27847 None.

27848 **SEE ALSO**

27849 *isalpha()*, *scanf()*, *strtod()*, *strtol()*, <stdlib.h>.

27850 **CHANGE HISTORY**

27851 First released in Issue 4.

27852 Derived from the ANSI C standard.

27853 **Issue 5**

27854 The DESCRIPTION is updated to indicate that **errno** will not be changed if the function is
 27855 successful.

27856 **NAME**

27857 strxfrm — string transformation

27858 **SYNOPSIS**

27859 #include <string.h>

27860 size_t strxfrm(char *s1, const char *s2, size_t n);

27861 **DESCRIPTION**

27862 The *strxfrm()* function transforms the string pointed to by *s2* and places the resulting string into
 27863 the array pointed to by *s1*. The transformation is such that if *strcmp()* is applied to two
 27864 transformed strings, it returns a value greater than, equal to or less than 0, corresponding to the
 27865 result of *strcoll()* applied to the same two original strings. No more than *n* bytes are placed into
 27866 the resulting array pointed to by *s1*, including the terminating null byte. If *n* is 0, *s1* is permitted
 27867 to be a null pointer. If copying takes place between objects that overlap, the behaviour is
 27868 undefined.

27869 The *strxfrm()* function will not change the setting of **errno** if successful.

27870 Because no return value is reserved to indicate an error, an application wishing to check for error
 27871 situations should set *errno* to 0, then call *strcoll()*, then check *errno*.

27872 **RETURN VALUE**

27873 Upon successful completion, *strxfrm()* returns the length of the transformed string (not
 27874 including the terminating null byte). If the value returned is *n* or more, the contents of the array
 27875 pointed to by *s1* are indeterminate.

27876 EX On error, *strxfrm()* may set *errno* but no return value is reserved to indicate an error.27877 **ERRORS**27878 The *strxfrm()* function may fail if:

27879 EX [EINVAL] The string pointed to by the *s2* argument contains characters outside the
 27880 domain of the collating sequence.

27881 **EXAMPLES**

27882 None.

27883 **APPLICATION USAGE**

27884 The transformation function is such that two transformed strings can be ordered by *strcmp()* as
 27885 appropriate to collating sequence information in the program's locale (category LC_COLLATE).

27886 The fact that when *n* is 0, *s1* is permitted to be a null pointer, is useful to determine the size of the
 27887 *s1* array prior to making the transformation.

27888 **FUTURE DIRECTIONS**

27889 None.

27890 **SEE ALSO**27891 *strcmp()*, *strcoll()*, <string.h>.27892 **CHANGE HISTORY**

27893 First released in Issue 3.

27894 Entry included for alignment with the ISO C standard.

27895 **Issue 4**

27896 The following changes are incorporated for alignment with the ISO C standard:

- 27897 • The function is no longer marked as an extension.

- 27898 • The type of argument *s2* is changed from **char *** to **const char ***.
- 27899 Other changes are incorporated as follows:
- 27900 • The DESCRIPTION is changed to make it clear when the function manipulates byte values |
27901 and when it manipulates characters.
- 27902 • The sentence describing error returns in the RETURN VALUE section is marked as an |
27903 extension, as is the [EINVAL] error.
- 27904 • The APPLICATION USAGE section is expanded. |
- 27905 **Issue 5** |
- 27906 The DESCRIPTION is updated to indicate that **errno** will not be changed if the function is |
27907 successful.

27908 **NAME**

27909 swab — swap bytes

27910 **SYNOPSIS**

27911 EX #include <unistd.h>

27912 void swab(const void *src, void *dest, ssize_t nbytes);

27913

27914 **DESCRIPTION**

27915 The *swab()* function copies *nbytes* bytes, which are pointed to by *src*, to the object pointed to by
 27916 *dest*, exchanging adjacent bytes. The *nbytes* argument should be even. If *nbytes* is odd *swab()*
 27917 copies and exchanges *nbytes*–1 bytes and the disposition of the last byte is unspecified. If
 27918 copying takes place between objects that overlap, the behaviour is undefined. If *nbytes* is
 27919 negative, *swab()* does nothing.

27920 **RETURN VALUE**

27921 None.

27922 **ERRORS**

27923 No errors are defined.

27924 **EXAMPLES**

27925 None.

27926 **APPLICATION USAGE**

27927 None.

27928 **FUTURE DIRECTIONS**

27929 None.

27930 **SEE ALSO**

27931 <unistd.h>.

27932 **CHANGE HISTORY**

27933 First released in Issue 1.

27934 Derived from Issue 1 of the SVID.

27935 **Issue 4**

27936 The following changes are incorporated in this issue:

- 27937 • The <unistd.h> header is added to the SYNOPSIS section.
- 27938 • The type of argument *src* is changed from **char *** to **const void***, *dest* is changed from **char *** to
 27939 **void***, and *nbytes* is changed from **int** to **ssize_t**.
- 27940 • The DESCRIPTION now states explicitly that copying between overlapping objects results in
 27941 undefined behaviour. is changed to take account of the type change to *nbyte*; that is,
 27942 previously it was defined as **int** and could be positive or negative, whereas now it is defined
 27943 as an **unsigned** type. Also a statement about overlapping objects is added to the
 27944 DESCRIPTION.
- 27945 • The APPLICATION USAGE section is removed.

27946 **NAME**

27947 swapcontext — swap user context

27948 **SYNOPSIS**27949 EX `#include <ucontext.h>`27950 `int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);`

27951

27952 **DESCRIPTION**27953 Refer to *makecontext()*.27954 **CHANGE HISTORY**

27955 First released in Issue 4, Version 2.

27956 **Issue 5**

27957 Moved from X/OPEN UNIX extension to BASE.

27958 **NAME**

27959 swprintf — print formatted wide-character output

27960 **SYNOPSIS**

27961 #include <stdio.h>

27962 #include <wchar.h>

27963 int swprintf(wchar_t *s, size_t n, const wchar_t *format, ...);

27964 **DESCRIPTION**27965 Refer to *fwprintf()*.27966 **CHANGE HISTORY**

27967 First released in Issue 5.

27968 Include for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

27969 **NAME**

27970 swscanf — convert formatted wide-character input

27971 **SYNOPSIS**

27972 #include <stdio.h>

27973 #include <wchar.h>

27974 int swscanf(const wchar_t *s, const wchar_t *format, ...);

27975 **DESCRIPTION**

27976 Refer to *fwscanf()*.

27977 **CHANGE HISTORY**

27978 First released in Issue 5.

27979 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

27980 **NAME**

27981 symlink — make symbolic link to a file

27982 **SYNOPSIS**

27983 EX #include <unistd.h>

27984 int symlink(const char *path1, const char *path2);

27985

27986 **DESCRIPTION**

27987 The *symlink()* function creates a symbolic link. Its name is the pathname pointed to by *path2*,
 27988 which must be a pathname that does not name an existing file or symbolic link. The contents of
 27989 the symbolic link are the string pointed to by *path1*.

27990 **RETURN VALUE**

27991 Upon successful completion, *symlink()* returns 0. Otherwise, it returns -1 and sets *errno* to
 27992 indicate the error.

27993 **ERRORS**27994 The *symlink()* function will fail if:

27995 [EACCES] Write permission is denied in the directory where the symbolic link is being
 27996 created, or search permission is denied for a component of the path prefix of
 27997 *path2*.

27998 [EEXIST] The *path2* argument names an existing file or symbolic link.

27999 [EIO] An I/O error occurs while reading from or writing to the file system.

28000 [ELOOP] Too many symbolic links were encountered in resolving *path2*.

28001 [ENAMETOOLONG]

28002 The length of the *path2* argument exceeds {PATH_MAX}, or a pathname
 28003 component is longer than {NAME_MAX}.

28004 [ENOENT] A component of *path2* does not name an existing file or *path2* is an empty
 28005 string.

28006 [ENOSPC] The directory in which the entry for the new symbolic link is being placed
 28007 cannot be extended because no space is left on the file system containing the
 28008 directory, or the new symbolic link cannot be created because no space is left
 28009 on the file system which will contain the link, or the file system is out of file-
 28010 allocation resources.

28011 [ENOTDIR] A component of the path prefix of *path2* is not a directory.

28012 [EROFS] The new symbolic link would reside on a read-only file system.

28013 The *symlink()* function may fail if:

28014 [ENAMETOOLONG]

28015 Pathname resolution of a symbolic link produced an intermediate result
 28016 whose length exceeds {PATH_MAX}.

28017 **EXAMPLES**

28018 None.

28019 **APPLICATION USAGE**

28020 Like a hard link, a symbolic link allows a file to have multiple logical names. The presence of a
 28021 hard link guarantees the existence of a file, even after the original name has been removed. A
 28022 symbolic link provides no such assurance; in fact, the file named by the *path1* argument need not

- 28023 exist when the link is created. A symbolic link can cross file system boundaries.
- 28024 Normal permission checks are made on each component of the symbolic link pathname during
28025 its resolution.
- 28026 **FUTURE DIRECTIONS** |
- 28027 None.
- 28028 **SEE ALSO**
- 28029 *lchown()*, *link()*, *lstat()*, *open()*, *readlink()*, **<unistd.h>**.
- 28030 **CHANGE HISTORY**
- 28031 First released in Issue 4, Version 2. |
- 28032 **Issue 5** |
- 28033 Moved from X/OPEN UNIX extension to BASE.

28034 **NAME**

28035 sync — schedule filesystem updates

28036 **SYNOPSIS**

28037 EX #include <unistd.h>

28038 void sync(void);

28039

28040 **DESCRIPTION**28041 The *sync()* function causes all information in memory that updates file systems to be scheduled
28042 for writing out to all file systems.28043 The writing, although scheduled, is not necessarily complete upon return from *sync()*.28044 **RETURN VALUE**28045 The *sync()* function returns no value.28046 **ERRORS**

28047 No errors are defined.

28048 **EXAMPLES**

28049 None.

28050 **APPLICATION USAGE**

28051 None.

28052 **FUTURE DIRECTIONS**

28053 None.

28054 **SEE ALSO**28055 *fsync()*, <unistd.h>.28056 **CHANGE HISTORY**

28057 First released in Issue 4, Version 2.

28058 **Issue 5**

28059 Moved from X/OPEN UNIX extension to BASE.

28060 **NAME**

28061 sysconf — get configurable system variables

28062 **SYNOPSIS**

28063 #include <unistd.h>

28064 long int sysconf(int name);

28065 **DESCRIPTION**

28066 The *sysconf()* function provides a method for the application to determine the current value of a
 28067 configurable system limit or option (*variable*).

28068 The *name* argument represents the system variable to be queried. The following table lists the
 28069 minimal set of system variables from <limits.h>, <unistd.h> or <time.h> (for CLK_TCK) that
 28070 can be returned by *sysconf()*, and the symbolic constants, defined in <unistd.h> that are the
 28071 corresponding values used for *name*:

28072

28073

28074

28075

28076

28077

28078

28079

28080

28081

28082

28083

28084

28085

28086 EX

28087

28088

28089

28090

28091

28092

28093

28094

28095

28096

28097

28098

28099

28100

28101

28102

28103 EX

28104

28105

28106

28107

Variable	Value of Name
ARG_MAX	_SC_ARG_MAX
BC_BASE_MAX	_SC_BC_BASE_MAX
BC_DIM_MAX	_SC_BC_DIM_MAX
BC_SCALE_MAX	_SC_BC_SCALE_MAX
BC_STRING_MAX	_SC_BC_STRING_MAX
CHILD_MAX	_SC_CHILD_MAX
CLK_TCK	_SC_CLK_TCK
COLL_WEIGHTS_MAX	_SC_COLL_WEIGHTS_MAX
EXPR_NEST_MAX	_SC_EXPR_NEST_MAX
LINE_MAX	_SC_LINE_MAX
NGROUPS_MAX	_SC_NGROUPS_MAX
OPEN_MAX	_SC_OPEN_MAX
PASS_MAX	_SC_PASS_MAX (LEGACY)
_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
_POSIX2_SW_DEV	_SC_2_SW_DEV
_POSIX2_UPE	_SC_2_UPE
_POSIX2_VERSION	_SC_2_VERSION
_POSIX_JOB_CONTROL	_SC_JOB_CONTROL
_POSIX_SAVED_IDS	_SC_SAVED_IDS
_POSIX_VERSION	_SC_VERSION
RE_DUP_MAX	_SC_RE_DUP_MAX
STREAM_MAX	_SC_STREAM_MAX
TZNAME_MAX	_SC_TZNAME_MAX
_XOPEN_CRYPT	_SC_XOPEN_CRYPT
_XOPEN_ENH_I18N	_SC_XOPEN_ENH_I18N
_XOPEN_SHM	_SC_XOPEN_SHM

	Variable	Value of Name
28108		
28109		
28110 EX	_XOPEN_VERSION	_SC_XOPEN_VERSION
28111	_XOPEN_XCU_VERSION	_SC_XOPEN_XCU_VERSION
28112	_XOPEN_REALTIME	_SC_XOPEN_REALTIME
28113	_XOPEN_REALTIME_THREADS	_SC_XOPEN_REALTIME_THREADS
28114	_XOPEN_LEGACY	_SC_XOPEN_LEGACY
28115	ATEXIT_MAX	_SC_ATEXIT_MAX
28116	IOV_MAX	_SC_IOV_MAX
28117	PAGESIZE	_SC_PAGESIZE
28118	PAGE_SIZE	_SC_PAGE_SIZE
28119	_XOPEN_UNIX	_SC_XOPEN_UNIX
28120	_XBS5_ILP32_OFF32	_SC_XBS5_ILP32_OFF32
28121	_XBS5_ILP32_OFFBIG	_SC_XBS5_ILP32_OFFBIG
28122	_XBS5_LP64_OFF64	_SC_XBS5_LP64_OFF64
28123	_XBS5_LPBIG_OFFBIG	_SC_XBS5_LPBIG_OFFBIG
28124 RT	AIO_LISTIO_MAX	_SC_AIO_LISTIO_MAX
28125	AIO_MAX	_SC_AIO_MAX
28126	AIO_PRIO_DELTA_MAX	_SC_AIO_PRIO_DELTA_MAX
28127	DELAYTIMER_MAX	_SC_DELAYTIMER_MAX
28128	MQ_OPEN_MAX	_SC_MQ_OPEN_MAX
28129	MQ_PRIO_MAX	_SC_MQ_PRIO_MAX
28130	RTSIG_MAX	_SC_RTSIG_MAX
28131	SEM_NSEMS_MAX	_SC_SEM_NSEMS_MAX
28132	SEM_VALUE_MAX	_SC_SEM_VALUE_MAX
28133	SIGQUEUE_MAX	_SC_SIGQUEUE_MAX
28134	TIMER_MAX	_SC_TIMER_MAX
28135	_POSIX_ASYNCHRONOUS_IO	_SC_ASYNCHRONOUS_IO
28136	_POSIX_FSYNC	_SC_FSYNC
28137	_POSIX_MAPPED_FILES	_SC_MAPPED_FILES
28138 RT	_POSIX_MEMLOCK	_SC_MEMLOCK
28139	_POSIX_MEMLOCK_RANGE	_SC_MEMLOCK_RANGE
28140	_POSIX_MEMORY_PROTECTION	_SC_MEMORY_PROTECTION
28141 RT	_POSIX_MESSAGE_PASSING	_SC_MESSAGE_PASSING
28142	_POSIX_PRIORITIZED_IO	_SC_PRIORITIZED_IO
28143	_POSIX_PRIORITY_SCHEDULING	_SC_PRIORITY_SCHEDULING
28144	_POSIX_REALTIME_SIGNALS	_SC_REALTIME_SIGNALS
28145	_POSIX_SEMAPHORES	_SC_SEMAPHORES
28146	_POSIX_SHARED_MEMORY_OBJECTS	_SC_SHARED_MEMORY_OBJECTS
28147	_POSIX_SYNCHRONIZED_IO	_SC_SYNCHRONIZED_IO
28148	_POSIX_TIMERS	_SC_TIMERS
28149	Maximum size of <i>getgrgid_r()</i> and	_SC_GETGR_R_SIZE_MAX
28150	<i>getgrnam_r()</i> data buffers	
28151	Maximum size of <i>getpwuid_r()</i> and	_SC_GETPW_R_SIZE_MAX
28152	<i>getpwnam_r()</i> data buffers	
28153	LOGIN_NAME_MAX	_SC_LOGIN_NAME_MAX
28154	PTHREAD_DESTRUCTOR_ITERATIONS	_SC_THREAD_DESTRUCTOR_ITERATIONS
28155	PTHREAD_KEYS_MAX	_SC_THREAD_KEYS_MAX
28156	PTHREAD_STACK_MIN	_SC_THREAD_STACK_MIN

Variable	Value of Name
PTHREAD_THREADS_MAX	_SC_THREAD_THREADS_MAX
TTY_NAME_MAX	_SC_TTY_NAME_MAX
_POSIX_THREADS	_SC_THREADS
_POSIX_THREAD_ATTR_STACKADDR	_SC_THREAD_ATTR_STACKADDR
_POSIX_THREAD_ATTR_STACKSIZE	_SC_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_PRIORITY_SCHEDULING	_SC_THREAD_PRIORITY_SCHEDULING
_POSIX_THREAD_PRIO_INHERIT	_SC_THREAD_PRIO_INHERIT
_POSIX_THREAD_PRIO_PROTECT	_SC_THREAD_PRIO_PROTECT
_POSIX_THREAD_PROCESS_SHARED	_SC_THREAD_PROCESS_SHARED
_POSIX_THREAD_SAFE_FUNCTIONS	_SC_THREAD_SAFE_FUNCTIONS

RETURN VALUE

If *name* is an invalid value, *sysconf()* returns `-1` and sets *errno* to indicate the error. If the variable corresponding to *name* is associated with functionality that is not supported by the system, *sysconf()* returns `-1` without changing the value of *errno*.

Otherwise, *sysconf()* returns the current variable value on the system. The value returned will not be more restrictive than the corresponding value described to the application when it was compiled with the implementation's `<limits.h>`, `<unistd.h>` or `<time.h>`. The value will not change during the lifetime of the calling process.

ERRORS

The *sysconf()* function will 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 XCU specification. This would mean that the application is not running in an environment that conforms to the XCU specification. Some applications might be able to deal with this, others might not. However, the interfaces defined in this specification will continue to operate as specified, even if:

```
sysconf(_SC_2_VERSION)
```

reports that the utilities no longer perform as specified.

FUTURE DIRECTIONS

None.

SEE ALSO

confstr(), *pathconf()*, `<limits.h>`, `<time.h>`, `<unistd.h>`, the XCU specification of *getconf*.

CHANGE HISTORY

First released in Issue 3.

28201 Entry included for alignment with the POSIX.1-1988 standard.

28202 **Issue 4**

28203 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 28204 • The variables {STREAM_MAX} and {TZNAME_MAX} are added to the table of variables in
28205 the DESCRIPTION.

28206 The following change is incorporated for alignment with the ISO POSIX-2 standard:

- 28207 • The following variables are added to the table of configurable system limits in the
28208 DESCRIPTION:

28209	BC_BASE_MAX	_POSIX2_C_BIND	_POSIX2_SW_DEV
28210	BC_DIM_MAX	_POSIX2_C_DEV	_POSIX2_VERSION
28211	BC_SCALE_MAX	_POSIX2_C_VERSION	RE_DUP_MAX
28212	BC_STRING_MAX	_POSIX2_CHAR_TERM	
28213	COLL_WEIGHTS_MAX	_POSIX2_FORT_DEV	
28214	EXPR_NEST_MAX	_POSIX2_FORT_RUN	
28215	LINE_MAX	_POSIX2_LOCALEDEF	

28216 Other changes are incorporated as follows:

- 28217 • The type of the function return value is expanded to **long int**.
- 28218 • _XOPEN_VERSION is added to the table of configurable system limits; this should have
28219 been included in Issue 3.
- 28220 • The following variables are added to the table of configurable system limits in the
28221 DESCRIPTION and marked as extensions:

28222	_XOPEN_CRYPT
28223	_XOPEN_ENH_I18N
28224	_XOPEN_SHM
28225	_XOPEN_UNIX
- 28226 • In the RETURN VALUE section the header <time.h> is given as an alternative to <limits.h>
28227 and <unistd.h>.
- 28228 • The second paragraph is added to the APPLICATION USAGE section.

28229 **Issue 4, Version 2**

28230 For X/OPEN UNIX conformance, the ATEXT_MAX, IOV_MAX, PAGESIZE, PAGE_SIZE and
28231 _XOPEN_UNIX variables are added to the list of configurable system values that can be
28232 determined by calling *sysconf*().

28233 **Issue 5**

28234 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
28235 Threads Extension.

28236 Added the _XBS_ variables and name values to the table of system variables in the
28237 DESCRIPTION. These are all marked EX.

28238 **NAME**

28239 syslog — log a message

28240 **SYNOPSIS**28241 EX `#include <syslog.h>`28242 `void syslog(int priority, const char *message, ... /* argument */);`

28243

28244 **DESCRIPTION**28245 Refer to *closelog()*.28246 **CHANGE HISTORY**

28247 First released in Issue 4, Version 2.

28248 **Issue 5**

28249 Moved from X/OPEN UNIX extension to BASE.

28250 **NAME**

28251 system — issue a command

28252 **SYNOPSIS**

28253 #include <stdlib.h>

28254 int system(const char **command*);28255 **DESCRIPTION**

28256 The *system()* function passes the string pointed to by *command* to the host environment to be
 28257 executed by a command processor in an implementation-dependent manner. If the
 28258 implementation supports the **XCU** specification commands, the environment of the executed
 28259 command will be as if a child process were created using *fork()*, and the child process invoked
 28260 the *sh* utility (see *sh* in the **XCU** specification) using *execl()* as follows:

28261 execl(<*shell path*>, "sh", "-c", *command*, (char *)0);28262 where <*shell path*> is an unspecified pathname for the *sh* utility.

28263 The *system()* function ignores the SIGINT and SIGQUIT signals, and blocks the SIGCHLD
 28264 signal, while waiting for the command to terminate. If this might cause the application to miss a
 28265 signal that would have killed it, then the application should examine the return value from
 28266 *system()* and take whatever action is appropriate to the application if the command terminated
 28267 due to receipt of a signal.

28268 The *system()* function will not affect the termination status of any child of the calling processes
 28269 other than the process or processes it itself creates.

28270 The *system()* function will not return until the child process has terminated.28271 **RETURN VALUE**28272 If *command* is a null pointer, *system()* returns non-zero only if a command processor is available.

28273 If *command* is not a null pointer, *system()* returns the termination status of the command
 28274 language interpreter in the format specified by *waitpid()*. The termination status of the
 28275 command language interpreter is as specified for the *sh* utility, except that if some error prevents
 28276 the command language interpreter from executing after the child process is created, the return
 28277 value from *system()* will be as if the command language interpreter had terminated using
 28278 *exit(127)* or *_exit(127)*. If a child process cannot be created, or if the termination status for the
 28279 command language interpreter cannot be obtained, *system()* returns -1 and sets *errno* to indicate
 28280 the error.

28281 **ERRORS**28282 The *system()* function may set *errno* values as described by *fork()*.28283 In addition, *system()* may fail if:28284 [ECHILD] The status of the child process created by *system()* is no longer available.28285 **EXAMPLES**

28286 None.

28287 **APPLICATION USAGE**

28288 If the return value of *system()* is not -1, its value can be decoded through the use of the macros
 28289 described in <**sys/wait.h**>. For convenience, these macros are also provided in <**stdlib.h**>.

28290 To determine whether or not the **XCU** specification's environment is present, use:

28291 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 will be 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 will be 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 will be closed, and the process ID and parent process ID will be different. Also, if the executed utility changes its environment variables or its current working directory, that change will not be 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.

28318 FUTURE DIRECTIONS

28319 None.

28320 SEE ALSO

28321 *exec*, *pipe()*, *waitpid()*, <limits.h>, <signal.h>, <stdlib.h>, the XCU specification.

28322 CHANGE HISTORY

28323 First released in Issue 1.

28324 Derived from Issue 1 of the SVID.

28325 Issue 4

28326 The following changes are incorporated for alignment with the ISO POSIX-2 standard:

- 28327 • The function is no longer marked as an extension.
- 28328 • The name of the argument is changed from *string* to *command*, and its type is changed from
28329 **char *** to **const char ***.
- 28330 • The DESCRIPTION and RETURN VALUE sections are completely replaced to bring them in
28331 line with ISO POSIX-2 standard. They still describe essentially the same functionality, albeit
28332 that the definition is more complete.
- 28333 • The ERRORS section is changed to indicate that *system()* may return error values described
28334 for *fork()*.
- 28335 • The APPLICATION USAGE section is added.

28336 **NAME**

28337 tan — tangent function

28338 **SYNOPSIS**

28339 #include <math.h>

28340 double tan(double x);

28341 **DESCRIPTION**28342 The *tan()* function computes the tangent of its argument *x*, measured in radians.28343 An application wishing to check for error situations should set *errno* to 0 before calling *tan()*. If
28344 *errno* is non-zero on return, or the return value is NaN, an error has occurred.28345 The *tan()* function may lose accuracy when its argument is far from 0.0 .28346 **RETURN VALUE**28347 Upon successful completion, *tan()* returns the tangent of *x*.28348 EX If *x* is NaN, NaN is returned and *errno* may be set to [EDOM].28349 EX If *x* is $\pm\text{Inf}$, either 0.0 is returned and *errno* is set to [EDOM], or NaN is returned and *errno* may be
28350 set to [EDOM].28351 If the correct value would cause overflow, $\pm\text{HUGE_VAL}$ is returned and *errno* is set to
28352 [ERANGE].28353 If the correct value would cause underflow, 0.0 is returned and *errno* may be set to [ERANGE].28354 **ERRORS**28355 The *tan()* function will fail if:

28356 [ERANGE] The value to be returned would cause overflow.

28357 The *tan()* function may fail if:28358 EX [EDOM] The value *x* is NaN or $\pm\text{Inf}$.

28359 [ERANGE] The value to be returned would cause underflow.

28360 EX No other errors will occur.

28361 **EXAMPLES**

28362 None.

28363 **APPLICATION USAGE**

28364 None.

28365 **FUTURE DIRECTIONS**

28366 None.

28367 **SEE ALSO**28368 *atan()*, *isnan()*, <math.h>.28369 **CHANGE HISTORY**

28370 First released in Issue 1.

28371 Derived from Issue 1 of the SVID.

28372 **Issue 4**

28373 The following changes are incorporated in this issue:

- 28374
- Removed references to *matherr()*.

28375	<ul style="list-style-type: none">• The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions.	
28376		
28377	<ul style="list-style-type: none">• The return value specified for [EDOM] is marked as an extension.	
28378	Issue 5 The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes in previous issues.	
28379		
28380		

28381 **NAME**

28382 tanh — hyperbolic tangent function

28383 **SYNOPSIS**

28384 #include <math.h>

28385 double tanh(double x);

28386 **DESCRIPTION**28387 The *tanh()* function computes the hyperbolic tangent of *x*.28388 An application wishing to check for error situations should set *errno* to 0 before calling *tanh()*. If
28389 *errno* is non-zero on return, or the return value is NaN, an error has occurred.28390 **RETURN VALUE**28391 Upon successful completion, *tanh()* returns the hyperbolic tangent of *x*.28392 EX If *x* is NaN, NaN is returned and *errno* may be set to [EDOM].28393 If the correct value would cause underflow, 0.0 is returned and *errno* may be set to [ERANGE].28394 **ERRORS**28395 The *tanh()* function may fail if:28396 EX [EDOM] The value of *x* is NaN.

28397 [ERANGE] The correct result would cause underflow.

28398 EX No other errors will occur.

28399 **EXAMPLES**

28400 None.

28401 **APPLICATION USAGE**

28402 None.

28403 **FUTURE DIRECTIONS**

28404 None.

28405 **SEE ALSO**28406 *atanh()*, *isnan()*, *tan()*, <math.h>.28407 **CHANGE HISTORY**

28408 First released in Issue 1.

28409 Derived from Issue 1 of the SVID.

28410 **Issue 4**

28411 The following changes are incorporated in this issue:

- 28412 • Removed references to *matherr()*.
- 28413 • The RETURN VALUE and ERRORS sections are substantially rewritten for alignment with
- 28414 the ISO C standard and to rationalise error handling in the mathematics functions.
- 28415 • The return value specified for [EDOM] is marked as an extension.

28416 **Issue 5**28417 The DESCRIPTION is updated to indicate how an application should check for an error. This
28418 text was previously published in the APPLICATION USAGE section.

28419 **NAME**

28420 tcdrain — wait for transmission of output

28421 **SYNOPSIS**

28422 #include <termios.h>

28423 int tcdrain(int *fildev*);28424 **DESCRIPTION**28425 The *tcdrain()* function waits until all output written to the object referred to by *fildev* is
28426 transmitted. The *fildev* argument is an open file descriptor associated with a terminal.28427 Any attempts to use *tcdrain()* from a process which is a member of a background process group
28428 on a *fildev* associated with its controlling terminal, will cause the process group to be sent a
28429 SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is
28430 allowed to perform the operation, and no signal is sent.28431 **RETURN VALUE**28432 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate
28433 the error.28434 **ERRORS**28435 The *tcdrain()* function will fail if:28436 [EBADF] The *fildev* argument is not a valid file descriptor.28437 [EINTR] A signal interrupted *tcdrain()*.28438 [ENOTTY] The file associated with *fildev* is not a terminal.28439 The *tcdrain()* function may fail if:28440 EX [EIO] The process group of the writing process is orphaned, and the writing process
28441 is not ignoring or blocking SIGTTOU.28442 **EXAMPLES**

28443 None.

28444 **APPLICATION USAGE**

28445 None.

28446 **FUTURE DIRECTIONS**28447 In the ISO POSIX-1 standard, the possibility of an [EIO] error occurring is described in , **Section**
28448 **9.1.4, Terminal Access Control**, but it is not mentioned in the *tcdrain()* interface definition. It
28449 has become clear that this omission was unintended, so it is likely that the [EIO] error will be
28450 reclassified as a “will fail” in a future issue of the POSIX standard.28451 **SEE ALSO**28452 *tcflush()*, <termios.h>, <unistd.h>, the XBD specification, **Chapter 9, General Terminal**
28453 **Interface**.28454 **CHANGE HISTORY**

28455 First released in Issue 3.

28456 Entry included for alignment with the POSIX.1-1988 standard.

28457 **Issue 4**

28458 The following change is incorporated for alignment with the FIPS requirements:

- 28459 • The words “If _POSIX_JOB_CONTROL is defined” are removed from the start of the second
28460 paragraph in the DESCRIPTION. This is because job control is defined as mandatory for
28461 Issue 4 conforming implementations.

28462 Other changes are incorporated as follows:

- 28463 • The [EIO] error is added to the ERRORS section.
28464 • The FUTURE DIRECTIONS section is added.

28465 **NAME**

28466 tcflow — suspend or restart the transmission or reception of data

28467 **SYNOPSIS**

28468 #include <termios.h>

28469 int tcflow(int *fildev*, int *action*);28470 **DESCRIPTION**

28471 The *tcflow()* function suspends transmission or reception of data on the object referred to by
 28472 *fildev*, depending on the value of *action*. The *fildev* argument is an open file descriptor associated
 28473 with a terminal.

- 28474 • If *action* is TCOOFF, output is suspended.
- 28475 • If *action* is TCOON, suspended output is restarted.
- 28476 • If *action* is TCIOFF, the system transmits a STOP character, which is intended to cause the
 28477 terminal device to stop transmitting data to the system.
- 28478 • If *action* is TCION, the system transmits a START character, which is intended to cause the
 28479 terminal device to start transmitting data to the system.

28480 The default on the opening of a terminal file is that neither its input nor its output are
 28481 suspended.

28482 Attempts to use *tcflow()* from a process which is a member of a background process group on a
 28483 *fildev* associated with its controlling terminal, will cause the process group to be sent a SIGTTOU
 28484 signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is allowed to
 28485 perform the operation, and no signal is sent.

28486 **RETURN VALUE**

28487 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate
 28488 the error.

28489 **ERRORS**28490 The *tcflow()* function will fail if:

- 28491 [EBADF] The *fildev* argument is not a valid file descriptor.
- 28492 [EINVAL] The *action* argument is not a supported value.
- 28493 [ENOTTY] The file associated with *fildev* is not a terminal.

28494 The *tcflow()* function may fail if:

- 28495 EX [EIO] The process group of the writing process is orphaned, and the writing process
 28496 is not ignoring or blocking SIGTTOU.

28497 **EXAMPLES**

28498 None.

28499 **APPLICATION USAGE**

28500 None.

28501 **FUTURE DIRECTIONS**

28502 In the ISO POSIX-1 standard, the possibility of an [EIO] error occurring is described in , **Section**
 28503 **9.1.4, Terminal Access Control**, but it is not mentioned in the *tcflow()* interface definition. It has
 28504 become clear that this omission was unintended, so it is likely that the [EIO] error will be re-
 28505 classified as a “will fail” in a future issue of the POSIX standard.

28506 **SEE ALSO**

28507 *tcsendbreak()*, `<termios.h>`, `<unistd.h>`, the **XBD** specification, **Chapter 9, General Terminal**
 28508 **Interface**.

28509 **CHANGE HISTORY**

28510 First released in Issue 3.

28511 Entry included for alignment with the POSIX.1-1988 standard.

28512 **Issue 4**

28513 The following change is incorporated for alignment with the FIPS requirements:

- 28514 • The words “If `_POSIX_JOB_CONTROL` is defined” are removed from the start of the second
 28515 paragraph in the DESCRIPTION. This is because job control is defined as mandatory for
 28516 Issue 4 conforming implementations.

28517 Other changes are incorporated as follows:

- 28518 • The descriptions of `TCIOFF` and `TCION` are reworded, indicating the intended consequences
 28519 of transmitting stop and start characters. Issue 3 implied that these consequences were
 28520 guaranteed.
- 28521 • The `[EIO]` error is added to the ERRORS section.
- 28522 • The FUTURE DIRECTIONS section is added.

28523 **NAME**

28524 tcflush — flush non-transmitted output data, non-read input data or both

28525 **SYNOPSIS**

28526 #include <termios.h>

28527 int tcflush(int *fildes*, int *queue_selector*);28528 **DESCRIPTION**

28529 Upon successful completion, *tcflush()* discards data written to the object referred to by *fildes* (an
 28530 open file descriptor associated with a terminal) but not transmitted, or data received but not
 28531 read, depending on the value of *queue_selector*:

- 28532 • If *queue_selector* is TCIFLUSH it flushes data received but not read.
- 28533 • If *queue_selector* is TCOFLUSH it flushes data written but not transmitted.
- 28534 • If *queue_selector* is TCIOFLUSH it flushes both data received but not read and data written
 28535 but not transmitted.

28536 **FIPS** Attempts to use *tcflush()* from a process which is a member of a background process group on a
 28537 *fildes* associated with its controlling terminal, will cause the process group to be sent a SIGTTOU
 28538 signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is allowed to
 28539 perform the operation, and no signal is sent.

28540 **RETURN VALUE**

28541 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate
 28542 the error.

28543 **ERRORS**28544 The *tcflush()* function will fail if:

- 28545 [EBADF] The *fildes* argument is not a valid file descriptor.
- 28546 [EINVAL] The *queue_selector* argument is not a supported value.
- 28547 [ENOTTY] The file associated with *fildes* is not a terminal.

28548 The *tcflow()* function may fail if:

- 28549 **EX** [EIO] The process group of the writing process is orphaned, and the writing process
 28550 is not ignoring or blocking SIGTTOU.

28551 **EXAMPLES**

28552 None.

28553 **APPLICATION USAGE**

28554 None.

28555 **FUTURE DIRECTIONS**

28556 In the ISO POSIX-1 standard, the possibility of an [EIO] error occurring is described in , **Section**
 28557 **9.1.4, Terminal Access Control**, but it is not mentioned in the *tcflow()* interface definition. It has
 28558 become clear that this omission was unintended, so it is likely that the [EIO] error will be
 28559 reclassified as a “will fail” in a future issue of the POSIX standard.

28560 **SEE ALSO**

28561 *tcdrain()*, <termios.h>, <unistd.h>, the XBD specification, **Chapter 9, General Terminal**
 28562 **Interface**.

28563 **CHANGE HISTORY**

28564 First released in Issue 3.

28565 Entry included for alignment with the POSIX.1-1988 standard.

28566 **Issue 4**

28567 The following change is incorporated for alignment with the FIPS requirements:

- 28568 • The words “If _POSIX_JOB_CONTROL is defined” are removed from the start of the second |
28569 paragraph in the DESCRIPTION. This is because job control is defined as mandatory for
28570 Issue 4 conforming implementations.

28571 Other changes are incorporated as follows:

- 28572 • The DESCRIPTION is modified to indicate that the flush operation will only result if the call |
28573 to *tcflush()* is successful.
- 28574 • The [EIO] error is added to the ERRORS section. |
- 28575 • The FUTURE DIRECTIONS section is added. |

28576 **NAME**

28577 tcgetattr — get the parameters associated with the terminal

28578 **SYNOPSIS**

28579 #include <termios.h>

28580 int tcgetattr(int *fildev*, struct termios **termios_p*);

28581 **DESCRIPTION**

28582 The *tcgetattr()* function gets the parameters associated with the terminal referred to by *fildev* and stores them in the **termios** structure referenced by *termios_p*. The *fildev* argument is an open file descriptor associated with a terminal.

28585 The *termios_p* argument is a pointer to a **termios** structure.

28586 The *tcgetattr()* operation is allowed from any process.

28587 If the terminal device supports different input and output baud rates, the baud rates stored in the **termios** structure returned by *tcgetattr()* reflect the actual baud rates, even if they are equal. If differing baud rates are not supported, the rate returned as the output baud rate is the actual baud rate. If the terminal device does not support split baud rates, the input baud rate stored in the **termios** structure will be 0.

28592 **RETURN VALUE**

28593 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

28595 **ERRORS**

28596 The *tcgetattr()* function will fail if:

28597 [EBADF] The *fildev* argument is not a valid file descriptor.

28598 [ENOTTY] The file associated with *fildev* is not a terminal.

28599 **EXAMPLES**

28600 None.

28601 **APPLICATION USAGE**

28602 None.

28603 **FUTURE DIRECTIONS**

28604 In a future issue of this document, implementations which do not support differing baud rates will be prohibited from returning 0 as the input baud rate.

28606 **SEE ALSO**

28607 *tcsetattr()*, <termios.h>, the XBD specification, **Chapter 9, General Terminal Interface**.

28608 **CHANGE HISTORY**

28609 First released in Issue 3.

28610 Entry included for alignment with the POSIX.1-1988 standard.

28611 **Issue 4**

28612 The following change is incorporated in this issue:

- 28613 • The FUTURE DIRECTIONS section is added to allow for alignment with the ISO POSIX-1
- 28614 standard.

28615 **NAME**

28616 tcgetpgrp — get the foreground process group ID

28617 **SYNOPSIS**

28618 OH #include <sys/types.h>

28619 #include <unistd.h>

28620 pid_t tcgetpgrp(int *fildev*);28621 **DESCRIPTION**28622 FIPS The *tcgetpgrp()* function will return the value of the process group ID of the foreground process group associated with the terminal.28624 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.28626 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.28629 **RETURN VALUE**28630 Upon successful completion, *tcgetpgrp()* returns the value of the process group ID of the foreground process associated with the terminal. Otherwise, -1 is returned and *errno* is set to indicate the error.28633 **ERRORS**28634 The *tcgetpgrp()* function will fail if:28635 [EBADF] The *fildev* argument is not a valid file descriptor.

28636 [ENOTTY] The calling process does not have a controlling terminal, or the file is not the controlling terminal.

28638 **EXAMPLES**

28639 None.

28640 **APPLICATION USAGE**

28641 None.

28642 **FUTURE DIRECTIONS**

28643 None.

28644 **SEE ALSO**28645 *setsid()*, *setpgid()*, *tcsetpgrp()*, <sys/types.h>, <unistd.h>.28646 **CHANGE HISTORY**

28647 First released in Issue 3.

28648 Entry included for alignment with the POSIX.1-1988 standard.

28649 **Issue 4**

28650 The following change is incorporated for alignment with the FIPS requirements:

- 28651 • The DESCRIPTION is clarified and the phrase “If _POSIX_JOB_CONTROL is defined” is
- 28652 removed because job control is now mandatory on all XSI-conformant systems.

- | | | |
|-------|--|--|
| 28653 | Other changes are incorporated as follows: | |
| 28654 | • The < sys/types.h > header is now marked as optional (OH); this header need not be included | |
| 28655 | on XSI-conformant systems. | |
| 28656 | • The < unistd.h > header is added to the SYNOPSIS section. | |

28657 **NAME**

28658 tcgetsid — get process group ID for session leader for controlling terminal

28659 **SYNOPSIS**28660 EX `#include <termios.h>`28661 `pid_t tcgetsid(int fildes);`

28662

28663 **DESCRIPTION**28664 The *tcgetsid()* function obtains the process group ID of the session for which the terminal
28665 specified by *fildes* is the controlling terminal.28666 **RETURN VALUE**28667 Upon successful completion, *tcgetsid()* returns the process group ID associated with the
28668 terminal. Otherwise, a value of (**pid_t**)−1 is returned and *errno* is set to indicate the error.28669 **ERRORS**28670 The *tcgetsid()* function will fail if:28671 [EBADF] The *fildes* argument is not a valid file descriptor.28672 [ENOTTY] The calling process does not have a controlling terminal, or the file is not the
28673 controlling terminal.28674 **EXAMPLES**

28675 None.

28676 **APPLICATION USAGE**

28677 None.

28678 **FUTURE DIRECTIONS**

28679 None.

28680 **SEE ALSO**28681 `<termios.h>`.28682 **CHANGE HISTORY**

28683 First released in Issue 4, Version 2.

28684 **Issue 5**

28685 Moved from X/OPEN UNIX extension to BASE.

28686 The [EACCES] error has been removed from the list of mandatory errors, and the description of
28687 [ENOTTY] has been reworded.

28688 **NAME**

28689 tcsendbreak — send a “break” for a specific duration

28690 **SYNOPSIS**

28691 #include <termios.h>

28692 int tcsendbreak(int *fildev*, int *duration*);28693 **DESCRIPTION**28694 The *fildev* argument is an open file descriptor associated with a terminal.

28695 If the terminal is using asynchronous serial data transmission, *tcsendbreak()* will cause
 28696 transmission of a continuous stream of zero-valued bits for a specific duration. If *duration* is 0, it
 28697 will cause transmission of zero-valued bits for at least 0.25 seconds, and not more than 0.5
 28698 seconds. If *duration* is not 0, it will send zero-valued bits for an implementation-dependent
 28699 period of time.

28700 If the terminal is not using asynchronous serial data transmission, it is implementation-
 28701 dependent whether *tcsendbreak()* sends data to generate a break condition or returns without
 28702 taking any action.

28703 **FIPS** Attempts to use *tcsendbreak()* from a process which is a member of a background process group
 28704 on a *fildev* associated with its controlling terminal, will cause the process group to be sent a
 28705 SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is
 28706 allowed to perform the operation, and no signal is sent.

28707 **RETURN VALUE**

28708 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate
 28709 the error.

28710 **ERRORS**28711 The *tcsendbreak()* function will fail if:28712 [EBADF] The *fildev* argument is not a valid file descriptor.28713 [ENOTTY] The file associated with *fildev* is not a terminal.28714 The *tcsendbreak()* function may fail if:

28715 **EX** [EIO] The process group of the writing process is orphaned, and the writing process
 28716 is not ignoring or blocking SIGTTOU.

28717 **EXAMPLES**

28718 None.

28719 **APPLICATION USAGE**

28720 None.

28721 **FUTURE DIRECTIONS**

28722 In the ISO POSIX-1 standard, the possibility of an [EIO] error occurring is described in , **Section**
 28723 **9.1.4, Terminal Access Control**, but it is not mentioned in the *tcsendbreak()* interface definition.
 28724 It has become clear that this omission was unintended, so it is likely that the [EIO] error will be
 28725 reclassified as a “will fail” in a future issue of the POSIX standard.

28726 **SEE ALSO**28727 <termios.h>, <unistd.h>, the XBD specification, **Chapter 9, General Terminal Interface**.28728 **CHANGE HISTORY**

28729 First released in Issue 3.

28730 Entry included for alignment with the POSIX.1-1988 standard.

28731 **Issue 4**

28732 The following change is incorporated for alignment with the FIPS requirements:

- 28733 • In the DESCRIPTION the phrase “If _POSIX_JOB_CONTROL is defined” is removed |
28734 because job control is now mandatory on all XSI-conformant systems.

28735 Another change is incorporated as follows:

- 28736 • The [EIO] error is added to the ERRORS section. |

28737 NAME

28738 tcsetattr — set the parameters associated with the terminal

28739 SYNOPSIS

28740 #include <termios.h>

```
28741 int tcsetattr(int fildes, int optional_actions,
28742               const struct termios *termios_p);
```

28743 DESCRIPTION

28744 The *tcsetattr()* function sets the parameters associated with the terminal referred to by the open
 28745 file descriptor *fildes* (an open file descriptor associated with a terminal) from the **termios**
 28746 structure referenced by *termios_p* as follows:

- 28747 • If *optional_actions* is TCSANOW, the change will occur immediately.
- 28748 • If *optional_actions* is TCSADRAIN, the change will occur after all output written to *fildes* is
 28749 transmitted. This function should be used when changing parameters that affect output.
- 28750 • If *optional_actions* is TCSAFLUSH, the change will occur after all output written to *fildes* is
 28751 transmitted, and all input so far received but not read will be discarded before the change is
 28752 made.

28753 If the output baud rate stored in the **termios** structure pointed to by *termios_p* is the zero baud
 28754 rate, B0, the modem control lines will no longer be asserted. Normally, this will disconnect the
 28755 line.

28756 If the input baud rate stored in the **termios** structure pointed to by *termios_p* is 0, the input baud
 28757 rate given to the hardware will be the same as the output baud rate stored in the **termios**
 28758 structure.

28759 The *tcsetattr()* function will return successfully if it was able to perform any of the requested
 28760 actions, even if some of the requested actions could not be performed. It will set all the attributes
 28761 that implementation supports as requested and leave all the attributes not supported by the
 28762 implementation unchanged. If no part of the request can be honoured, it will return `-1` and set
 28763 *errno* to `[EINVAL]`. If the input and output baud rates differ and are a combination that is not
 28764 supported, neither baud rate is changed. A subsequent call to *tcgetattr()* will return the actual
 28765 state of the terminal device (reflecting both the changes made and not made in the previous
 28766 *tcsetattr()* call). The *tcsetattr()* function will not change the values in the **termios** structure
 28767 whether or not it actually accepts them.

28768 The effect of *tcsetattr()* is undefined if the value of the **termios** structure pointed to by *termios_p*
 28769 was not derived from the result of a call to *tcgetattr()* on *fildes*; an application should modify
 28770 only fields and flags defined by this specification between the call to *tcgetattr()* and *tcsetattr()*,
 28771 leaving all other fields and flags unmodified.

28772 No actions defined by this specification, other than a call to *tcsetattr()* or a close of the last file
 28773 descriptor in the system associated with this terminal device, will cause any of the terminal
 28774 attributes defined by this specification to change.

28775 FIPS Attempts to use *tcsetattr()* from a process which is a member of a background process group on
 28776 a *fildes* associated with its controlling terminal, will cause the process group to be sent a
 28777 SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is
 28778 allowed to perform the operation, and no signal is sent.

28779 RETURN VALUE

28780 Upon successful completion, 0 is returned. Otherwise, `-1` is returned and *errno* is set to indicate
 28781 the error.

28782 **ERRORS**28783 The *tcsetattr()* function will fail if:28784 [EBADF] The *fildev* argument is not a valid file descriptor.28785 [EINTR] A signal interrupted *tcsetattr()*.28786 [EINVAL] The *optional_actions* argument is not a supported value, or an attempt was
28787 made to change an attribute represented in the **termios** structure to an
28788 unsupported value.28789 [ENOTTY] The file associated with *fildev* is not a terminal.28790 The *tcsetattr()* function may fail if:28791 EX [EIO] The process group of the writing process is orphaned, and the writing process
28792 is not ignoring or blocking SIGTTOU.28793 **EXAMPLES**

28794 None.

28795 **APPLICATION USAGE**28796 If trying to change baud rates, applications should call *tcsetattr()* then call *tcgetattr()* in order to
28797 determine what baud rates were actually selected.28798 **FUTURE DIRECTIONS**28799 Using an input baud rate of 0 to set the input rate equal to the output rate will not necessarily be
28800 supported in future issues of this document.28801 In the ISO POSIX-1 standard, the possibility of an [EIO] error occurring is described in , **Section**
28802 **9.1.4, Terminal Access Control**, but it is not mentioned in the *tcsetattr()* interface definition. It
28803 has become clear that this omission was unintended, so it is likely that the [EIO] error will be
28804 reclassified as a “will fail” in a future issue of the POSIX standard.28805 **SEE ALSO**28806 *cfgetispeed()*, *tcgetattr()*, **<termios.h>**, **<unistd.h>**, the **XBD** specification, **Chapter 9, General**
28807 **Terminal Interface**.28808 **CHANGE HISTORY**

28809 First released in Issue 3.

28810 Entry included for alignment with the POSIX.1-1988 standard.

28811 **Issue 4**

28812 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 28813
- The argument *termios_p* is changed from type **struct termios *** to **const struct termios ***.

28814 The following change is incorporated for alignment with the FIPS requirements:

- 28815
- In the DESCRIPTION the phrase “If _POSIX_JOB_CONTROL is defined” is removed
28816 because job control is now mandatory on all XSI-conformant systems.

28817 Other changes are incorporated as follows:

- 28818
- The words “and stores them in” are changed to “from” in the first paragraph of the
28819 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
28820 standard.
- 28821
-
- 28822

28823 **NAME**

28824 tcsetpgrp — set the foreground process group ID

28825 **SYNOPSIS**

28826 OH #include <sys/types.h>

28827 #include <unistd.h>

28828 int tcsetpgrp(int *fildes*, pid_t *pgid_id*);28829 **DESCRIPTION**

28830 FIPS If the process has a controlling terminal, *tcsetpgrp()* will set the foreground process group ID
 28831 associated with the terminal to *pgid_id*. The file associated with *fildes* must be the controlling
 28832 terminal of the calling process and the controlling terminal must be currently associated with the
 28833 session of the calling process. The value of *pgid_id* must match a process group ID of a process
 28834 in the same session as the calling process.

28835 **RETURN VALUE**

28836 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate
 28837 the error.

28838 **ERRORS**28839 The *tcsetpgrp()* function will fail if:28840 [EBADF] The *fildes* argument is not a valid file descriptor.28841 [EINVAL] This implementation does not support the value in the *pgid_id* argument.

28842 [ENOTTY] The calling process does not have a controlling terminal, or the file is not the
 28843 controlling terminal, or the controlling terminal is no longer associated with
 28844 the session of the calling process.

28845 FIPS [EPERM] The value of *pgid_id* does not match the process group ID of a process in the
 28846 same session as the calling process.

28847 **EXAMPLES**

28848 None.

28849 **APPLICATION USAGE**

28850 None.

28851 **FUTURE DIRECTIONS**

28852 None.

28853 **SEE ALSO**28854 *tcgetpgrp()*, <sys/types.h>, <unistd.h>.28855 **CHANGE HISTORY**

28856 First released in Issue 3.

28857 Entry included for alignment with the POSIX.1-1988 standard.

28858 **Issue 4**

28859 The following change is incorporated for alignment with the FIPS requirements:

- 28860 • In the DESCRIPTION the phrase “If _POSIX_JOB_CONTROL is defined” is removed
 28861 because job control is now mandatory on all XSI-conformant systems.

28862 Other changes are incorporated as follows:

- 28863 • The <sys/types.h> header is now marked as optional (OH); this header need not be included
 28864 on XSI-conformant systems.

- | | | |
|-------|--|--|
| 28865 | • The header < unistd.h > is added to the SYNOPSIS section. | |
| 28866 | • The [ENOSYS] error is removed from the ERRORS section. | |

28867 **NAME**

28868 tdelete — delete node from binary search tree

28869 **SYNOPSIS**

28870 EX #include <search.h>

```
28871       void *tdelete(const void *key, void **rootp,  
28872                    int (*compar)(const void *, const void *));
```

28873

28874 **DESCRIPTION**28875 Refer to *tsearch()*.28876 **CHANGE HISTORY**

28877 First released in Issue 1.

28878 Derived from Issue 1 of the SVID.

28879 **Issue 4**

28880 The following change is incorporated in this issue:

- 28881 • The function return value is changed from *char ** to **void***, the type of argument *key* is
28882 changed from **char *** to **const void***, *rootp* is changed from **char **** to **void****, and arguments
28883 to *compar()* are formally defined.

28884 **NAME**

28885 telldir — current location of a named directory stream

28886 **SYNOPSIS**

28887 EX #include <dirent.h>

28888 long int telldir(DIR *dirp);

28889

28890 **DESCRIPTION**28891 The *telldir()* function obtains the current location associated with the directory stream specified
28892 by *dirp*.28893 If the most recent operation on the directory stream was a *seekdir()*, the directory position
28894 returned from the *telldir()* is the same as that supplied as a *loc* argument for *seekdir()*.28895 **RETURN VALUE**28896 Upon successful completion, *telldir()* returns the current location of the specified directory
28897 stream.28898 **ERRORS**

28899 No errors are defined.

28900 **EXAMPLES**

28901 None.

28902 **APPLICATION USAGE**

28903 None.

28904 **FUTURE DIRECTIONS**

28905 None.

28906 **SEE ALSO**28907 *opendir()*, *readdir()*, *seekdir()*, <dirent.h>.28908 **CHANGE HISTORY**

28909 First released in Issue 2.

28910 **Issue 4**

28911 The following changes are incorporated in this issue:

- 28912 • The <sys/types.h> header is removed from the SYNOPSIS section.
- 28913 • The function return value is expanded to **long int**.

28914 **Issue 4, Version 2**28915 The DESCRIPTION is updated for X/OPEN UNIX conformance to indicate that a call to *telldir()*
28916 immediately following a call to *seekdir()*, returns the *loc* value passed to the *seekdir()* call.

28917 **NAME**

28918 tempnam — create a name for a temporary file

28919 **SYNOPSIS**28920 EX `#include <stdio.h>`28921 `char *tempnam(const char *dir, const char *pfx);`

28922

28923 **DESCRIPTION**28924 The *tempnam()* function generates a pathname that may be used for a temporary file.

28925 The *tempnam()* function allows the user to control the choice of a directory. The *dir* argument
 28926 points to the name of the directory in which the file is to be created. If *dir* is a null pointer or
 28927 points to a string which is not a name for an appropriate directory, the path prefix defined as
 28928 {P_tmpdir} in the <stdio.h> header is used. If that directory is not accessible, an
 28929 implementation-dependent directory may be used.

28930 Many applications prefer their temporary files to have certain initial letter sequences in their
 28931 names. The *pfx* argument should be used for this. This argument may be a null pointer or point
 28932 to a string of up to five bytes to be used as the beginning of the filename.

28933 Some implementations of *tempnam()* may use *tmpnam()* internally. On such implementations, if
 28934 called more than {TMP_MAX} times in a single process, the behaviour is implementation-
 28935 dependent.

28936 **RETURN VALUE**

28937 Upon successful completion, *tempnam()* allocates space for a string, puts the generated
 28938 pathname in that space and returns a pointer to it. The pointer is suitable for use in a
 28939 subsequent call to *free()*. Otherwise it returns a null pointer and sets *errno* to indicate the error.

28940 **ERRORS**28941 The *tempnam()* function will fail if:

28942 [ENOMEM] Insufficient storage space is available.

28943 **EXAMPLES**

28944 None.

28945 **APPLICATION USAGE**

28946 This function only creates pathnames. It is the application's responsibility to create and remove
 28947 the files. Between the time a pathname is created and the file is opened, it is possible for some
 28948 other process to create a file with the same name. Applications may find *tmpfile()* more useful.

28949 **FUTURE DIRECTIONS**

28950 None.

28951 **SEE ALSO**28952 *fopen()*, *free()*, *open()*, *tmpfile()*, *tmpnam()*, *unlink()*, <stdio.h>.28953 **CHANGE HISTORY**

28954 First released in Issue 1.

28955 Derived from Issue 1 of the SVID.

28956 **Issue 4**

28957 The following changes are incorporated in this issue:

- 28958 • The type of arguments *dir* and *pfx* is changed from **char *** to **const char ***.

28959	<ul style="list-style-type: none">• The DESCRIPTION is changed to indicate that <i>pf</i>x is treated as a string of bytes and not as a string of (possibly multi-byte) characters.	
28960		
28961	<ul style="list-style-type: none">• The second paragraph of the APPLICATION USAGE section is expanded.	
28962	Issue 5 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.	
28963		
28964		

28965 **NAME**

28966 tfind — search binary search tree

28967 **SYNOPSIS**

28968 EX #include <search.h>

```
28969       void *tfind(const void *key, void *const *rootp,  
28970                   int (*compar)(const void *, const void *));
```

28971

28972 **DESCRIPTION**28973 Refer to *tsearch()*.28974 **CHANGE HISTORY**

28975 First released in Issue 1.

28976 Derived from Issue 1 of the SVID.

28977 **Issue 4**

28978 The following changes are incorporated in this issue:

- 28979 • The function return value is changed from **char *** to **void***.
- 28980 • The type of argument *key* is changed from **char *** to **const void***; the type of argument *rootp* is
28981 changed from **char **** to **void* const***.
- 28982 • Arguments to *compar()* are formally defined.

28983 **NAME**

28984 time — get time

28985 **SYNOPSIS**

28986 #include <time.h>

28987 time_t time(time_t *tloc);

28988 **DESCRIPTION**28989 The *time()* function returns the value of time in seconds since the Epoch.

28990 The *tloc* argument points to an area where the return value is also stored. If *tloc* is a null pointer,
 28991 no value is stored.

28992 **RETURN VALUE**28993 Upon successful completion, *time()* returns the value of time. Otherwise, **(time_t)−1** is returned.28994 **ERRORS**

28995 No errors are defined.

28996 **EXAMPLES**

28997 None.

28998 **APPLICATION USAGE**

28999 None.

29000 **FUTURE DIRECTIONS**

29001 None.

29002 **SEE ALSO**

29003 *asctime()*, *clock()*, *ctime()*, *difftime()*, *gmtime()*, *localtime()*, *mktime()*, *strftime()*, *strptime()*, *utime()*,
 29004 <time.h>.

29005 **CHANGE HISTORY**

29006 First released in Issue 1.

29007 Derived from Issue 1 of the SVID.

29008 **Issue 4**

29009 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 29010 • The RETURN VALUE section is updated to indicate that **(time_t)−1** will be returned on error.

29011 NAME

29012 timer_create — create a per-process timer (**REALTIME**)

29013 SYNOPSIS

29014 RT #include <time.h>

29015 #include <signal.h>

```
29016 int timer_create(clockid_t clockid, struct sigevent *evp,
29017                 timer_t *timerid);
```

29018

29019 DESCRIPTION

29020 The *timer_create()* function creates a per-process timer using the specified clock, *clock_id*, as the
 29021 timing base. The *timer_create()* function returns, in the location referenced by *timerid*, a timer ID
 29022 of type **timer_t** used to identify the timer in timer requests. This timer ID will be unique within
 29023 the calling process until the timer is deleted. The particular clock, *clock_id*, is defined in
 29024 <time.h>. The timer whose ID is returned will be in a disarmed state upon return from
 29025 *timer_create()*.

29026 The *evp* argument, if non-NULL, points to a **sigevent** structure. This structure, allocated by the
 29027 application, defines the asynchronous notification to occur as specified in **Signal Generation**
 29028 **and Delivery** on page 808 when the timer expires. If the *evp* argument is NULL, the effect is as if
 29029 the *evp* argument pointed to a *sigevent* structure with the *sigev_notify* member having the value
 29030 SIGEV_SIGNAL, the *sigev_signo* having a default signal number, and the *sigev_value* member
 29031 having the value of the timer ID.

29032 Each implementation defines a set of clocks that can be used as timing bases for per-process
 29033 timers. All implementations support a *clock_id* of CLOCK_REALTIME.

29034 Per-process timers are not inherited by a child process across a *fork()* and are disarmed and
 29035 deleted by an *exec*.

29036 RETURN VALUE

29037 If the call succeeds, *timer_create()* returns zero and updates the location referenced by *timerid* to a
 29038 **timer_t**, which can be passed to the per-process timer calls. If an error occurs, the function
 29039 returns a value of -1 and sets *errno* to indicate the error. The value of *timerid* is undefined if an
 29040 error occurs.

29041 ERRORS

29042 The *timer_create()* function will fail if:

29043 [EAGAIN] The system lacks sufficient signal queuing resources to honour the request.

29044 [EAGAIN] The calling process has already created all of the timers it is allowed by this
 29045 implementation.

29046 [EINVAL] The specified clock ID is not defined.

29047 [ENOSYS] The function *timer_create()* is not supported by this implementation.

29048 EXAMPLES

29049 None.

29050 APPLICATION USAGE

29051 None.

29052 FUTURE DIRECTIONS

29053 None.

29054 **SEE ALSO**

29055 *timer_delete()*, *clock_gettime()*, *clock_settime()*, *clock_getres()*, *timer_gettime()*, *timer_settime()*,
29056 **<time.h>**.

29057 **CHANGE HISTORY**

29058 First released in Issue 5.

29059 Included for alignment with the POSIX Realtime Extension.

29060 **NAME**

29061 timer_delete — delete a per-process timer (**REALTIME**)

29062 **SYNOPSIS**

29063 RT `#include <time.h>`

29064 `int timer_delete(timer_t timerid);`

29065

29066 **DESCRIPTION**

29067 The *timer_delete()* function deletes the specified timer, *timerid*, previously created by the
29068 *timer_create()* function. If the timer is armed when *timer_delete()* is called, the behaviour will be
29069 as if the timer is automatically disarmed before removal. The disposition of pending signals for
29070 the deleted timer is unspecified.

29071 **RETURN VALUE**

29072 If successful, the function returns a value of zero. Otherwise, the function returns a value of -1
29073 and sets *errno* to indicate the error.

29074 **ERRORS**

29075 The *timer_delete()* function will fail if:

29076 [EINVAL] The timer ID specified by *timerid* is not a valid timer ID.

29077 [ENOSYS] The function *timer_delete()* is not supported by this implementation.

29078 **EXAMPLES**

29079 None.

29080 **APPLICATION USAGE**

29081 None.

29082 **FUTURE DIRECTIONS**

29083 None.

29084 **SEE ALSO**

29085 *timer_create()*, **<time.h>**.

29086 **CHANGE HISTORY**

29087 First released in Issue 5.

29088 Included for alignment with the POSIX Realtime Extension.

29089 **NAME**29090 timer_settime, timer_gettime, timer_getoverrun — per-process timers (**REALTIME**)29091 **SYNOPSIS**29092 RT `#include <time.h>`

```

29093 int timer_settime(timer_t timerid, int flags,
29094     const struct itimerspec *value, struct itimerspec *ovalue);
29095 int timer_gettime(timer_t timerid, struct itimerspec *value);
29096 int timer_getoverrun(timer_t timerid);
29097

```

29098 **DESCRIPTION**

29099 The *timer_settime()* function sets the time until the next expiration of the timer specified by
 29100 *timerid* from the *it_value* member of the *value* argument and arm the timer if the *it_value* member
 29101 of *value* is non-zero. If the specified timer was already armed when *timer_settime()* is called, this
 29102 call resets the time until next expiration to the *value* specified. If the *it_value* member of *value* is
 29103 zero, the timer is disarmed. The effect of disarming or resetting a timer on pending expiration
 29104 notifications is unspecified.

29105 If the flag **TIMER_ABSTIME** is not set in the argument *flags*, *timer_settime()* behaves as if the
 29106 time until next expiration is set to be equal to the interval specified by the *it_value* member of
 29107 *value*. That is, the timer expires in *it_value* nanoseconds from when the call is made. If the flag
 29108 **TIMER_ABSTIME** is set in the argument *flags*, *timer_settime()* behaves as if the time until next
 29109 expiration is set to be equal to the difference between the absolute time specified by the *it_value*
 29110 member of *value* and the current value of the clock associated with *timerid*. That is, the timer
 29111 expires when the clock reaches the value specified by the *it_value* member of *value*. If the
 29112 specified time has already passed, the function succeeds and the expiration notification is made.

29113 The reload value of the timer is set to the value specified by the *it_interval* member of *value*.
 29114 When a timer is armed with a non-zero *it_interval*, a periodic (or repetitive) timer is specified.

29115 Time values that are between two consecutive non-negative integer multiples of the resolution
 29116 of the specified timer will be rounded up to the larger multiple of the resolution. Quantization
 29117 error will not cause the timer to expire earlier than the rounded time value.

29118 If the argument *ovalue* is not NULL, the function *timer_settime()* stores, in the location referenced
 29119 by *ovalue*, a value representing the previous amount of time before the timer would have
 29120 expired or zero if the timer was disarmed, together with the previous timer reload value. The
 29121 members of *ovalue* are subject to the resolution of the timer, and they are the same values that
 29122 would be returned by a *timer_gettime()* call at that point in time.

29123 The *timer_gettime()* function stores the amount of time until the specified timer, *timerid*, expires
 29124 and the reload value of the timer into the space pointed to by the *value* argument. The *it_value*
 29125 member of this structure contains the amount of time before the timer expires, or zero if the
 29126 timer is disarmed. This value is returned as the interval until timer expiration, even if the timer
 29127 was armed with absolute time. The *it_interval* member of *value* contains the reload value last set
 29128 by *timer_settime()*.

29129 Only a single signal will be queued to the process for a given timer at any point in time. When a
 29130 timer for which a signal is still pending expires, no signal will be queued, and a timer overrun
 29131 occurs. When a timer expiration signal is delivered to or accepted by a process, if the
 29132 implementation supports the Realtime Signals Extension, the *timer_getoverrun()* function returns
 29133 the timer expiration overrun count for the specified timer. The overrun count returned contains
 29134 the number of extra timer expirations that occurred between the time the signal was generated
 29135 (queued) and when it was delivered or accepted, up to but not including an implementation-
 29136 dependent maximum of {**DELAYTIMER_MAX**}. If the number of such extra expirations is

greater than or equal to {DELAYTIMER_MAX}, then the overrun count will be set to {DELAYTIMER_MAX}. The value returned by *timer_getoverrun()* applies to the most recent expiration signal delivery or acceptance for the timer. If no expiration signal has been delivered for the timer, or if the Realtime Signals Extension is not supported, the meaning of the overrun count returned is undefined.

29142 RETURN VALUE

If the *timer_settime()* or *timer_gettime()* functions succeed, a value of 0 is returned. If an error occurs for either of these functions, the value -1 is returned, and *errno* is set to indicate the error. If the *timer_getoverrun()* function succeeds, it returns the timer expiration overrun count as explained above.

29147 ERRORS

The *timer_settime()*, *timer_gettime()* and *timer_getoverrun()* functions will fail if:

[EINVAL] The *timerid* argument does not correspond to an id returned by *timer_create()* but not yet deleted by *timer_delete()*.

[ENOSYS] The functions *timer_settime()*, *timer_gettime()*, and *timer_getoverrun()* are not supported by this implementation.

The *timer_settime()* function will fail if:

[EINVAL] A *value* structure specified a nanosecond value less than zero or greater than or equal to 1000 million.

29156 EXAMPLES

None.

29158 APPLICATION USAGE

None.

29160 FUTURE DIRECTIONS

None.

29162 SEE ALSO

clock_gettime(), *timer_create()*, <time.h>.

29164 CHANGE HISTORY

First released in Issue 5.

Included for alignment with the POSIX Realtime Extension.

29167 **NAME**

29168 times — get process and waited-for child process times

29169 **SYNOPSIS**

29170 #include <sys/times.h>

29171 clock_t times(struct tms *buffer);

29172 **DESCRIPTION**29173 The *times()* function fills the **tms** structure pointed to by *buffer* with time-accounting
29174 information. The structure **tms** is defined in <sys/times.h>.

29175 All times are measured in terms of the number of clock ticks used.

29176 The times of a terminated child process are included in the **tms_cutime** and **tms_cstime**
29177 elements of the parent when *wait()* or *waitpid()* returns the process ID of this terminated child.
29178 If a child process has not waited for its children, their times will not be included in its times.29179 • The **tms_utime** structure member is the CPU time charged for the execution of user
29180 instructions of the calling process.29181 • The **tms_stime** structure member is the CPU time charged for execution by the system on
29182 behalf of the calling process.29183 • The **tms_cutime** structure member is the sum of the **tms_utime** and **tms_cutime** times of the
29184 child processes.29185 • The **tms_cstime** structure member is the sum of the **tms_stime** and **tms_cstime** times of the
29186 child processes.29187 **RETURN VALUE**29188 Upon successful completion, *times()* returns the elapsed real time, in clock ticks, since an
29189 arbitrary point in the past (for example, system start-up time). This point does not change from
29190 one invocation of *times()* within the process to another. The return value may overflow the
29191 possible range of type **clock_t**. If *times()* fails, (**clock_t**)−1 is returned and *errno* is set to indicate
29192 the error.29193 **ERRORS**

29194 No errors are defined.

29195 **EXAMPLES**

29196 None.

29197 **APPLICATION USAGE**29198 Applications should use *sysconf(_SC_CLK_TCK)* to determine the number of clock ticks per
29199 second as it may vary from system to system.29200 **FUTURE DIRECTIONS**

29201 None.

29202 **SEE ALSO**29203 *exec*, *fork()*, *sysconf()*, *time()*, *wait()*, <sys/times.h>.29204 **CHANGE HISTORY**

29205 First released in Issue 1.

29206 Derived from Issue 1 of the SVID.

29207 **Issue 4**

29208 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- 29209 • All references to the constant {CLK_TCK} are removed.
- 29210 • The RETURN VALUE section is updated to indicate that (**clock_t**)−1 will be returned on
- 29211 error. |

29212 **NAME**

29213 `timezone` — difference from UTC and local standard time

29214 **SYNOPSIS**

29215 EX `#include <time.h>`

29216 `extern long int timezone;`

29217

29218 **DESCRIPTION**

29219 Refer to `tzset()`.

29220 **CHANGE HISTORY**

29221 First released in Issue 1.

29222 Derived from Issue 1 of the SVID.

29223 **Issue 4**

29224 The following changes are incorporated in this issue:

- 29225 • In the NAME section, “GMT” is changed to “UTC”.
- 29226 • The interface is marked as an extension.
- 29227 • The type of *timezone* is expanded to **extern long int**.

29228 **NAME**

29229 tmpfile — create a temporary file

29230 **SYNOPSIS**

29231 #include <stdio.h>

29232 FILE *tmpfile(void);

29233 **DESCRIPTION**

29234 The *tmpfile()* function creates a temporary file and opens a corresponding stream. The file will
 29235 automatically be deleted when all references to the file are closed. The file is opened as in
 29236 *fopen()* for update (w+).

29237 EX The largest value that can be represented correctly in an object of type **off_t** will be established
 29238 as the offset maximum in the open file description.

29239 If the process is killed in the period between file creation and unlinking, a permanent file may be
 29240 left behind.

29241 An error message may be written to standard error if the stream cannot be opened.

29242 **RETURN VALUE**

29243 Upon successful completion, *tmpfile()* returns a pointer to the stream of the file that is created.

29244 Otherwise, it returns a null pointer and sets *errno* to indicate the error.

29245 **ERRORS**

29246 The *tmpfile()* function will fail if:

29247 [EINTR] A signal was caught during *tmpfile()*.

29248 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

29249 [ENFILE] The maximum allowable number of files is currently open in the system.

29250 [ENOSPC] The directory or file system which would contain the new file cannot be
 29251 expanded.

29252 EX [EOVERFLOW] The file is a regular file and the size of the file cannot be represented correctly
 29253 in an object of type **off_t**.

29254 The *tmpfile()* function may fail if:

29255 EX [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

29256 [ENOMEM] Insufficient storage space is available.

29257 **EXAMPLES**

29258 None.

29259 **APPLICATION USAGE**

29260 None.

29261 **FUTURE DIRECTIONS**

29262 None.

29263 **SEE ALSO**

29264 *fopen()*, *tmpnam()*, *unlink()*, <stdio.h>.

29265 **CHANGE HISTORY**

29266 First released in Issue 1.

29267 Derived from Issue 1 of the SVID.

29268 **Issue 4**

29269 The following changes are incorporated in this issue:

- 29270 • The argument list is explicitly defined as **void**.
- 29271 • The [EINTR] error is moved to the “will fail” part of the ERRORS section; [EMFILE],
- 29272 [ENFILE] and [ENOSPC] are no longer marked as extensions; [EACCES], [ENOTDIR] and
- 29273 [EROFS] are removed; and the [EMFILE] error in the “may fail” part is marked as an
- 29274 extension.

29275 **Issue 5**

29276 Large File Summit extensions added.

29277 The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes
29278 in previous issues.

29279 **NAME**

29280 tmpnam — create a name for a temporary file

29281 **SYNOPSIS**

29282 #include <stdio.h>

29283 char *tmpnam(char *s);

29284 **DESCRIPTION**29285 The *tmpnam()* function generates a string that is a valid filename and that is not the same as the
29286 name of an existing file.29287 The *tmpnam()* function generates a different string each time it is called from the same process,
29288 up to {TMP_MAX} times. If it is called more than {TMP_MAX} times, the behaviour is
29289 implementation-dependent.29290 The implementation will behave as if no function defined in this document calls *tmpnam()*.29291 If the application uses any of the interfaces guaranteed to be available if either
29292 _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS is defined, the *tmpnam()*
29293 function must be called with a non-NULL parameter.29294 **RETURN VALUE**29295 Upon successful completion, *tmpnam()* returns a pointer to a string.29296 If the argument *s* is a null pointer, *tmpnam()* leaves its result in an internal static object and
29297 returns a pointer to that object. Subsequent calls to *tmpnam()* may modify the same object. If
29298 the argument *s* is not a null pointer, it is presumed to point to an array of at least {L_tmpnam}
29299 chars; *tmpnam()* writes its result in that array and returns the argument as its value.29300 **ERRORS**

29301 No errors are defined.

29302 **EXAMPLES**

29303 None.

29304 **APPLICATION USAGE**29305 This function only creates filenames. It is the application's responsibility to create and remove
29306 the files.29307 Between the time a pathname is created and the file is opened, it is possible for some other
29308 process to create a file with the same name. Applications may find *tmpfile()* more useful.29309 **FUTURE DIRECTIONS**

29310 None.

29311 **SEE ALSO**29312 *fopen()*, *open()*, *tmpnam()*, *tmpfile()*, *unlink()*, <stdio.h>.29313 **CHANGE HISTORY**

29314 First released in Issue 1.

29315 Derived from Issue 1 of the SVID.

29316 **Issue 5**

29317 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

29318 **NAME**

29319 toascii — translate integer to a 7-bit ASCII character

29320 **SYNOPSIS**

29321 EX #include <ctype.h>

29322 int toascii(int c);

29323

29324 **DESCRIPTION**29325 The *toascii()* function converts its argument into a 7-bit ASCII character.29326 **RETURN VALUE**29327 The *toascii()* function returns the value (*c* & 0x7f).29328 **ERRORS**

29329 No errors are returned.

29330 **EXAMPLES**

29331 None.

29332 **APPLICATION USAGE**

29333 None.

29334 **FUTURE DIRECTIONS**

29335 None.

29336 **SEE ALSO**29337 *isascii()*, <ctype.h>.29338 **CHANGE HISTORY**

29339 First released in Issue 1.

29340 Derived from Issue 1 of the SVID.

29341 **NAME**

29342 _toupper — transliterate upper-case characters to lower-case

29343 **SYNOPSIS**

29344 EX #include <ctype.h>

29345 int _tolower(int c);

29346

29347 **DESCRIPTION**29348 The *_tolower()* macro is equivalent to *tolower(c)* except that the argument *c* must be an upper-
29349 case letter.29350 **RETURN VALUE**29351 On successful completion, *_tolower()* returns the lower-case letter corresponding to the
29352 argument passed.29353 **ERRORS**

29354 No errors are defined.

29355 **EXAMPLES**

29356 None.

29357 **APPLICATION USAGE**

29358 None.

29359 **FUTURE DIRECTIONS**

29360 None.

29361 **SEE ALSO**29362 *tolower()*, *isupper()*, <ctype.h>, the XBD specification, **Chapter 5, Locale**.29363 **CHANGE HISTORY**

29364 First released in Issue 1.

29365 Derived from Issue 1 of the SVID.

29366 **Issue 4**

29367 The following change is incorporated in this issue:

- 29368
- The RETURN VALUE section is expanded.

29369 **NAME**

29370 tolower — transliterate upper-case characters to lower-case

29371 **SYNOPSIS**

29372 #include <ctype.h>

29373 int tolower(int c);

29374 **DESCRIPTION**

29375 The *tolower()* function has as a domain a type **int**, the value of which is representable as an
 29376 **unsigned char** or the value of EOF. If the argument has any other value, the behaviour is
 29377 undefined. If the argument of *tolower()* represents an upper-case letter, and there exists a
 29378 corresponding lower-case letter (as defined by character type information in the program locale
 29379 category LC_CTYPE), the result is the corresponding lower-case letter. All other arguments in
 29380 the domain are returned unchanged.

29381 **RETURN VALUE**

29382 On successful completion, *tolower()* returns the lower-case letter corresponding to the argument
 29383 passed; otherwise it returns the argument unchanged.

29384 **ERRORS**

29385 No errors are defined.

29386 **EXAMPLES**

29387 None.

29388 **APPLICATION USAGE**

29389 None.

29390 **FUTURE DIRECTIONS**

29391 None.

29392 **SEE ALSO**29393 *setlocale()*, <ctype.h>, the XBD specification, **Chapter 5, Locale**.29394 **CHANGE HISTORY**

29395 First released in Issue 1.

29396 Derived from Issue 1 of the SVID.

29397 **Issue 4**

29398 The following changes are incorporated in this issue:

- 29399 • Reference to “shift information” is replaced by “character type information”.
- 29400 • The RETURN VALUE section is added.

29401 **NAME**

29402 _toupper — transliterate lower-case characters to upper-case

29403 **SYNOPSIS**

29404 EX #include <ctype.h>

29405 int _toupper(int c);

29406

29407 **DESCRIPTION**29408 The *_toupper()* macro is equivalent to *toupper()* except that the argument *c* must be a lower-case letter.29410 **RETURN VALUE**29411 On successful completion, *_toupper()* returns the upper-case letter corresponding to the argument passed.29413 **ERRORS**

29414 No errors are defined.

29415 **EXAMPLES**

29416 None.

29417 **APPLICATION USAGE**

29418 None.

29419 **FUTURE DIRECTIONS**

29420 None.

29421 **SEE ALSO**29422 *islower()*, *toupper()*, <ctype.h>, the XBD specification, **Chapter 5, Locale**.29423 **CHANGE HISTORY**

29424 First released in Issue 1.

29425 Derived from Issue 1 of the SVID.

29426 **Issue 4**

29427 The following change is incorporated in this issue:

- 29428
- The RETURN VALUE section is expanded.

29429 **NAME**

29430 toupper — transliterate lower-case characters to upper-case

29431 **SYNOPSIS**

29432 #include <ctype.h>

29433 int toupper(int c);

29434 **DESCRIPTION**

29435 The *toupper()* function has as a domain a type **int**, the value of which is representable as an
29436 **unsigned char** or the value of EOF. If the argument has any other value, the behaviour is
29437 undefined. If the argument of *toupper()* represents a lower-case letter, and there exists a
29438 corresponding upper-case letter (as defined by character type information in the program locale
29439 category LC_CTYPE), the result is the corresponding upper-case letter. All other arguments in
29440 the domain are returned unchanged.

29441 **RETURN VALUE**

29442 On successful completion, *toupper()* returns the upper-case letter corresponding to the argument
29443 passed.

29444 **ERRORS**

29445 No errors are defined.

29446 **EXAMPLES**

29447 None.

29448 **APPLICATION USAGE**

29449 None.

29450 **FUTURE DIRECTIONS**

29451 None.

29452 **SEE ALSO**

29453 *setlocale()*, <ctype.h>, the XBD specification, **Chapter 5, Locale**.

29454 **CHANGE HISTORY**

29455 First released in Issue 1.

29456 Derived from Issue 1 of the SVID.

29457 **Issue 4**

29458 The following changes are incorporated in this issue:

- 29459 • Reference to “shift information” is replaced by “character type information”.
- 29460 • The RETURN VALUE section is added.

29461 **NAME**

29462 towctrans — character transliteration

29463 **SYNOPSIS**

29464 #include <wctype.h>

29465 wint_t towctrans(wint_t *wc*, wctrans_t *desc*);29466 **DESCRIPTION**

29467 The *towctrans()* function transliterates the wide-character code *wc* using the mapping described
 29468 by *desc*. The current setting of the LC_CTYPE category should be the same as during the call to
 29469 *wctrans()* that returned the value *desc*. If the value of *desc* is invalid (that is, not obtained by a
 29470 call to *wctrans()* or *desc* is invalidated by a subsequent call to *setlocale()* that has affected
 29471 category LC_CTYPE) the result is implementation-dependent.

29472 **RETURN VALUE**

29473 If successful, the *towctrans()* function returns the mapped value of *wc* using the mapping
 29474 described by *desc*. Otherwise it returns *wc* unchanged.

29475 **ERRORS**29476 The *towctrans()* function may fail if:29477 [EINVAL] *desc* contains an invalid transliteration descriptor.29478 **EXAMPLES**

29479 None.

29480 **APPLICATION USAGE**

29481 The strings — "tolower" and "toupper" — are reserved for the standard mapping names. In the
 29482 table below, the functions in the left column are equivalent to the functions in the right column.

29483 tolower(<i>wc</i>)	towctrans(<i>wc</i> , wctrans("tolower"))
29484 toupper(<i>wc</i>)	towctrans(<i>wc</i> , wctrans("toupper"))

29485 **FUTURE DIRECTIONS**

29486 None.

29487 **SEE ALSO**29488 *tolower()*, *toupper()*, *wctrans()*, <wctype.h>.29489 **CHANGE HISTORY**

29490 First released in Issue 5.

29491 Derived from ISO/IEC 9899:1990/Amendment 1:1994 (E).

29492 **NAME**

29493 tolower — transliterate upper-case wide-character code to lower-case

29494 **SYNOPSIS**

29495 #include <wctype.h>

29496 wint_t tolower(wint_t wc);

29497 **DESCRIPTION**

29498 The *tolower()* function has as a domain a type **wint_t**, the value of which must be a character
 29499 representable as a **wchar_t**, and must be a wide-character code corresponding to a valid
 29500 character in the current locale or the value of WEOF. If the argument has any other value, the
 29501 behaviour is undefined. If the argument of *tolower()* represents an upper-case wide-character
 29502 code, and there exists a corresponding lower-case wide-character code (as defined by character
 29503 type information in the program locale category LC_CTYPE), the result is the corresponding
 29504 lower-case wide-character code. All other arguments in the domain are returned unchanged.

29505 **RETURN VALUE**

29506 On successful completion, *tolower()* returns the lower-case letter corresponding to the
 29507 argument passed; otherwise it returns the argument unchanged.

29508 **ERRORS**

29509 No errors are defined.

29510 **EXAMPLES**

29511 None.

29512 **APPLICATION USAGE**

29513 None.

29514 **FUTURE DIRECTIONS**

29515 None.

29516 **SEE ALSO**29517 *setlocale()*, <wctype.h>, <wchar.h>, the XBD specification, **Chapter 5, Locale**.29518 **CHANGE HISTORY**

29519 First released in Issue 4.

29520 **Issue 5**

29521 The following change has been made in this issue for alignment with ISO/IEC
 29522 9899:1990/Amendment 1:1994 (E).

- 29523 • The SYNOPSIS has been changed to indicate that this function and associated data types are
 29524 now made visible by inclusion of the header <wctype.h> rather than <wchar.h>.

29525 **NAME**

29526 towupper — transliterate lower-case wide-character code to upper-case

29527 **SYNOPSIS**

29528 #include <wctype.h>

29529 wint_t towupper(wint_t wc);

29530 **DESCRIPTION**

29531 The *towupper()* function has as a domain a type **wint_t**, the value of which must be a character
 29532 representable as a **wchar_t**, and must be a wide-character code corresponding to a valid
 29533 character in the current locale or the value of WEOF. If the argument has any other value, the
 29534 behaviour is undefined. If the argument of *towupper()* represents a lower-case wide-character
 29535 code, and there exists a corresponding upper-case wide-character code (as defined by character
 29536 type information in the program locale category LC_CTYPE), the result is the corresponding
 29537 upper-case wide-character code. All other arguments in the domain are returned unchanged.

29538 **RETURN VALUE**

29539 Upon successful completion, *towupper()* returns the upper-case letter corresponding to the
 29540 argument passed. Otherwise it returns the argument unchanged.

29541 **ERRORS**

29542 No errors are defined.

29543 **EXAMPLES**

29544 None.

29545 **APPLICATION USAGE**

29546 None.

29547 **FUTURE DIRECTIONS**

29548 None.

29549 **SEE ALSO**

29550 *setlocale()*, <wctype.h>, <wchar.h>, the XBD specification, **Chapter 5, Locale**.

29551 **CHANGE HISTORY**

29552 First released in Issue 4.

29553 **Issue 5**

29554 The following change has been made in this issue for alignment with ISO/IEC
 29555 9899:1990/Amendment 1:1994 (E).

- 29556 • The SYNOPSIS has been changed to indicate that this function and associated data types are
 29557 now made visible by inclusion of the header <wctype.h> rather than <wchar.h>.

29558 **NAME**

29559 truncate — truncate a file to a specified length

29560 **SYNOPSIS**

29561 EX #include <unistd.h>

29562 int truncate(const char *path, off_t length);

29563

29564 **DESCRIPTION**29565 Refer to *ftruncate()*.29566 **CHANGE HISTORY**

29567 First released in Issue 4, Version 2.

29568 **Issue 5**

29569 Moved from X/OPEN UNIX extension to BASE.

29570 NAME

29571 tdelete, tfind, tsearch, twalk — manage a binary search tree

29572 SYNOPSIS

29573 EX

```
#include <search.h>
```

```
29574 void *tsearch(const void *key, void **rootp,
29575             int (*compar)(const void *, const void *));
29576 void *tfind(const void *key, void *const *rootp,
29577            int (*compar)(const void *, const void *));
29578 void *tdelete(const void *key, void **rootp,
29579              int (*compar)(const void *, const void *));
29580 void twalk(const void *root,
29581           void (*action)(const void *, VISIT, int));
29582
```

29583 DESCRIPTION

29584 The *tsearch()*, *tfind()*, *tdelete()* and *twalk()* functions manipulate binary search trees.
 29585 Comparisons are made with a user-supplied routine, the address of which is passed as the
 29586 *compar* argument. This routine is called with two arguments, the pointers to the elements being
 29587 compared. The user-supplied routine must return an integer less than, equal to or greater than 0,
 29588 according to whether the first argument is to be considered less than, equal to or greater than the
 29589 second argument. The comparison function need not compare every byte, so arbitrary data may
 29590 be contained in the elements in addition to the values being compared.

29591 The *tsearch()* function is used to build and access the tree. The *key* argument is a pointer to an
 29592 element to be accessed or stored. If there is a node in the tree whose element is equal to the
 29593 value pointed to by *key*, a pointer to this found node is returned. Otherwise, the value pointed to
 29594 by *key* is inserted (that is, a new node is created and the value of *key* is copied to this node), and a
 29595 pointer to this node returned. Only pointers are copied, so the calling routine must store the
 29596 data. The *rootp* argument points to a variable that points to the root node of the tree. A null
 29597 pointer value for the variable pointed to by *rootp* denotes an empty tree; in this case, the variable
 29598 will be set to point to the node which will be at the root of the new tree.

29599 Like *tsearch()*, *tfind()* will search for a node in the tree, returning a pointer to it if found.
 29600 However, if it is not found, *tfind()* will return a null pointer. The arguments for *tfind()* are the
 29601 same as for *tsearch()*.

29602 The *tdelete()* function deletes a node from a binary search tree. The arguments are the same as
 29603 for *tsearch()*. The variable pointed to by *rootp* will be changed if the deleted node was the root of
 29604 the tree. The *tdelete()* function returns a pointer to the parent of the deleted node, or a null
 29605 pointer if the node is not found.

29606 The *twalk()* function traverses a binary search tree. The *root* argument is a pointer to the root
 29607 node of the tree to be traversed. (Any node in a tree may be used as the root for a walk below
 29608 that node.) The argument *action* is the name of a routine to be invoked at each node. This
 29609 routine is, in turn, called with three arguments. The first argument is the address of the node
 29610 being visited. The structure pointed to by this argument is unspecified and must not be
 29611 modified by the application, but it is guaranteed that a pointer-to-node can be converted to
 29612 pointer-to-pointer-to-element to access the element stored in the node. The second argument is
 29613 a value from an enumeration data type:

```
29614     typedef enum { preorder, postorder, endorder, leaf } VISIT;
```

29615 (defined in *<search.h>*), depending on whether this is the first, second or third time that the
 29616 node is visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a
 29617 leaf. The third argument is the level of the node in the tree, with the root being level 0.

29618 If the calling function alters the pointer to the root, the result is undefined.

29619 RETURN VALUE

29620 If the node is found, both *tsearch()* and *tfind()* return a pointer to it. If not, *tfind()* returns a null
29621 pointer, and *tsearch()* returns a pointer to the inserted item.

29622 A null pointer is returned by *tsearch()* if there is not enough space available to create a new node.

29623 A null pointer is returned by *tsearch()*, *tfind()* and *tdelete()* if *rootp* is a null pointer on entry.

29624 The *tdelete()* function returns a pointer to the parent of the deleted node, or a null pointer if the
29625 node is not found.

29626 The *twalk()* function returns no value.

29627 ERRORS

29628 No errors are defined.

29629 EXAMPLES

29630 The following code reads in strings and stores structures containing a pointer to each string and
29631 a count of its length. It then walks the tree, printing out the stored strings and their lengths in
29632 alphabetical order.

```
29633     #include <search.h>
29634     #include <string.h>
29635     #include <stdio.h>
29636     #define STRSZ    10000
29637     #define NODSZ    500
29638     struct node {      /* pointers to these are stored in the tree */
29639         char    *string;
29640         int     length;
29641     };
29642     char    string_space[STRSZ]; /* space to store strings */
29643     struct node nodes[NODSZ];    /* nodes to store */
29644     void    *root = NULL;        /* this points to the root */
29645     int main(int argc, char *argv[])
29646     {
29647         char    *strptr = string_space;
29648         struct node *nodeptr = nodes;
29649         void    print_node(const void *, VISIT, int);
29650         int     i = 0, node_compare(const void *, const void *);
29651         while (gets(strptr) != NULL && i++ < NODSZ) {
29652             /* set node */
29653             nodeptr->string = strptr;
29654             nodeptr->length = strlen(strptr);
29655             /* put node into the tree */
29656             (void) tsearch((void *)nodeptr, (void **)&root,
29657                 node_compare);
29658             /* adjust pointers, so we do not overwrite tree */
29659             strptr += nodeptr->length + 1;
29660             nodeptr++;
29661         }
29662         twalk(root, print_node);
29663         return 0;
29664     }
29665     /*
```

```

29666      * This routine compares two nodes, based on an
29667      * alphabetical ordering of the string field.
29668      */
29669      int
29670      node_compare(const void *node1, const void *node2)
29671      {
29672          return strcmp(((const struct node *) node1)->string,
29673                      ((const struct node *) node2)->string);
29674      }
29675      /*
29676      * This routine prints out a node, the second time
29677      * twalk encounters it or if it is a leaf.
29678      */
29679      void
29680      print_node(const void *ptr, VISIT order, int level)
29681      {
29682          const struct node *p = *(const struct node **) ptr;
29683
29684          if (order == postorder || order == leaf) {
29685              (void) printf("string = %s, length = %d\n",
29686                          p->string, p->length);
29687          }
29688      }

```

29688 APPLICATION USAGE

29689 The *root* argument to *twalk()* is one level of indirection less than the *rootp* arguments to *tsearch()* and *tdelete()*.

29691 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**.

29696 FUTURE DIRECTIONS

29697 None.

29698 SEE ALSO

29699 *bsearch()*, *hsearch()*, *lsearch()*, <**search.h**>.

29700 CHANGE HISTORY

29701 First released in Issue 1.

29702 Derived from Issue 1 of the SVID.

29703 Issue 4

29704 The following changes are incorporated in this issue:

- 29705 • The type of argument *key* in the definition of *tsearch()* is changed from **void*** to **const void***.
29706 The definitions of other functions are changed as indicated on their respective entries.
- 29707 • Various minor wording changes are made in the DESCRIPTION to improve clarity and
29708 accuracy. In particular, additional notes are added about constraints on the first argument to
29709 *twalk()*.
- 29710 • The sample code in the EXAMPLES section is updated to use ISO C syntax. Also the
29711 definition of the *root* and *argv* items is changed.

29712	• The paragraph in the APPLICATION USAGE section about casts is removed.	
29713	Issue 5	
29714	The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in	
29715	previous issues.	

NAME

ttyname, ttyname_r — find pathname 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 returns a pointer to a string containing a null-terminated pathname 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()* interface need not be reentrant.

The *ttyname_r()* function stores the null-terminated pathname 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()* returns a pointer to a string. Otherwise, a null pointer is returned and *errno* is set to indicate the error.

If successful, the *ttyname_r()* function returns zero. Otherwise, an error number is 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 device.

The *ttyname_r()* function may fail if:

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

[ENOTTY] The *fildes* argument does not refer to a tty.

[ERANGE] The value of *namesize* is smaller than the length of the string to be returned including the terminating null character.

EXAMPLES

None.

APPLICATION USAGE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<unistd.h>.

CHANGE HISTORY

First released in Issue 1.

Derived from Issue 1 of the SVID.

29756 **Issue 4**

29757 The following changes are incorporated in this issue:

- 29758 • The **<unistd.h>** header is added to the SYNOPSIS. |
- 29759 • The statement indicating that *errno* will be set on error in the RETURN VALUE section, and |
- 29760 the errors [EBADF] and [ENOTTY], are marked as extensions. |

29761 **Issue 5**

29762 The *ttyname_r()* function is included for alignment with the POSIX Threads Extension. |

29763 A note indicating that the *ttyname()* interface need not be reentrant is added to the |

29764 DESCRIPTION. |

29765 NAME

29766 ttyslot — find the slot of the current user in the user accounting database (**LEGACY**)

29767 SYNOPSIS

29768 EX #include <stdlib.h>

29769 int ttyslot(void);

29770

29771 DESCRIPTION

29772 The *ttyslot()* function returns the index of the current user's entry in the user accounting database. The current user's entry is an entry for which the **utline** member matches the name of a terminal device associated with any of the process' file descriptors 0, 1 or 2. The index is an ordinal number representing the record number in the database of the current user's entry. The first entry in the database is represented by the return value 0.

29777 This interface need not be reentrant.

29778 RETURN VALUE

29779 Upon successful completion, *ttyslot()* returns the index of the current user's entry in the user accounting database. The *ttyslot()* function returns -1 if an error was encountered while searching the database or if none of file descriptors 0, 1 or 2 is associated with a terminal device.

29782 ERRORS

29783 No errors are defined.

29784 EXAMPLES

29785 None.

29786 APPLICATION USAGE

29787 None.

29788 FUTURE DIRECTIONS

29789 None.

29790 SEE ALSO

29791 *endutxent()*, *ttyname()*, <stdlib.h>.

29792 CHANGE HISTORY

29793 First released in Issue 4, Version 2.

29794 Issue 5

29795 Marked LEGACY.

29796 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

29797 **NAME**

29798 twalk — traverse a binary search tree

29799 **SYNOPSIS**29800 EX `#include <search.h>`

```
29801 void twalk(const void *root,  
29802           void (*action)(const void *, VISIT, int ));
```

29803

29804 **DESCRIPTION**29805 Refer to *tsearch()*.29806 **CHANGE HISTORY**

29807 First released in Issue 3.

29808 Derived from Issue 1 of the SVID.

29809 **Issue 4**

29810 The following changes are incorporated in this issue:

- 29811 • The type of argument *root* is changed from **char *** to **const void***, and the argument list to
- 29812 *action()* is formally defined.

29813 **NAME**

29814 tzname — timezone strings

29815 **SYNOPSIS**

29816 #include <time.h>

29817 extern char *tzname[];

29818 **DESCRIPTION**29819 Refer to *tzset()*.29820 **CHANGE HISTORY**

29821 First released in Issue 1.

29822 Derived from Issue 1 of the SVID.

29823 **Issue 4**

29824 The following change is incorporated in this issue:

- 29825
- The <**time.h**> header is added to the SYNOPSIS section.

29826 **NAME**

29827 tzset — set time zone conversion information

29828 **SYNOPSIS**

29829 #include <time.h>

29830 void tzset (void);

29831 extern char *tzname[];

29832 EX extern long int timezone;

29833 extern int daylight;

29834

29835 **DESCRIPTION**

29836 The *tzset()* function uses the value of the environment variable *TZ* to set time conversion information used by *localtime()*, *ctime()*, *strftime()* and *mktime()*. If *TZ* is absent from the environment, implementation-dependent default time zone information is used.

29839 The *tzset()* function sets the external variable *tzname* as follows:

29840 tzname[0] = "std";

29841 tzname[1] = "dst";

29842 where *std* and *dst* are as described in the XBD specification, **Chapter 6, Environment Variables**.

29843 EX The *tzset()* function also sets the external variable *daylight* to 0 if Daylight Savings Time conversions should never be applied for the time zone in use; otherwise non-zero. The external variable *timezone* is set to the difference, in seconds, between Coordinated Universal Time (UTC) and local standard time, for example:

29847

29848

29849

29850

29851

29852

29853

29854

	TZ	<i>timezone</i>
	EST	5*60*60
	GMT	0*60*60
	JST	-9*60*60
	MET	-1*60*60
	MST	7*60*60
	PST	8*60*60

29855

29856 **RETURN VALUE**29857 The *tzset()* function returns no value.29858 **ERRORS**

29859 No errors are defined.

29860 **EXAMPLES**

29861 None.

29862 **APPLICATION USAGE**

29863 None.

29864 **FUTURE DIRECTIONS**

29865 None.

29866 **SEE ALSO**29867 *ctime()*, *localtime()*, *mktime()*, *strftime()*, <time.h>.

29868 **CHANGE HISTORY**

29869 First released in Issue 1.

29870 Derived from Issue 1 of the SVID.

29871 **Issue 4**

29872 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 29873
- The argument list is explicitly defined as **void**.

29874 Another change is incorporated as follows:

- 29875
- The reference to *timezone* in the SYNOPSIS section is marked as an extension.

29876 **NAME**29877 *ualarm* — set the interval timer29878 **SYNOPSIS**29879 EX `#include <unistd.h>`29880 `useconds_t ualarm(useconds_t useconds, useconds_t interval);`

29881

29882 **DESCRIPTION**

29883 The *ualarm()* function causes the SIGALRM signal to be generated for the calling process after
 29884 the number of real-time microseconds specified by the *useconds* argument has elapsed. When
 29885 the *interval* argument is non-zero, repeated timeout notification occurs with a period in
 29886 microseconds specified by the *interval* argument. If the notification signal, SIGALRM, is not
 29887 caught or ignored, the calling process is terminated.

29888 Implementations may place limitations on the granularity of timer values. For each interval
 29889 timer, if the requested timer value requires a finer granularity than the implementation supports,
 29890 the actual timer value will be rounded up to the next supported value.

29891 Interactions between *ualarm()* and any of the following are unspecified:

29892 *alarm()*29893 RT *nanosleep()*29894 *setitimer()*29895 RT *timer_create()*29896 *timer_delete()*29897 *timer_getoverrun()*29898 *timer_gettime()*29899 *timer_settime()*29900 *sleep()*29901 **RETURN VALUE**

29902 The *ualarm()* function returns the number of microseconds remaining from the previous
 29903 *ualarm()* call. If no timeouts are pending or if *ualarm()* has not previously been called, *ualarm()*
 29904 returns 0.

29905 **ERRORS**

29906 No errors are defined.

29907 **EXAMPLES**

29908 None.

29909 **APPLICATION USAGE**

29910 The *ualarm()* function is a simplified interface to *setitimer()*, and uses the ITIMER_REAL interval
 29911 timer.

29912 **FUTURE DIRECTIONS**

29913 None.

29914 **SEE ALSO**

29915 *alarm()*, *nanosleep()*, *setitimer()*, *sleep()*, *timer_create()*, *timer_delete()*, *timer_getoverrun()*,
 29916 *timer_gettime()*, *timer_settime()* <unistd.h>.

29917 **CHANGE HISTORY**

29918 First released in Issue 4, Version 2.

29919 **Issue 5**

29920 Moved from X/OPEN UNIX extension to BASE.

|

29921 **NAME**

29922 ulimit — get and set process limits

29923 **SYNOPSIS**

29924 EX #include <ulimit.h>

29925 long int ulimit(int *cmd*, ...);

29926

29927 **DESCRIPTION**29928 The *ulimit()* function provides for control over process limits. The *cmd* values, defined in
29929 <ulimit.h> include:29930 UL_GETFSIZE Return the soft file size limit of the process. The limit is in units of 512-byte
29931 blocks and is inherited by child processes. Files of any size can be read. The
29932 return value is the integer part of the soft file size limit divided by 512. If the
29933 result cannot be represented as a **long int**, the result is unspecified.29934 UL_SETFSIZE Set the hard and soft file size limits for output operations of the process to the
29935 value of the second argument, taken as a **long int**. Any process may decrease
29936 its own hard limit, but only a process with appropriate privileges may
29937 increase the limit. The new file size limit is returned. The hard and soft file
29938 size limits are set to the specified value multiplied by 512. If the result would
29939 overflow an **rlim_t**, the actual value set is unspecified.29940 The *ulimit()* function will not change the setting of **errno** if successful.29941 **RETURN VALUE**29942 Upon successful completion, *ulimit()* returns the value of the requested limit. Otherwise -1 is
29943 returned and *errno* is set to indicate the error.29944 **ERRORS**29945 The *ulimit()* function will fail and the limit will be unchanged if:29946 [EINVAL] The *cmd* argument is not valid.29947 [EPERM] A process not having appropriate privileges attempts to increase its file size
29948 limit.29949 **EXAMPLES**

29950 None.

29951 **APPLICATION USAGE**29952 As all return values are permissible in a successful situation, an application wishing to check for
29953 error situations should set *errno* to 0, then call *ulimit()*, and, if it returns -1, check to see if *errno* is
29954 non-zero.29955 **FUTURE DIRECTIONS**

29956 None.

29957 **SEE ALSO**29958 *getrlimit()*, *setrlimit()*, *write()*, <ulimit.h>.29959 **CHANGE HISTORY**

29960 First released in Issue 1.

29961 Derived from Issue 1 of the SVID.

29962 **Issue 4**

29963 The following change is incorporated in this issue:

29964	<ul style="list-style-type: none">• The use of long is replaced by long int in the SYNOPSIS and the DESCRIPTION sections.	
29965	Issue 4, Version 2	
29966	In the DESCRIPTION, the discussion of UL_GETFSIZE and UL_SETFSIZE is revised generally to	
29967	distinguish between the soft and the hard file size limit of the process. For UL_GETFSIZE, the	
29968	return value is defined more precisely. For UL_SETFSIZE, the effect on both file size limits is	
29969	specified, as is the effect if the result would overflow an rlim_t .	
29970	Issue 5	
29971	In the description of UL_SETFSIZE, the text is corrected to refer to rlim_t rather than the	
29972	spurious rlimit_t .	
29973	The DESCRIPTION is updated to indicate that errno will not be changed if the function is	
29974	successful.	

29975 **NAME**

29976 umask — set and get file mode creation mask

29977 **SYNOPSIS**

29978 OH #include <sys/types.h>

29979 #include <sys/stat.h>

29980 mode_t umask(mode_t *cmask*);29981 **DESCRIPTION**

29982 The *umask()* function sets the process' file mode creation mask to *cmask* and returns the previous
 29983 value of the mask. Only the file permission bits of *cmask* (see <sys/stat.h>) are used; the
 29984 meaning of the other bits is implementation-dependent.

29985 The process' file mode creation mask is used during *open()*, *creat()*, *mkdir()* and *mkfifo()* to turn
 29986 off permission bits in the *mode* argument supplied. Bit positions that are set in *cmask* are cleared
 29987 in the mode of the created file.

29988 **RETURN VALUE**

29989 The file permission bits in the value returned by *umask()* will be the previous value of the file
 29990 mode creation mask. The state of any other bits in that value is unspecified, except that a
 29991 subsequent call to *umask()* with the returned value as *cmask* will leave the state of the mask the
 29992 same as its state before the first call, including any unspecified use of those bits.

29993 **ERRORS**

29994 No errors are defined.

29995 **EXAMPLES**

29996 None.

29997 **APPLICATION USAGE**

29998 None.

29999 **FUTURE DIRECTIONS**

30000 None.

30001 **SEE ALSO**30002 *creat()*, *mkdir()*, *mkfifo()*, *open()*, <sys/stat.h>, <sys/types.h>.30003 **CHANGE HISTORY**

30004 First released in Issue 1.

30005 Derived from Issue 1 of the SVID.

30006 **Issue 4**

30007 The following changes are incorporated in this issue:

- 30008 • The <sys/types.h> header is now marked as optional (OH); this header need not be included
- 30009 on XSI-conformant systems.
- 30010 • The RETURN VALUE section is expanded, in line with the ISO POSIX-1 standard, to describe
- 30011 the situation with regard to additional bits in the file mode creation mask.

30012 **NAME**

30013 uname — get name of current system

30014 **SYNOPSIS**

30015 #include <sys/utsname.h>

30016 int uname(struct utsname *name);

30017 **DESCRIPTION**30018 The *uname()* function stores information identifying the current system in the structure pointed
30019 to by *name*.30020 The *uname()* function uses the *utsname* structure defined in <sys/utsname.h>.30021 The *uname()* function returns a string naming the current system in the character array *sysname*.
30022 Similarly, *nodename* contains the name that the system is known by on a communications
30023 network. The arrays *release* and *version* further identify the operating system. The array *machine*
30024 contains a name that identifies the hardware that the system is running on.

30025 The format of each member is implementation-dependent.

30026 **RETURN VALUE**30027 Upon successful completion, a non-negative value is returned. Otherwise, -1 is returned and
30028 *errno* is set to indicate the error.30029 **ERRORS**

30030 No errors are defined.

30031 **EXAMPLES**

30032 None.

30033 **APPLICATION USAGE**30034 The inclusion of the *nodename* member in this structure does not imply that it is sufficient
30035 information for interfacing to communications networks.30036 **FUTURE DIRECTIONS**

30037 None.

30038 **SEE ALSO**

30039 <sys/utsname.h>.

30040 **CHANGE HISTORY**

30041 First released in Issue 1.

30042 Derived from Issue 1 of the SVID.

30043 **Issue 4**

30044 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- 30045 • The DESCRIPTION is changed to indicate that the format of members in the **utsname**
- 30046 structure is implementation-dependent.
- 30047 • The RETURN VALUE section is updated to indicate that -1 will be returned and *errno* set to
- 30048 indicate an error.

30049 **NAME**

30050 ungetc — push byte back into input stream

30051 **SYNOPSIS**

30052 #include <stdio.h>

30053 int ungetc(int *c*, FILE **stream*);30054 **DESCRIPTION**

30055 The *ungetc()* function pushes the byte specified by *c* (converted to an **unsigned char**) back onto
 30056 the input stream pointed to by *stream*. The pushed-back bytes will be returned by subsequent
 30057 reads on that stream in the reverse order of their pushing. A successful intervening call (with
 30058 the stream pointed to by *stream*) to a file-positioning function (*fseek()*, *fsetpos()* or *rewind()*)
 30059 discards any pushed-back bytes for the stream. The external storage corresponding to the
 30060 stream is unchanged.

30061 One byte of push-back is guaranteed. If *ungetc()* is called too many times on the same stream
 30062 without an intervening read or file-positioning operation on that stream, the operation may fail.

30063 If the value of *c* equals that of the macro EOF, the operation fails and the input stream is
 30064 unchanged.

30065 A successful call to *ungetc()* clears the end-of-file indicator for the stream. The value of the file-
 30066 position indicator for the stream after reading or discarding all pushed-back bytes will be the
 30067 same as it was before the bytes were pushed back. The file-position indicator is decremented by
 30068 each successful call to *ungetc()*; if its value was 0 before a call, its value is indeterminate after the
 30069 call.

30070 **RETURN VALUE**

30071 Upon successful completion, *ungetc()* returns the byte pushed back after conversion. Otherwise
 30072 it returns EOF.

30073 **ERRORS**

30074 No errors are defined.

30075 **EXAMPLES**

30076 None.

30077 **APPLICATION USAGE**

30078 None.

30079 **FUTURE DIRECTIONS**

30080 None.

30081 **SEE ALSO**30082 *fseek()*, *getc()*, *fsetpos()*, *read()*, *rewind()*, *setbuf()*, <stdio.h>.30083 **CHANGE HISTORY**

30084 First released in Issue 1.

30085 Derived from Issue 1 of the SVID.

30086 **Issue 4**

30087 The following changes are incorporated for alignment with the ISO C standard:

- 30088 • The *fsetpos()* function is added to the list of file-positioning functions in the DESCRIPTION.
- 30089 • Also this issue states that the file-position indicator is decremented by each successful call to
 30090 *ungetc()*, although note that XSI-conformant systems do not distinguish between text and
 30091 binary streams. Previous issues state that the disposition of this indicator is unspecified.

30092 Other changes are incorporated as follows:

- 30093 • The DESCRIPTION is changed to make it clear that *ungetc()* manipulates bytes rather than |
- 30094 (possibly multi-byte) characters.
- 30095 • The APPLICATION USAGE section is removed. |

30096 **NAME**

30097 ungetwc — push wide-character code back into input stream

30098 **SYNOPSIS**

30099 #include <stdio.h>

30100 #include <wchar.h>

30101 wint_t ungetwc(wint_t wc, FILE *stream);

30102 **DESCRIPTION**

30103 The *ungetwc()* function pushes the character corresponding to the wide-character code specified
 30104 by *wc* back onto the input stream pointed to by *stream*. The pushed-back characters will be
 30105 returned by subsequent reads on that stream in the reverse order of their pushing. A successful
 30106 intervening call (with the stream pointed to by *stream*) to a file-positioning function (*fseek()*,
 30107 *fsetpos()* or *rewind()*) discards any pushed-back characters for the stream. The external storage
 30108 corresponding to the stream is unchanged.

30109 One character of push-back is guaranteed. If *ungetwc()* is called too many times on the same
 30110 stream without an intervening read or file-positioning operation on that stream, the operation
 30111 may fail.

30112 If the value of *wc* equals that of the macro WEOF, the operation fails and the input stream is
 30113 unchanged.

30114 A successful call to *ungetwc()* clears the end-of-file indicator for the stream. The value of the
 30115 file-position indicator for the stream after reading or discarding all pushed-back characters will
 30116 be the same as it was before the characters were pushed back. The file-position indicator is
 30117 decremented (by one or more) by each successful call to *ungetwc()*; if its value was 0 before a
 30118 call, its value is indeterminate after the call.

30119 **RETURN VALUE**

30120 Upon successful completion, *ungetwc()* returns the wide-character code corresponding to the
 30121 pushed-back character. Otherwise it returns WEOF.

30122 **ERRORS**30123 The *ungetwc()* function may fail if:

30124	[EILSEQ]	An invalid character sequence is detected, or a wide-character code does not
30125		correspond to a valid character.

30126 **EXAMPLES**

30127 None.

30128 **APPLICATION USAGE**

30129 None.

30130 **FUTURE DIRECTIONS**

30131 None.

30132 **SEE ALSO**30133 *fseek()*, *fsetpos()*, *read()*, *rewind()*, *setbuf()*, <stdio.h>, <wchar.h>.30134 **CHANGE HISTORY**

30135 First released in Issue 4.

30136 Derived from the MSE working draft.

30137 **Issue 5**

30138 The Optional Header (OH) marking is removed from <stdio.h>.

30139 NAME

30140 unlink — remove a directory entry

30141 SYNOPSIS

30142 #include <unistd.h>

30143 int unlink(const char *path);

30144 DESCRIPTION

30145 EX The *unlink()* function removes a link to a file. If *path* names a symbolic link, *unlink()* removes the
30146 symbolic link named by *path* and does not affect any file or directory named by the contents of
30147 the symbolic link. Otherwise, *unlink()* removes the link named by the pathname pointed to by
30148 *path* and decrements the link count of the file referenced by the link.

30149 When the file's link count becomes 0 and no process has the file open, the space occupied by the
30150 file will be freed and the file will no longer be accessible. If one or more processes have the file
30151 open when the last link is removed, the link will be removed before *unlink()* returns, but the
30152 removal of the file contents will be postponed until all references to the file are closed.

30153 The *path* argument must not name a directory unless the process has appropriate privileges and
30154 the implementation supports using *unlink()* on directories.

30155 Upon successful completion, *unlink()* will mark for update the *st_ctime* and *st_mtime* fields of the
30156 parent directory. Also, if the file's link count is not 0, the *st_ctime* field of the file will be marked
30157 for update.

30158 RETURN VALUE

30159 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate
30160 the error. If -1 is returned, the named file will not be changed.

30161 ERRORS

30162 The *unlink()* function will fail and not unlink the file if:

30163 [EACCES] Search permission is denied for a component of the path prefix, or write
30164 permission is denied on the directory containing the directory entry to be
30165 removed.

30166 [EBUSY] The file named by the *path* argument cannot be unlinked because it is being
30167 used by the system or another process and the implementation considers this
30168 an error.

30169 EX [ELOOP] Too many symbolic links were encountered in resolving *path*.

30170 FIPS [ENAMETOOLONG]
30171 The length of the *path* argument exceeds {PATH_MAX} or a pathname
30172 component is longer than {NAME_MAX}.

30173 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

30174 [ENOTDIR] A component of the path prefix is not a directory.

30175 [EPERM] The file named by *path* is a directory, and either the calling process does not
30176 have appropriate privileges, or the implementation prohibits using *unlink()*
30177 on directories.

30178 EX [EPERM] or [EACCES]
30179 The S_ISVTX flag is set on the directory containing the file referred to by the
30180 *path* argument and the caller is not the file owner, nor is the caller the
30181 directory owner, nor does the caller have appropriate privileges.

30182	[EROFS]	The directory entry to be unlinked is part of a read-only file system.	
30183		The <i>unlink()</i> function may fail and not unlink the file if:	
30184	EX	[EBUSY] The file named by <i>path</i> is a named STREAM.	
30185		[ENAMETOOLONG]	
30186		Pathname resolution of a symbolic link produced an intermediate result	
30187		whose length exceeds {PATH_MAX}.	
30188		[ETXTBSY] The entry to be unlinked is the last directory entry to a pure procedure (shared	
30189		text) file that is being executed.	
30190	EXAMPLES		
30191		None.	
30192	APPLICATION USAGE		
30193		Applications should use <i>rmdir()</i> to remove a directory.	
30194	FUTURE DIRECTIONS		
30195		None.	
30196	SEE ALSO		
30197		<i>close()</i> , <i>link()</i> , <i>remove()</i> , <i>rmdir()</i> , < unistd.h >.	
30198	CHANGE HISTORY		
30199		First released in Issue 1.	
30200		Derived from Issue 1 of the SVID.	
30201	Issue 4		
30202		The following change is incorporated for alignment with the ISO POSIX-1 standard:	
30203		<ul style="list-style-type: none"> The type of argument <i>path</i> is changed from char * to const char *. 	
30204		The following change is incorporated for alignment with the FIPS requirements:	
30205		<ul style="list-style-type: none"> In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a 	
30206		pathname component is larger than {NAME_MAX} is now defined as mandatory and marked	
30207		as an extension.	
30208		Other changes are incorporated as follows:	
30209		<ul style="list-style-type: none"> The <unistd.h> header is added to the SYNOPSIS section. 	
30210		<ul style="list-style-type: none"> The error [ETXTBSY] is marked as an extension. 	
30211	Issue 4, Version 2		
30212		The entry is updated for X/OPEN UNIX conformance as follows:	
30213		<ul style="list-style-type: none"> In the DESCRIPTION, the effect is specified if <i>path</i> specifies a symbolic link. 	
30214		<ul style="list-style-type: none"> In the ERRORS section, [ELOOP] is added to indicate that too many symbolic links were 	
30215		encountered during pathname resolution	
30216		<ul style="list-style-type: none"> In the ERRORS section, [EPERM] or [EACCES] are added to indicate a permission check 	
30217		failure when operating on directories with S_ISVTX set.	
30218		<ul style="list-style-type: none"> In the ERRORS section, a second [ENAMETOOLONG] condition is defined that may report 	
30219		excessive length of an intermediate result of pathname resolution of a symbolic link.	
30220	Issue 5		
30221		The [EBUSY] error is added to the “may fail” part of the ERRORS section.	

30222 **NAME**

30223 unlockpt — unlock a pseudo-terminal master/slave pair

30224 **SYNOPSIS**30225 EX `#include <stdlib.h>`30226 `int unlockpt(int fildev);`

30227

30228 **DESCRIPTION**30229 The *unlockpt()* function unlocks the slave pseudo-terminal device associated with the master to
30230 which *fildev* refers.30231 Portable applications must call *unlockpt()* before opening the slave side of a pseudo-terminal
30232 device.30233 **RETURN VALUE**30234 Upon successful completion, *unlockpt()* returns 0. Otherwise, it returns -1 and sets *errno* to
30235 indicate the error.30236 **ERRORS**30237 The *unlockpt()* function may fail if:30238 [EBADF] The *fildev* argument is not a file descriptor open for writing.30239 [EINVAL] The *fildev* argument is not associated with a master pseudo-terminal device.30240 **EXAMPLES**

30241 None.

30242 **APPLICATION USAGE**

30243 None.

30244 **FUTURE DIRECTIONS**

30245 None.

30246 **SEE ALSO**30247 *grantpt()*, *open()*, *ptsname()*, *<stdlib.h>*.30248 **CHANGE HISTORY**

30249 First released in Issue 4, Version 2.

30250 **Issue 5**

30251 Moved from X/OPEN UNIX extension to BASE.

30252 **NAME**

30253 usleep — suspend execution for an interval

30254 **SYNOPSIS**

30255 EX #include <unistd.h>

30256 int usleep(useconds_t useconds);

30257

30258 **DESCRIPTION**

30259 The *usleep()* function will cause the calling thread to be suspended from execution until either
 30260 the number of real-time microseconds specified by the argument *useconds* has elapsed or a signal
 30261 is delivered to the calling thread and its action is to invoke a signal-catching function or to
 30262 terminate the process. The suspension time may be longer than requested due to the scheduling
 30263 of other activity by the system.

30264 The *useconds* argument must be less than 1,000,000. If the value of *useconds* is 0, then the call has
 30265 no effect.

30266 If a SIGALRM signal is generated for the calling process during execution of *usleep()* and if the
 30267 SIGALRM signal is being ignored or blocked from delivery, it is unspecified whether *usleep()*
 30268 returns when the SIGALRM signal is scheduled. If the signal is being blocked, it is also
 30269 unspecified whether it remains pending after *usleep()* returns or it is discarded.

30270 If a SIGALRM signal is generated for the calling process during execution of *usleep()*, except as a
 30271 result of a prior call to *alarm()*, and if the SIGALRM signal is not being ignored or blocked from
 30272 delivery, it is unspecified whether that signal has any effect other than causing *usleep()* to return.

30273 If a signal-catching function interrupts *usleep()* and examines or changes either the time a
 30274 SIGALRM is scheduled to be generated, the action associated with the SIGALRM signal, or
 30275 whether the SIGALRM signal is blocked from delivery, the results are unspecified.

30276 If a signal-catching function interrupts *usleep()* and calls *siglongjmp()* or *longjmp()* to restore an
 30277 environment saved prior to the *usleep()* call, the action associated with the SIGALRM signal and
 30278 the time at which a SIGALRM signal is scheduled to be generated are unspecified. It is also
 30279 unspecified whether the SIGALRM signal is blocked, unless the process' signal mask is restored
 30280 as part of the environment.

30281 Implementations may place limitations on the granularity of timer values. For each interval
 30282 timer, if the requested timer value requires a finer granularity than the implementation supports,
 30283 the actual timer value will be rounded up to the next supported value.

30284 Interactions between *usleep()* and any of the following are unspecified:

30285 RT *nanosleep()*30286 *setitimer()*30287 RT *timer_create()*30288 *timer_delete()*30289 *timer_getoverrun()*30290 *timer_gettime()*30291 *timer_settime()*30292 *ualarm()*30293 *sleep()*30294 **RETURN VALUE**

30295 On successful completion, *usleep()* returns 0. Otherwise, it returns -1 and sets *errno* to indicate
 30296 the error.

30297 **ERRORS**

30298 The *usleep()* function may fail if:

30299 [EINVAL] The time interval specified 1,000,000 or more microseconds.

30300 **EXAMPLES**

30301 None.

30302 **APPLICATION USAGE**

30303 Applications are recommended to use *setitimer()*, *timer_create()*, *timer_delete()*,
30304 *timer_getoverrun()*, *timer_gettime()* or *timer_settime()* instead of this interface.

30305 **FUTURE DIRECTIONS**

30306 None.

30307 **SEE ALSO**

30308 *alarm()*, *getitimer()*, *nanosleep()*, *sigaction()*, *sleep()*, *timer_create()*, *timer_delete()*,
30309 *timer_getoverrun()*, *timer_gettime()*, *timer_settime()*, <unistd.h>.

30310 **CHANGE HISTORY**

30311 First released in Issue 4, Version 2.

30312 **Issue 5**

30313 Moved from X/OPEN UNIX extension to BASE.

30314 The DESCRIPTION is changed to indicate that timers are now thread-based rather than
30315 process-based.

30316 **NAME**

30317 utime — set file access and modification times

30318 **SYNOPSIS**

30319 OH #include <sys/types.h>

30320 #include <utime.h>

30321 int utime(const char *path, const struct utimbuf *times);

30322 **DESCRIPTION**30323 The *utime()* function sets the access and modification times of the file named by the *path*
30324 argument.30325 If *times* is a null pointer, the access and modification times of the file are set to the current time.
30326 The effective user ID of the process must match the owner of the file, or the process must have
30327 write permission to the file or have appropriate privileges, to use *utime()* in this manner.30328 If *times* is not a null pointer, *times* is interpreted as a pointer to a **utimbuf** structure and the
30329 access and modification times are set to the values contained in the designated structure. Only a
30330 process with effective user ID equal to the user ID of the file or a process with appropriate
30331 privileges may use *utime()* this way.30332 The **utimbuf** structure is defined by the header <utime.h>. The times in the structure **utimbuf**
30333 are measured in seconds since the Epoch.30334 Upon successful completion, *utime()* will mark the time of the last file status change, **st_ctime**, to
30335 be updated, see <sys/stat.h>.30336 **RETURN VALUE**30337 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate
30338 the error, and the file times will not be affected.30339 **ERRORS**30340 The *utime()* function will fail if:30341 [EACCES] Search permission is denied by a component of the path prefix; or the *times*
30342 argument is a null pointer and the effective user ID of the process does not
30343 match the owner of the file and write access is denied.30344 EX [ELOOP] Too many symbolic links were encountered in resolving *path*.

30345 FIPS [ENAMETOOLONG]

30346 The length of the *path* argument exceeds {PATH_MAX} or a pathname
30347 component is longer than {NAME_MAX}.30348 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

30349 [ENOTDIR] A component of the path prefix is not a directory.

30350 [EPERM] The *times* argument is not a null pointer and the calling process' effective user
30351 ID has write access to the file but does not match the owner of the file and the
30352 calling process does not have the appropriate privileges.

30353 [EROFS] The file system containing the file is read-only.

30354 The *utime()* function may fail if:

30355 EX [ENAMETOOLONG]

30356 Pathname resolution of a symbolic link produced an intermediate result
30357 whose length exceeds {PATH_MAX}.

30358 **EXAMPLES**

30359 None.

30360 **APPLICATION USAGE**

30361 None.

30362 **FUTURE DIRECTIONS**

30363 None.

30364 **SEE ALSO**

30365 <sys/types.h>, <utime.h>.

30366 **CHANGE HISTORY**

30367 First released in Issue 1.

30368 Derived from Issue 1 of the SVID.

30369 **Issue 4**

30370 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 30371 • The type of argument *path* is changed from **char *** to **const char ***, and *times* is changed from
- 30372 **struct utimbuf*** to **const struct utimbuf***.

30373 The following change is incorporated for alignment with the FIPS requirements:

- 30374 • In the ERRORS section, the condition whereby [ENAMETOOLONG] will be returned if a
- 30375 pathname component is larger than {NAME_MAX} is now defined as mandatory and marked
- 30376 as an extension.

30377 Another change is incorporated as follows:

- 30378 • The <sys/types.h> header is now marked as optional (OH); this header need not be included
- 30379 on XSI-conformant systems.

30380 **Issue 4, Version 2**

30381 The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- 30382 • It states that [ELOOP] will be returned if too many symbolic links are encountered during
- 30383 pathname resolution.
- 30384 • A second [ENAMETOOLONG] condition is defined that may report excessive length of an
- 30385 intermediate result of pathname resolution of a symbolic link.

30386 **NAME**

30387 utimes — set file access and modification times

30388 **SYNOPSIS**30389 EX `#include <sys/time.h>`30390 `int utimes(const char *path, const struct timeval times[2]);`

30391

30392 **DESCRIPTION**

30393 The *utimes()* function sets the access and modification times of the file pointed to by the *path*
 30394 argument to the value of the *times* argument. The *utimes()* function allows time specifications
 30395 accurate to the microsecond.

30396 For *utimes()*, the *times* argument is an array of **timeval** structures. The first array member
 30397 represents the date and time of last access, and the second member represents the date and time
 30398 of last modification. The times in the **timeval** structure are measured in seconds and
 30399 microseconds since the Epoch, although rounding toward the nearest second may occur.

30400 If the *times* argument is a null pointer, the access and modification times of the file are set to the
 30401 current time. The effective user ID of the process must be the same as the owner of the file, or
 30402 must have write access to the file or appropriate privileges to use this call in this manner. Upon
 30403 completion, *utimes()* will mark the time of the last file status change, *st_ctime*, for update.

30404 **RETURN VALUE**

30405 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate
 30406 the error, and the file times will not be affected.

30407 **ERRORS**30408 The *utimes()* function will fail if:

30409 [EACCES] Search permission is denied by a component of the path prefix; or the *times*
 30410 argument is a null pointer and the effective user ID of the process does not
 30411 match the owner of the file and write access is denied.

30412 [ELOOP] Too many symbolic links were encountered in resolving *path*.

30413 [ENAMETOOLONG]

30414 The length of the *path* argument exceeds {PATH_MAX} or a pathname
 30415 component is longer than {NAME_MAX}.

30416 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

30417 [ENOTDIR] A component of the path prefix is not a directory.

30418 [EPERM] The *times* argument is not a null pointer and the calling process' effective user
 30419 ID has write access to the file but does not match the owner of the file and the
 30420 calling process does not have the appropriate privileges.

30421 [EROFS] The file system containing the file is read-only.

30422 The *utimes()* function may fail if:

30423 [ENAMETOOLONG]

30424 Pathname resolution of a symbolic link produced an intermediate result
 30425 whose length exceeds {PATH_MAX}.

30426 **EXAMPLES**

30427 None.

30428 **APPLICATION USAGE**

30429 None.

30430 **FUTURE DIRECTIONS**

30431 None.

30432 **SEE ALSO**

30433 <sys/time.h>.

30434 **CHANGE HISTORY**

30435 First released in Issue 4, Version 2.

30436 **Issue 5**

30437 Moved from X/OPEN UNIX extension to BASE.

30438 **NAME**30439 valloc — page-aligned memory allocator (**LEGACY**)30440 **SYNOPSIS**

30441 EX #include <stdlib.h>

30442 void *valloc(size_t size);

30443

30444 **DESCRIPTION**30445 The *valloc()* function has the same effect as *malloc()*, except that the allocated memory will be
30446 aligned to a multiple of the value returned by *sysconf(_SC_PAGESIZE)*.

30447 This interface need not be reentrant.

30448 **RETURN VALUE**30449 Upon successful completion, *valloc()* returns a pointer to the allocated memory. Otherwise,
30450 *valloc()* returns a null pointer and sets *errno* to indicate the error.30451 If *size* is 0, the behaviour is implementation-dependent; the value returned will be either a null
30452 pointer or a unique pointer. When *size* is 0 and *valloc()* returns a null pointer, *errno* is not
30453 modified.30454 **ERRORS**30455 The *valloc()* function will fail if:

30456 [ENOMEM] Storage space available is insufficient.

30457 **EXAMPLES**

30458 None.

30459 **APPLICATION USAGE**30460 Applications should avoid using *valloc()* but should use *malloc()* or *mmap()* instead. On systems
30461 with a large page size, the number of successful *valloc()* operations may be zero.30462 **FUTURE DIRECTIONS**

30463 None.

30464 **SEE ALSO**30465 *malloc()*, *sysconf()*, <stdlib.h>.30466 **CHANGE HISTORY**

30467 First released in Issue 4, Version 2.

30468 **Issue 5**

30469 Marked LEGACY.

30470 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

30471 **NAME**

30472 va_arg, va_end, va_start — handle variable argument list

30473 **SYNOPSIS**

30474 #include <stdarg.h>

30475 type va_arg(va_list ap, type);

30476 void va_end(va_list ap);

30477 void va_start(va_list ap, argN);

30478 **DESCRIPTION**

30479 Refer to <stdarg.h>.

30480 **CHANGE HISTORY**

30481 First released in Issue 4.

30482 Derived from the ANSI C standard.

30483 **NAME**

30484 vfork — create new process; share virtual memory

30485 **SYNOPSIS**30486 EX `#include <unistd.h>`30487 `pid_t vfork(void);`

30488

30489 **DESCRIPTION**

30490 The `vfork()` function has the same effect as `fork()`, except that the behaviour is undefined if the
 30491 process created by `vfork()` either modifies any data other than a variable of type `pid_t` used to
 30492 store the return value from `vfork()`, or returns from the function in which `vfork()` was called, or
 30493 calls any other function before successfully calling `_exit()` or one of the `exec` family of functions.

30494 **RETURN VALUE**

30495 Upon successful completion, `vfork()` returns 0 to the child process and returns the process ID of
 30496 the child process to the parent process. Otherwise, `-1` is returned to the parent, no child process
 30497 is created, and `errno` is set to indicate the error.

30498 **ERRORS**30499 The `vfork()` function will fail if:

30500 [EAGAIN] The system-wide limit on the total number of processes under execution
 30501 would be exceeded, or the system-imposed limit on the total number of
 30502 processes under execution by a single user would be exceeded.

30503 [ENOMEM] There is insufficient swap space for the new process.

30504 **EXAMPLES**

30505 None.

30506 **APPLICATION USAGE**30507 On some systems, `vfork()` is the same as `fork()`.

30508 The `vfork()` function differs from `fork()` only in that the child process can share code and data
 30509 with the calling process (parent process). This speeds cloning activity significantly at a risk to
 30510 the integrity of the parent process if `vfork()` is misused.

30511 The use of `vfork()` for any purpose except as a prelude to an immediate call to a function from
 30512 the `exec` family, or to `_exit()`, is not advised.

30513 The `vfork()` function can be used to create new processes without fully copying the address
 30514 space of the old process. If a forked process is simply going to call `exec`, the data space copied
 30515 from the parent to the child by `fork()` is not used. This is particularly inefficient in a paged
 30516 environment, making `vfork()` particularly useful. Depending upon the size of the parent's data
 30517 space, `vfork()` can give a significant performance improvement over `fork()`.

30518 The `vfork()` function can normally be used just like `fork()`. It does not work, however, to return
 30519 while running in the child's context from the caller of `vfork()` since the eventual return from
 30520 `vfork()` would then return to a no longer existent stack frame. Be careful, also, to call `_exit()`
 30521 rather than `exit()` if you cannot `exec`, since `exit()` flushes and closes standard I/O channels,
 30522 thereby damaging the parent process' standard I/O data structures. (Even with `fork()`, it is
 30523 wrong to call `exit()`, since buffered data would then be flushed twice.)

30524 If signal handlers are invoked in the child process after `vfork()`, they must follow the same rules
 30525 as other code in the child process.

30526 **FUTURE DIRECTIONS**

30527 None.

30528 **SEE ALSO**

30529 *exec*, *exit()*, *fork()*, *wait()*, **<unistd.h>**.

30530 **CHANGE HISTORY**

30531 First released in Issue 4, Version 2.

30532 **Issue 5**

30533 Moved from X/OPEN UNIX extension to BASE.

30534 **NAME**

30535 vfprintf, vprintf, vsnprintf, vsprintf — format output of a stdarg argument list

30536 **SYNOPSIS**

30537 #include <stdarg.h>

30538 #include <stdio.h>

30539 int vfprintf(FILE *stream, const char *format, va_list ap);

30540 int vprintf(const char *format, va_list ap);

30541 EX int vsnprintf(char *s, size_t n, const char *format, va_list ap);

30542 int vsprintf(char *s, const char *format, va_list ap);

30543 **DESCRIPTION**

30544 EX The *vprintf()*, *vfprintf()*, *vsnprintf()* and *vsprintf()* functions are the same as *printf()*, *fprintf()*,
 30545 *snprintf()* and *sprintf()* respectively, except that instead of being called with a variable number of
 30546 arguments, they are called with an argument list as defined by <stdarg.h>.

30547 These functions do not invoke the *va_end* macro. As these functions invoke the *va_arg* macro,
 30548 the value of *ap* after the return is indeterminate.

30549 **RETURN VALUE**30550 Refer to *printf()*.30551 **ERRORS**30552 Refer to *printf()*.30553 **EXAMPLES**

30554 None.

30555 **APPLICATION USAGE**30556 Applications using these functions should call *va_end(ap)* afterwards to clean up.30557 **FUTURE DIRECTIONS**

30558 None.

30559 **SEE ALSO**30560 *printf()*, <stdarg.h>, <stdio.h>.30561 **CHANGE HISTORY**

30562 First released in Issue 1.

30563 Derived from Issue 1 of the SVID.

30564 **Issue 4**

30565 The following changes are incorporated for alignment with the ISO C standard:

- 30566 • These functions are no longer marked as extensions.
- 30567 • The type of argument *format* is changed from **char *** to **const char ***.
- 30568 • Reference to the <varargs.h> header in the DESCRIPTION is replaced by <stdarg.h>. The
- 30569 last paragraph has also been added to indicate interactions with the *va_arg* and *va_end*
- 30570 macros.

30571 Other changes are incorporated as follows:

- 30572 • The APPLICATION USAGE section is added.
- 30573 • The FUTURE DIRECTIONS section is removed.

30574 **Issue 5**

30575 The *vsnprintf()* function is added.

|

30576 **NAME**

30577 vfwprintf, vwprintf, vswprintf — wide-character formatted output of a stdarg argument list

30578 **SYNOPSIS**

30579 #include <stdarg.h>

30580 #include <stdio.h>

30581 #include <wchar.h>

30582 int vwprintf(const wchar_t *format, va_list arg);

30583 int vfwprintf(FILE *stream, const wchar_t *format, va_list arg);

30584 int vswprintf(wchar_t *s, size_t n, const wchar_t *format,

30585 va_list arg);

30586 **DESCRIPTION**

30587 The *vwprintf()*, *vfwprintf()* and *vswprintf()* functions are the same as *wprintf()*, *fwprintf()* and *swprintf()* 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>.

30590 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.

30592 **RETURN VALUE**

30593 Refer to *fwprintf()*.

30594 **ERRORS**

30595 Refer to *fwprintf()*.

30596 **EXAMPLES**

30597 None.

30598 **APPLICATION USAGE**

30599 Applications using these functions should call *va_end(ap)* afterwards to clean up.

30600 **FUTURE DIRECTIONS**

30601 None.

30602 **SEE ALSO**

30603 *fwprintf()*, <stdarg.h>, <stdio.h>, <wchar.h>.

30604 **CHANGE HISTORY**

30605 First released in Issue 5.

30606 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

30607 NAME

30608 vsprintf, vsnprintf — print formatted output

30609 SYNOPSIS

30610 #include <stdarg.h>

30611 #include <stdio.h>

30612 int vsprintf(char *s, const char *format, va_list ap);

30613 EX int vsnprintf(char *s, size_t n, const char *format, va_list ap);

30614

30615 DESCRIPTION

30616 Refer to *vfprintf()*.

30617 CHANGE HISTORY

30618 First released in Issue 5.

30619 NAME

30620 wait, waitpid — wait for a child process to stop or terminate

30621 SYNOPSIS

30622 OH #include <sys/types.h>

30623 #include <sys/wait.h>

30624 pid_t wait(int *stat_loc);

30625 pid_t waitpid(pid_t pid, int *stat_loc, int options);

30626 DESCRIPTION

30627 The *wait()* and *waitpid()* functions allow the calling process to obtain status information
 30628 pertaining to one of its child processes. Various options permit status information to be
 30629 obtained for child processes that have terminated or stopped. If status information is available
 30630 for two or more child processes, the order in which their status is reported is unspecified.

30631 The *wait()* function will suspend execution of the calling thread until status information for one
 30632 of its terminated child processes is available, or until delivery of a signal whose action is either
 30633 to execute a signal-catching function or to terminate the process. If more than one thread is
 30634 suspended in *wait()* or *waitpid()* awaiting termination of the same process, exactly one thread
 30635 will return the process status at the time of the target process termination. If status information
 30636 is available prior to the call to *wait()*, return will be immediate.

30637 The *waitpid()* function will behave identically to *wait()*, if the *pid* argument is (**pid_t**)−1 and the
 30638 *options* argument is 0. Otherwise, its behaviour will be modified by the values of the *pid* and
 30639 *options* arguments.

30640 The *pid* argument specifies a set of child processes for which status is requested. The *waitpid()*
 30641 function will only return the status of a child process from this set:

- 30642 • If *pid* is equal to (**pid_t**)−1, status is requested for any child process. In this respect, *waitpid()*
 30643 is then equivalent to *wait()*.
- 30644 • If *pid* is greater than 0, it specifies the process ID of a single child process for which status is
 30645 requested.
- 30646 • If *pid* is 0, status is requested for any child process whose process group ID is equal to that of
 30647 the calling process.
- 30648 • If *pid* is less than (**pid_t**)−1, status is requested for any child process whose process group ID
 30649 is equal to the absolute value of *pid*.

30650 The *options* argument is constructed from the bitwise-inclusive OR of zero or more of the
 30651 following flags, defined in the header <sys/wait.h>.

30652 EX WCONTINUED The *waitpid()* function will report the status of any continued child process
 30653 specified by *pid* whose status has not been reported since it continued from a
 30654 job control stop.

30655 WNOHANG The *waitpid()* function will not suspend execution of the calling thread if
 30656 status is not immediately available for one of the child processes specified by
 30657 *pid*.

30658 WUNTRACED The status of any child processes specified by *pid* that are stopped, and whose
 30659 status has not yet been reported since they stopped, will also be reported to
 30660 the requesting process.

30661 EX If the calling process has SA_NOCLDWAIT set or has SIGCHLD set to SIG_IGN, and the
 30662 process has no unwaited for children that were transformed into zombie processes, the calling
 30663 thread will block until all of the children of the process containing the calling thread terminate,

and *wait()* and *waitpid()* will fail and set *errno* to [ECHILD].

If *wait()* or *waitpid()* return because the status of a child process is available, these functions will 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 will be stored in the location pointed to by *stat_loc*. If and only if the status returned is from a terminated child process that returned 0 from *main()* or passed 0 as the *status* argument to *_exit()* or *exit()*, the value stored at the location pointed to by *stat_loc* will be 0. 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.

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.

If the information pointed to by *stat_loc* was stored by a call to *waitpid()* that specified the WUNTRACED flag and did not specify the WCONTINUED flag, exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), and WIFSTOPPED(**stat_loc*), will 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*), will 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*) will 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*), will evaluate to a non-zero value.

There may be additional implementation-dependent circumstances under which *wait()* or *waitpid()* report status. This will not occur unless the calling process or one of its child processes

30710 explicitly makes use of a non-standard extension. In these cases the interpretation of the
 30711 reported status is implementation-dependent.

30712 If a parent process terminates without waiting for all of its child processes to terminate, the
 30713 remaining child processes will be assigned a new parent process ID corresponding to an
 30714 implementation-dependent system process.

30715 RETURN VALUE

30716 If *wait()* or *waitpid()* returns because the status of a child process is available, these functions
 30717 will return a value equal to the process ID of the child process for which status is reported. If
 30718 *wait()* or *waitpid()* returns due to the delivery of a signal to the calling process, *-1* will be
 30719 returned and *errno* will be set to [EINTR]. If *waitpid()* was invoked with WNOHANG set in
 30720 *options*, it has at least one child process specified by *pid* for which status is not available, and
 30721 status is not available for any process specified by *pid*, *0* will be returned. Otherwise, (*pid_t*)-*1*
 30722 will be returned, and *errno* will be set to indicate the error.

30723 ERRORS

30724 The *wait()* function will fail if:

30725 [ECHILD] The calling process has no existing unwaited-for child processes.

30726 [EINTR] The function was interrupted by a signal. The value of the location pointed to
 30727 by *stat_loc* is undefined.

30728 The *waitpid()* function will fail if:

30729 [ECHILD] The process or process group specified by *pid* does not exist or is not a child of
 30730 the calling process.

30731 [EINTR] The function was interrupted by a signal. The value of the location pointed to
 30732 by *stat_loc* is undefined.

30733 [EINVAL] The *options* argument is not valid.

30734 EXAMPLES

30735 None.

30736 APPLICATION USAGE

30737 None.

30738 FUTURE DIRECTIONS

30739 None.

30740 SEE ALSO

30741 *exec*, *exit()*, *fork()*, *wait3()*, *waitid()*, <sys/types.h>, <sys/wait.h>.

30742 CHANGE HISTORY

30743 First released in Issue 1.

30744 Derived from Issue 1 of the SVID.

30745 Issue 4

30746 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 30747 • Text describing conditions under which *0* will be returned when WNOHUNG is set in *options*
- 30748 is added to the RETURN VALUE section.

30749 Other changes are incorporated as follows:

- 30750 • The <**sys/types.h**> header is now marked as optional (OH); this header need not be included |
- 30751 on XSI-conformant systems.
- 30752 • Error return values throughout the DESCRIPTION and RETURN VALUE sections are |
- 30753 changed to show the proper casting (that is, (**pid_t**) –1).
- 30754 • The words “If the implementation supports job control” are removed from the description of
- 30755 WUNTRACED. This is because job control is defined as mandatory for Issue 4 conforming
- 30756 implementations.

30757 **Issue 4, Version 2**

30758 The following changes are incorporated in the *DESCRIPTION* for X/OPEN UNIX conformance:

- 30759 • The WCONTINUED *options* flag and the WIFCONTINUED(*stat_val*) macro are added.
- 30760 • Text following the list of *options* flags explains the implications of setting the |
- 30761 SA_NOCLDWAIT signal flag, or setting SIGCHLD to SIG_IGN.
- 30762 • Text following the list of macros, which explains what macros return non-zero values in
- 30763 certain cases, is expanded and the value of the WCONTINUED flag on the previous call to |
- 30764 *waitpid()* is taken into account.

30765 **Issue 5**

30766 The DESCRIPTION is updated for alignment with the POSIX Threads Extension. |

30767 **NAME**30768 wait3 — wait for a child process to change state (**LEGACY**)30769 **SYNOPSIS**

30770 EX #include <sys/wait.h>

30771 pid_t wait3 (int *stat_loc, int options, struct rusage *resource_usage);
3077230773 **DESCRIPTION**30774 The *wait3()* function allows the calling thread to obtain status information for specified child
30775 processes.

30776 The following call:

30777 wait3(stat_loc, options, resource_usage);

30778 is equivalent to the call:

30779 waitpid((pid_t)-1, stat_loc, options);

30780 except that on successful completion, if the *resource_usage* argument to *wait3()* is not a null
30781 pointer, the rusage structure that the third argument points to is filled in for the child process
30782 identified by the return value.

30783 This interface need not be reentrant.

30784 **RETURN VALUE**30785 See *waitpid()*.30786 **ERRORS**30787 In addition to the error conditions specified on *waitpid()*, under the following conditions, *wait3()*
30788 may fail and set *errno* to:30789 [ECHILD] The calling process has no existing unwaited-for child processes, or if the set
30790 of processes specified by the argument *pid* can never be in the states specified
30791 by the argument *options*.30792 [ENOSYS] The *wait3()* function is not supported on this implementation.30793 **EXAMPLES**

30794 None.

30795 **APPLICATION USAGE**30796 New applications should use *waitpid()*.30797 **FUTURE DIRECTIONS**

30798 None.

30799 **SEE ALSO**30800 *exec*, *exit()*, *fork()*, *pause()*, *waitpid()*, <sys/wait.h>.30801 **CHANGE HISTORY**

30802 First released in Issue 4, Version 2.

30803 **Issue 5**

30804 Moved from X/OPEN UNIX extension to BASE.

30805 A note indicating that this interface need not be reentrant is added to the DESCRIPTION.

30806 Marked LEGACY.

30807 NAME

30808 waitid — wait for a child process to change state

30809 SYNOPSIS

30810 EX

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

30811

```
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

30812

30813 DESCRIPTION

30814 The *waitid()* function suspends the calling thread until one child of the process containing the
 30815 calling thread changes state. It records the current state of a child in the structure pointed to by
 30816 *infop*. If a child process changed state prior to the call to *waitid()*, *waitid()* returns immediately.
 30817 If more than one thread is suspended in *wait()* or *waitpid()* waiting termination of the same
 30818 process, exactly one thread will return the process status at the time of the target process
 30819 termination

30820 The *idtype* and *id* arguments are used to specify which children *waitid()* will wait for.

30821 If *idtype* is P_PID, *waitid()* will wait for the child with a process ID equal to (**pid_t**)*id*.

30822 If *idtype* is P_PGID, *waitid()* will wait for any child with a process group ID equal to (**pid_t**)*id*.

30823 If *idtype* is P_ALL, *waitid()* will wait for any children and *id* is ignored.

30824 The *options* argument is used to specify which state changes *waitid()* will wait for. It is formed
 30825 by OR-ing together one or more of the following flags:

30826 WEXITED Wait for processes that have exited.

30827 WSTOPPED Status will be returned for any child that has stopped upon receipt of a signal.

30828 WCONTINUED Status will be returned for any child that was stopped and has been continued.

30829 WNOHANG Return immediately if there are no children to wait for.

30830 WNOWAIT Keep the process whose status is returned in *infop* in a waitable state. This
 30831 will not affect the state of the process; the process may be waited for again
 30832 after this call completes.

30833 The *infop* argument must point to a **siginfo_t** structure. If *waitid()* returns because a child
 30834 process was found that satisfied the conditions indicated by the arguments *idtype* and *options*,
 30835 then the structure pointed to by *infop* will be filled in by the system with the status of the
 30836 process. The **si_signo** member will always be equal to SIGCHLD.

30837 RETURN VALUE

30838 If *waitid()* returns due to the change of state of one of its children, 0 is returned. Otherwise, -1 is
 30839 returned and *errno* is set to indicate the error.

30840 ERRORS

30841 The *waitid()* function will fail if:

30842 [ECHILD] The calling process has no existing unwaited-for child processes.

30843 [EINTR] The *waitid()* function was interrupted by a signal.

30844 [EINVAL] An invalid value was specified for *options*, or *idtype* and *id* specify an invalid
 30845 set of processes.

30846 **EXAMPLES**

30847 None.

30848 **APPLICATION USAGE**

30849 None.

30850 **FUTURE DIRECTIONS**

30851 None.

30852 **SEE ALSO**30853 *exec*, *exit()*, *wait()*, <sys/wait.h>.30854 **CHANGE HISTORY**

30855 First released in Issue 4, Version 2.

30856 **Issue 5**

30857 Moved from X/OPEN UNIX extension to BASE.

30858 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

30859 NAME

30860 waitpid — wait for a child process to stop or terminate

30861 SYNOPSIS

30862 OH #include <sys/types.h>

30863 #include <sys/wait.h>

30864 pid_t waitpid(pid_t *pid*, int **stat_loc*, int *options*);

30865 DESCRIPTION

30866 Refer to *wait()*.

30867 CHANGE HISTORY

30868 First released in Issue 4, Version 2.

30869 **NAME**30870 `wrtomb` — convert a wide-character code to a character (restartable)30871 **SYNOPSIS**30872 `#include <stdio.h>`30873 `size_t wrtomb(char *s, wchar_t wc, mbstate_t *ps);`30874 **DESCRIPTION**30875 If *s* is a null pointer, the `wrtomb()` function is equivalent to the call:30876 `wrtomb(buf, L'\0', ps)`30877 where *buf* is an internal buffer.

30878 If *s* is not a null pointer, the `wrtomb()` function determines the number of bytes needed to
 30879 represent the character that corresponds to the wide-character given by *wc* (including any shift
 30880 sequences), and stores the resulting bytes in the array whose first element is pointed to by *s*. At
 30881 most MB_CUR_MAX bytes are stored. If *wc* is a null wide-character, a null byte is stored,
 30882 preceded by any shift sequence needed to restore the initial shift state. The resulting state
 30883 described is the initial conversion state.

30884 If *ps* is a null pointer, the `wrtomb()` function uses its own internal `mbstate_t` object, which is
 30885 initialised at program startup to the initial conversion state. Otherwise, the `mbstate_t` object
 30886 pointed to by *ps* is used to completely describe the current conversion state of the associated
 30887 character sequence. The implementation will behave as if no function defined in this
 30888 specification calls `wrtomb()`.

30889 The behaviour of this function is affected by the LC_CTYPE category of the current locale.

30890 **RETURN VALUE**

30891 The `wrtomb()` function returns the number of bytes stored in the array object (including any
 30892 shift sequences). When *wc* is not a valid wide-character, an encoding error occurs. In this case,
 30893 the function stores the value of the macros EILSEQ in *errno* and returns **(size_t)−1**; the
 30894 conversion state is undefined.

30895 **ERRORS**30896 The `wrtomb()` function may fail if:30897 [EINVAL] *ps* points to an object that contains an invalid conversion state.

30898 [EILSEQ] Invalid wide-character code is detected.

30899 **EXAMPLES**

30900 None.

30901 **APPLICATION USAGE**

30902 None.

30903 **FUTURE DIRECTIONS**

30904 None.

30905 **SEE ALSO**30906 `mbsinit()`, `<wchar.h>`.30907 **CHANGE HISTORY**

30908 First released in Issue 5.

30909 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

30910 NAME

30911 `wcscat` — concatenate two wide-character strings

30912 SYNOPSIS

30913 `#include <wchar.h>`

30914 `wchar_t *wcscat(wchar_t *ws1, const wchar_t *ws2);`

30915 DESCRIPTION

30916 The `wcscat()` function appends a copy of the wide-character string pointed to by `ws2` (including
30917 the terminating null wide-character code) to the end of the wide-character string pointed to by
30918 `ws1`. The initial wide-character code of `ws2` overwrites the null wide-character code at the end of
30919 `ws1`. If copying takes place between objects that overlap, the behaviour is undefined.

30920 RETURN VALUE

30921 The `wcscat()` function returns `s1`; no return value is reserved to indicate an error.

30922 ERRORS

30923 No errors are defined.

30924 EXAMPLES

30925 None.

30926 APPLICATION USAGE

30927 None.

30928 FUTURE DIRECTIONS

30929 None.

30930 SEE ALSO

30931 `wcsncat()`, `<wchar.h>`.

30932 CHANGE HISTORY

30933 First released in Issue 4.

30934 Derived from the MSE working draft.

30935 **NAME**30936 `wcschr` — wide-character string scanning operation30937 **SYNOPSIS**30938 `#include <wchar.h>`30939 `wchar_t *wcschr(const wchar_t *ws, wchar_t wc);`30940 **DESCRIPTION**

30941 The `wcschr()` function locates the first occurrence of `wc` in the wide-character string pointed to by
30942 `ws`. The value of `wc` must be a character representable as a type **wchar_t** and must be a wide-
30943 character code corresponding to a valid character in the current locale. The terminating null
30944 wide-character code is considered to be part of the wide-character string.

30945 **RETURN VALUE**

30946 Upon completion, `wcschr()` returns a pointer to the wide-character code, or a null pointer if the
30947 wide-character code is not found.

30948 **ERRORS**

30949 No errors are defined.

30950 **EXAMPLES**

30951 None.

30952 **APPLICATION USAGE**

30953 None.

30954 **FUTURE DIRECTIONS**

30955 None.

30956 **SEE ALSO**30957 `wcsrchr()`, `<wchar.h>`.30958 **CHANGE HISTORY**

30959 First released in Issue 4.

30960 Derived from the MSE working draft.

30961 NAME

30962 **wcscmp** — compare two wide-character strings

30963 SYNOPSIS

30964 `#include <wchar.h>`

30965 `int wcscmp(const wchar_t *ws1, const wchar_t *ws2);`

30966 DESCRIPTION

30967 The *wcscmp()* function compares the wide-character string pointed to by *ws1* to the wide-character string pointed to by *ws2*.

30969 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.

30971 RETURN VALUE

30972 Upon completion, *wcscmp()* returns 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.

30975 ERRORS

30976 No errors are defined.

30977 EXAMPLES

30978 None.

30979 APPLICATION USAGE

30980 None.

30981 FUTURE DIRECTIONS

30982 None.

30983 SEE ALSO

30984 *wcsncmp()*, `<wchar.h>`.

30985 CHANGE HISTORY

30986 First released in Issue 4.

30987 Derived from the MSE working draft.

30988 **NAME**

30989 wcscoll — wide-character string comparison using collating information

30990 **SYNOPSIS**

30991 #include <wchar.h>

30992 int wcscoll(const wchar_t *ws1, const wchar_t *ws2);

30993 **DESCRIPTION**

30994 The *wcscoll()* function compares 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.

30997 The *wcscoll()* function will not change the setting of **errno** if successful.

30998 An application wishing to check for error situations should set *errno* to 0 before calling *wcscoll()*.

30999 If *errno* is non-zero on return, an error has occurred.

31000 **RETURN VALUE**

31001 Upon successful completion, *wcscoll()* returns 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 to the current locale. On error, *wcscoll()* may set *errno*, but no return value is reserved to indicate an error.

31006 **ERRORS**

31007 The *wcscoll()* function may fail if:

31008	[EINVAL]	The <i>ws1</i> or <i>ws2</i> arguments contain wide-character codes outside the domain of the collating sequence.
31009		

31010 **EXAMPLES**

31011 None.

31012 **APPLICATION USAGE**

31013 The *wcsxfrm()* and *wscmp()* functions should be used for sorting large lists.

31014 **FUTURE DIRECTIONS**

31015 None.

31016 **SEE ALSO**

31017 *wscmp()*, *wcsxfrm()*, <wchar.h>.

31018 **CHANGE HISTORY**

31019 First released in Issue 4.

31020 Derived from the MSE working draft.

31021 **Issue 5**

31022 Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

31023 The DESCRIPTION is updated to indicate that **errno** will not be changed if the function is successful.

31024

31025 NAME

31026 **wcscpy** — copy a wide-character string

31027 SYNOPSIS

31028 `#include <wchar.h>`

31029 `wchar_t *wcscpy(wchar_t *ws1, const wchar_t *ws2);`

31030 DESCRIPTION

31031 The *wcscpy()* function copies the wide-character string pointed to by *ws2* (including the
31032 terminating null wide-character code) into the array pointed to by *ws1*. If copying takes place
31033 between objects that overlap, the behaviour is undefined.

31034 RETURN VALUE

31035 The *wcscpy()* function returns *ws1*; no return value is reserved to indicate an error.

31036 ERRORS

31037 No errors are defined.

31038 EXAMPLES

31039 None.

31040 APPLICATION USAGE

31041 Wide-character code movement is performed differently in different implementations. Thus
31042 overlapping moves may yield surprises.

31043 FUTURE DIRECTIONS

31044 None.

31045 SEE ALSO

31046 *wcsncpy()*, *<wchar.h>*.

31047 CHANGE HISTORY

31048 First released in Issue 4.

31049 Derived from the MSE working draft.

31050 **NAME**

31051 wcscspn — get length of a complementary wide substring

31052 **SYNOPSIS**

31053 #include <wchar.h>

31054 size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);

31055 **DESCRIPTION**

31056 The *wcscspn()* function computes the length of the maximum initial segment of the wide-
31057 character string pointed to by *ws1* which consists entirely of wide-character codes *not* from the
31058 wide-character string pointed to by *ws2*.

31059 **RETURN VALUE**

31060 The *wcscspn()* function returns the length of the initial substring of *ws1*; no return value is
31061 reserved to indicate an error.

31062 **ERRORS**

31063 No errors are defined.

31064 **EXAMPLES**

31065 None.

31066 **APPLICATION USAGE**

31067 None.

31068 **FUTURE DIRECTIONS**

31069 None.

31070 **SEE ALSO**31071 *wcsspn()*, <wchar.h>.31072 **CHANGE HISTORY**

31073 First released in Issue 4.

31074 Derived from the MSE working draft.

31075 **Issue 5**

31076 The RETURN VALUE section is updated to indicate that *wcscspn()* returns the length of *ws1*,
31077 rather than *ws1* itself.

31078 **NAME**

31079 wcsftime — convert date and time to a wide-character string

31080 **SYNOPSIS**

31081 #include <wchar.h>

```
31082       size_t wcsftime(wchar_t *wcs, size_t maxsize, const wchar_t *format,
31083                       const struct tm *timptr);
```

31084 **DESCRIPTION**31085 The *wcsftime()* function is equivalent to the *strftime()* function, except that:

- 31086 • The argument *wcs* points to the initial element of an array of wide-characters into which the
- 31087 generated output is to be placed.
- 31088 • The argument *maxsize* indicates the maximum number of wide-characters to be placed in the
- 31089 output array.
- 31090 • The argument *format* is a wide-character string and the conversion specifications are replaced
- 31091 by corresponding sequences of wide-characters.
- 31092 • The return value indicates the number of wide-characters placed in the output array.

31093 If copying takes place between objects that overlap, the behaviour is undefined.

31094 **RETURN VALUE**

31095 If the total number of resulting wide-character codes including the terminating null wide-

31096 character code is no more than *maxsize*, *wcsftime()* returns the number of wide-character codes

31097 placed into the array pointed to by *wcs*, not including the terminating null wide-character code.

31098 Otherwise 0 is returned and the contents of the array are indeterminate. If the function is not

31099 implemented, *errno* will be set to indicate the error.

31100 **ERRORS**

31101 No errors are defined.

31102 **EXAMPLES**

31103 None.

31104 **APPLICATION USAGE**

31105 None.

31106 **FUTURE DIRECTIONS**

31107 None.

31108 **SEE ALSO**31109 *strftime()*, <wchar.h>.31110 **CHANGE HISTORY**

31111 First released in Issue 4.

31112 **Issue 5**

31113 Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

31114 Aligned with ISO/IEC 9899:1990/Amendment 1:1994 (E). Specifically, the type of the format

31115 argument is changed from **const char*** to **const wchar_t***.

31116 **NAME**

31117 wcslen — get wide-character string length

31118 **SYNOPSIS**

31119 #include <wchar.h>

31120 size_t wcslen(const wchar_t *ws);

31121 **DESCRIPTION**

31122 The *wcslen()* function computes the number of wide-character codes in the wide-character string
31123 to which *ws* points, not including the terminating null wide-character code.

31124 **RETURN VALUE**31125 The *wcslen()* function returns the length of *ws*; no return value is reserved to indicate an error.31126 **ERRORS**

31127 No errors are defined.

31128 **EXAMPLES**

31129 None.

31130 **APPLICATION USAGE**

31131 None.

31132 **FUTURE DIRECTIONS**

31133 None.

31134 **SEE ALSO**

31135 <wchar.h>.

31136 **CHANGE HISTORY**

31137 First released in Issue 4.

31138 Derived from the MSE working draft.

31139 NAME

31140 **wcsncat** — concatenate part of two wide-character strings

31141 SYNOPSIS

31142 `#include <wchar.h>`

31143 `wchar_t *wcsncat(wchar_t *ws1, const wchar_t *ws2, size_t n);`

31144 DESCRIPTION

31145 The `wcsncat()` function appends not more than *n* wide-character codes (a null wide-character
31146 code and wide-character codes that follow it are not appended) from the array pointed to by *ws2*
31147 to the end of the wide-character string pointed to by *ws1*. The initial wide-character code of *ws2*
31148 overwrites the null wide-character code at the end of *ws1*. A terminating null wide-character
31149 code is always appended to the result. If copying takes place between objects that overlap, the
31150 behaviour is undefined.

31151 RETURN VALUE

31152 The `wcsncat()` function returns *ws1*; no return value is reserved to indicate an error.

31153 ERRORS

31154 No errors are defined.

31155 EXAMPLES

31156 None.

31157 APPLICATION USAGE

31158 None.

31159 FUTURE DIRECTIONS

31160 None.

31161 SEE ALSO

31162 `wscat()`, `<wchar.h>`.

31163 CHANGE HISTORY

31164 First released in Issue 4.

31165 Derived from the MSE working draft.

31166 **NAME**

31167 wcsncmp — compare part of two wide-character strings

31168 **SYNOPSIS**

31169 #include <wchar.h>

31170 int wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);

31171 **DESCRIPTION**

31172 The *wcsncmp()* function compares not more than *n* wide-character codes (wide-character codes
31173 that follow a null wide-character code are not compared) from the array pointed to by *ws1* to the
31174 array pointed to by *ws2*.

31175 The sign of a non-zero return value is determined by the sign of the difference between the
31176 values of the first pair of wide-character codes that differ in the objects being compared.

31177 **RETURN VALUE**

31178 Upon successful completion, *wcsncmp()* returns an integer greater than, equal to or less than 0, if
31179 the possibly null-terminated array pointed to by *ws1* is greater than, equal to or less than the
31180 possibly null-terminated array pointed to by *ws2* respectively.

31181 **ERRORS**

31182 No errors are defined.

31183 **EXAMPLES**

31184 None.

31185 **APPLICATION USAGE**

31186 None.

31187 **FUTURE DIRECTIONS**

31188 None.

31189 **SEE ALSO**31190 *wscmp()*, <wchar.h>.31191 **CHANGE HISTORY**

31192 First released in Issue 4.

31193 Derived from the MSE working draft.

31194 **NAME**

31195 wcsncpy — copy part of a wide-character string

31196 **SYNOPSIS**

31197 #include <wchar.h>

31198 wchar_t *wcsncpy(wchar_t *ws1, const wchar_t *ws2, size_t n);

31199 **DESCRIPTION**

31200 The *wcsncpy()* function copies not more than *n* wide-character codes (wide-character codes that
31201 follow a null wide-character code are not copied) from the array pointed to by *ws2* to the array
31202 pointed to by *ws1*. If copying takes place between objects that overlap, the behaviour is
31203 undefined.

31204 If the array pointed to by *ws2* is a wide-character string that is shorter than *n* wide-character
31205 codes, null wide-character codes are appended to the copy in the array pointed to by *ws1*, until *n*
31206 wide-character codes in all are written.

31207 **RETURN VALUE**31208 The *wcsncpy()* function returns *ws1*; no return value is reserved to indicate an error.31209 **ERRORS**

31210 No errors are defined.

31211 **EXAMPLES**

31212 None.

31213 **APPLICATION USAGE**

31214 Wide-character code movement is performed differently in different implementations. Thus
31215 overlapping moves may yield surprises.

31216 If there is no null wide-character code in the first *n* wide-character codes of the array pointed to
31217 by *ws2*, the result will not be null-terminated.

31218 **FUTURE DIRECTIONS**

31219 None.

31220 **SEE ALSO**31221 *wscpy()*, <wchar.h>.31222 **CHANGE HISTORY**

31223 First released in Issue 4.

31224 Derived from the MSE working draft.

31225 **NAME**

31226 wcsprk — scan wide-character string for a wide-character code

31227 **SYNOPSIS**

31228 #include <wchar.h>

31229 wchar_t *wcsprk(const wchar_t *ws1, const wchar_t *ws2);

31230 **DESCRIPTION**

31231 The *wcsprk()* function locates the first occurrence in the wide-character string pointed to by *ws1*
31232 of any wide-character code from the wide-character string pointed to by *ws2*.

31233 **RETURN VALUE**

31234 Upon successful completion, *wcsprk()* returns a pointer to the wide-character code or a null
31235 pointer if no wide-character code from *ws2* occurs in *ws1*.

31236 **ERRORS**

31237 No errors are defined.

31238 **EXAMPLES**

31239 None.

31240 **APPLICATION USAGE**

31241 None.

31242 **FUTURE DIRECTIONS**

31243 None.

31244 **SEE ALSO**

31245 *wcschr()*, *wcsrchr()*, <wchar.h>.

31246 **CHANGE HISTORY**

31247 First released in Issue 4.

31248 Derived from the MSE working draft.

31249 NAME

31250 wcsrchr — wide-character string scanning operation

31251 SYNOPSIS

31252 #include <wchar.h>

31253 wchar_t *wcsrchr(const wchar_t *ws, wchar_t wc);

31254 DESCRIPTION

31255 The *wcsrchr()* function locates the last occurrence of *wc* in the wide-character string pointed to
31256 by *ws*. The value of *wc* must be a character representable as a type **wchar_t** and must be a wide-
31257 character code corresponding to a valid character in the current locale. The terminating null
31258 wide-character code is considered to be part of the wide-character string.

31259 RETURN VALUE

31260 Upon successful completion, *wcsrchr()* returns a pointer to the wide-character code or a null
31261 pointer if *wc* does not occur in the wide-character string.

31262 ERRORS

31263 No errors are defined.

31264 EXAMPLES

31265 None.

31266 APPLICATION USAGE

31267 None.

31268 FUTURE DIRECTIONS

31269 None.

31270 SEE ALSO

31271 *wcschr()*, <**wchar.h**>.

31272 CHANGE HISTORY

31273 First released in Issue 4.

31274 Derived from the MSE working draft.

31275 **NAME**31276 `wcsrtombs` — convert a wide-character string to a character string (restartable)31277 **SYNOPSIS**31278 `#include <wchar.h>`31279 `size_t wcsrtombs(char *dst, const wchar_t **src, size_t len,`
31280 `mbstate_t *ps);`31281 **DESCRIPTION**31282 The `wcsrtombs()` function converts a sequence of wide-characters from the array indirectly
31283 pointed to by `src` into a sequence of corresponding characters, beginning in the conversion state
31284 described by the object pointed to by `ps`. If `dst` is not a null pointer, the converted characters are
31285 then stored into the array pointed to by `dst`. Conversion continues up to and including a
31286 terminating null wide-character, which is also stored. Conversion stops earlier in the following
31287 cases:

- 31288 • When a code is reached that does not correspond to a valid character.
-
- 31289 • When the next character would exceed the limit of
- `len`
- total bytes to be stored in the array
-
- 31290 pointed to by
- `dst`
- (and
- `dst`
- is not a null pointer).

31291 Each conversion takes place as if by a call to the `wcrtomb()` function.31292 If `dst` is not a null pointer, the pointer object pointed to by `src` is assigned either a null pointer (if
31293 conversion stopped due to reaching a terminating null wide-character) or the address just past
31294 the last wide-character converted (if any). If conversion stopped due to reaching a terminating
31295 null wide-character, the resulting state described is the initial conversion state.31296 If `ps` is a null pointer, the `wcsrtombs()` function uses its own internal `mbstate_t` object, which is
31297 initialised at program startup to the initial conversion state. Otherwise, the `mbstate_t` object
31298 pointed to by `ps` is used to completely describe the current conversion state of the associated
31299 character sequence. The implementation will behave as if no function defined in this
31300 specification calls `wcsrtombs()`.31301 The behaviour of this function is affected by the `LC_CTYPE` category of the current locale.31302 **RETURN VALUE**31303 If conversion stops because a code is reached that does not correspond to a valid character, an
31304 encoding error occurs. In this case, the `wcsrtombs()` function stores the value of the macro
31305 `EILSEQ` in `errno` and returns `(size_t)-1`; the conversion state is undefined. Otherwise, it returns
31306 the number of bytes in the resulting character sequence, not including the terminating null (if
31307 any).31308 **ERRORS**31309 The `wcsrtombs()` function may fail if:

- 31310
- `[EINVAL]`
- `ps`
- points to an object that contains an invalid conversion state.
-
- 31311
- `[EILSEQ]`
- A wide-character code does not correspond to a valid character.

31312 **EXAMPLES**

31313 None.

31314 **APPLICATION USAGE**

31315 None.

31316 **FUTURE DIRECTIONS**

31317 None.

31318 **SEE ALSO**31319 *mbsinit()*, *wcrtomb()*, **<wchar.h>**.31320 **CHANGE HISTORY**

31321 First released in Issue 5.

31322 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

31323 **NAME**

31324 wcssp — get length of a wide substring

31325 **SYNOPSIS**

31326 #include <wchar.h>

31327 size_t wcssp(const wchar_t *ws1, const wchar_t *ws2);

31328 **DESCRIPTION**

31329 The *wcssp()* function computes the length of the maximum initial segment of the wide-
31330 character string pointed to by *ws1* which consists entirely of wide-character codes from the
31331 wide-character string pointed to by *ws2*.

31332 **RETURN VALUE**31333 The *wcssp()* function returns the length *ws1*; no return value is reserved to indicate an error.31334 **ERRORS**

31335 No errors are defined.

31336 **EXAMPLES**

31337 None.

31338 **APPLICATION USAGE**

31339 None.

31340 **FUTURE DIRECTIONS**

31341 None.

31342 **SEE ALSO**31343 *wscspn()*, <wchar.h>.31344 **CHANGE HISTORY**

31345 First released in Issue 4.

31346 Derived from the MSE working draft.

31347 **Issue 5**

31348 The RETURN VALUE section is updated to indicate that *wcssp()* returns the length of *ws1*
31349 rather than *ws1* itself.

31350 **NAME**31351 **wcsstr** — find a wide-character substring31352 **SYNOPSIS**31353 `#include <wchar.h>`31354 `wchar_t *wcsstr(const wchar_t *ws1, const wchar_t *ws2);`31355 **DESCRIPTION**

31356 The *wcsstr()* function locates the first occurrence in the wide-character string pointed to by *ws1*
31357 of the sequence of wide-characters (excluding the terminating null wide-character) in the wide-
31358 character string pointed to by *ws2*.

31359 **RETURN VALUE**

31360 On successful completion, *wcsstr()* returns a pointer to the located wide-character string, or a
31361 null pointer if the wide-character string is not found.

31362 If *ws2* points to a wide-character string with zero length, the function returns *ws1*.31363 **ERRORS**

31364 No errors are defined.

31365 **EXAMPLES**

31366 None.

31367 **APPLICATION USAGE**

31368 None.

31369 **FUTURE DIRECTIONS**

31370 None.

31371 **SEE ALSO**31372 *wchr()*, *<wchar.h>*.31373 **CHANGE HISTORY**

31374 First released in Issue 5.

31375 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

31376 NAME

31377 wcstod — convert a wide-character string to a double-precision number

31378 SYNOPSIS

31379 #include <wchar.h>

31380 double wcstod(const wchar_t *nptr, wchar_t **endptr);

31381 DESCRIPTION

31382 The *wcstod()* function converts the initial portion of the wide-character string pointed to by *nptr*
 31383 to **double** representation. First it decomposes the input wide-character string into three parts:
 31384 an initial, possibly empty, sequence of white-space wide-character codes (as specified by
 31385 *iswspace()*); a subject sequence interpreted as a floating-point constant; and a final wide-
 31386 character string of one or more unrecognised wide-character codes, including the terminating
 31387 null wide-character code of the input wide-character string. Then it attempts to convert the
 31388 subject sequence to a floating-point number, and returns the result.

31389 The expected form of the subject sequence is an optional + or – sign, then a non-empty sequence
 31390 of digits optionally containing a radix, then an optional exponent part. An exponent part
 31391 consists of e or E, followed by an optional sign, followed by one or more decimal digits. The
 31392 subject sequence is defined as the longest initial subsequence of the input wide-character string,
 31393 starting with the first non-white-space wide-character code, that is of the expected form. The
 31394 subject sequence contains no wide-character codes if the input wide-character string is empty or
 31395 consists entirely of white-space wide-character codes, or if the first wide-character code that is
 31396 not white space other than a sign, a digit or a radix.

31397 If the subject sequence has the expected form, the sequence of wide-character codes starting
 31398 with the first digit or the radix (whichever occurs first) is interpreted as a floating constant as
 31399 defined in the C language, except that the radix is used in place of a period, and that if neither an
 31400 exponent part nor a radix appears, a radix is assumed to follow the last digit in the wide-
 31401 character string. If the subject sequence begins with a minus sign, the value resulting from the
 31402 conversion is negated. A pointer to the final wide-character string is stored in the object pointed
 31403 to by *endptr*, provided that *endptr* is not a null pointer.

31404 The radix is defined in the program's locale (category LC_NUMERIC). In the POSIX locale, or in
 31405 a locale where the radix is not defined, the radix defaults to a period (.).

31406 In other than the POSIX locale, other implementation-dependent subject sequence forms may be
 31407 accepted.

31408 If the subject sequence is empty or does not have the expected form, no conversion is performed;
 31409 the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null
 31410 pointer.

31411 The *wcstod()* function will not change the setting of **errno** if successful.

31412 Because 0 is returned on error and is also a valid return on success, an application wishing to
 31413 check for error situations should set *errno* to 0, then call *wcstod()*, then check *errno*.

31414 RETURN VALUE

31415 The *wcstod()* function returns the converted value, if any. If no conversion could be performed,
 31416 EX 0 is returned and *errno* may be set to [EINVAL].

31417 If the correct value is outside the range of representable values, ±HUGE_VAL is returned
 31418 (according to the sign of the value), and *errno* is set to [ERANGE] .

31419 If the correct value would cause underflow, 0 is returned and *errno* is set to [ERANGE] .

31420 **ERRORS**

31421 The *wcstod()* function will fail if:

31422 [ERANGE] The value to be returned would cause overflow or underflow.

31423 The *wcstod()* function may fail if:

31424 EX [EINVAL] No conversion could be performed.

31425 **EXAMPLES**

31426 None.

31427 **APPLICATION USAGE**

31428 None.

31429 **FUTURE DIRECTIONS**

31430 None.

31431 **SEE ALSO**

31432 *iswspace()*, *localeconv()*, *scanf()*, *setlocale()*, *wcstol()*, *<wchar.h>*, the XBD specification, **Chapter**
31433 **5, Locale**.

31434 **CHANGE HISTORY**

31435 First released in Issue 4.

31436 Derived from the MSE working draft.

31437 **Issue 5**

31438 The DESCRIPTION is updated to indicate that **errno** will not be changed if the function is
31439 successful.

31440 **NAME**

31441 wcstok — split wide-character string into tokens

31442 **SYNOPSIS**

31443 #include <wchar.h>

31444 wchar_t *wcstok(wchar_t *ws1, const wchar_t *ws2, wchar_t **ptr);

31445 **DESCRIPTION**

31446 A sequence of calls to *wcstok()* breaks the wide-character string pointed to by *ws1* into a
 31447 sequence of tokens, each of which is delimited by a wide-character code from the wide-character
 31448 string pointed to by *ws2*. The third argument points to a caller-provided **wchar_t** pointer into
 31449 which the *wcstok()* function stores information necessary for it to continue scanning the same
 31450 wide-character string.

31451 The first call in the sequence has *ws1* as its first argument, and is followed by calls with a null
 31452 pointer as their first argument. The separator string pointed to by *ws2* may be different from call
 31453 to call.

31454 The first call in the sequence searches the wide-character string pointed to by *ws1* for the first
 31455 wide-character code that is *not* contained in the current separator string pointed to by *ws2*. If no
 31456 such wide-character code is found, then there are no tokens in the wide-character string pointed
 31457 to by *ws1* and *wcstok()* returns a null pointer. If such a wide-character code is found, it is the
 31458 start of the first token.

31459 The *wcstok()* function then searches from there for a wide-character code that *is* contained in the
 31460 current separator string. If no such wide-character code is found, the current token extends to
 31461 the end of the wide-character string pointed to by *ws1*, and subsequent searches for a token will
 31462 return a null pointer. If such a wide-character code is found, it is overwritten by a null wide-
 31463 character, which terminates the current token. The *wcstok()* function saves a pointer to the
 31464 following wide-character code, from which the next search for a token will start.

31465 Each subsequent call, with a null pointer as the value of the first argument, starts searching from
 31466 the saved pointer and behaves as described above.

31467 The implementation will behave as if no function calls *wcstok()*.

31468 **RETURN VALUE**

31469 Upon successful completion, the *wcstok()* function returns a pointer to the first wide-character
 31470 code of a token. Otherwise, if there is no token, *wcstok()* returns a null pointer.

31471 **ERRORS**

31472 No errors are defined.

31473 **EXAMPLES**

31474 None.

31475 **APPLICATION USAGE**

31476 None.

31477 **FUTURE DIRECTIONS**

31478 None.

31479 **SEE ALSO**

31480 <wchar.h>.

31481 **CHANGE HISTORY**

31482 First released in Issue 4.

31483 **Issue 5**

31484 Aligned with ISO/IEC 9899:1990/Amendment 1:1994 (E). Specifically, a third argument is
31485 added to the definition of this function in the SYNOPSIS.

31486 **NAME**

31487 wcstol — convert a wide-character string to a long integer

31488 **SYNOPSIS**

31489 #include <wchar.h>

31490 long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);

31491 **DESCRIPTION**

31492 The *wcstol()* function converts the initial portion of the wide-character string pointed to by *nptr*
 31493 to **long int** representation. First it decomposes the input wide-character string into three parts:
 31494 an initial, possibly empty, sequence of white-space wide-character codes (as specified by
 31495 *iswspace()*), a subject sequence interpreted as an integer represented in some radix determined
 31496 by the value of *base*; and a final wide-character string of one or more unrecognised wide-
 31497 character codes, including the terminating null wide-character code of the input wide-character
 31498 string. Then it attempts to convert the subject sequence to an integer, and returns the result.

31499 If *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant
 31500 or hexadecimal constant, any of which may be preceded by a + or – sign. A decimal constant
 31501 begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant
 31502 consists of the prefix 0 optionally followed by a sequence of the digits 0 to 7 only. A
 31503 hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits
 31504 and letters a (or A) to f (or F) with values 10 to 15 respectively.

31505 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
 31506 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
 31507 by a + or – sign, but not including an integer suffix. The letters from a (or A) to z (or Z) inclusive
 31508 are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are
 31509 permitted. If the value of *base* is 16, the wide-character code representations of 0x or 0X may
 31510 optionally precede the sequence of letters and digits, following the sign if present.

31511 The subject sequence is defined as the longest initial subsequence of the input wide-character
 31512 string, starting with the first non-white-space wide-character code, that is of the expected form.
 31513 The subject sequence contains no wide-character codes if the input wide-character string is
 31514 empty or consists entirely of white-space wide-character code, or if the first non-white-space
 31515 wide-character code is other than a sign or a permissible letter or digit.

31516 If the subject sequence has the expected form and *base* is 0, the sequence of wide-character codes
 31517 starting with the first digit is interpreted as an integer constant. If the subject sequence has the
 31518 expected form and the value of *base* is between 2 and 36, it is used as the base for conversion,
 31519 ascribing to each letter its value as given above. If the subject sequence begins with a minus
 31520 sign, the value resulting from the conversion is negated. A pointer to the final wide-character
 31521 string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

31522 In other than the POSIX locale, additional implementation-dependent subject sequence forms
 31523 may be accepted.

31524 If the subject sequence is empty or does not have the expected form, no conversion is performed;
 31525 the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null
 31526 pointer.

31527 The *wcstol()* function will not change the setting of **errno** if successful.

31528 Because 0, {LONG_MIN} and {LONG_MAX} are returned on error and are also valid returns on
 31529 success, an application wishing to check for error situations should set *errno* to 0, then call
 31530 *wcstol()*, then check *errno*.

31531 **RETURN VALUE**

31532 Upon successful completion, *wcstol()* returns the converted value, if any. If no conversion could
 31533 be performed, 0 is returned and *errno* may be set to indicate the error. If the correct value is
 31534 outside the range of representable values, {LONG_MAX} or {LONG_MIN} is returned
 31535 (according to the sign of the value), and *errno* is set to [ERANGE] .

31536 **ERRORS**

31537 The *wcstol()* function will fail if:

31538 [EINVAL] The value of *base* is not supported.

31539 [ERANGE] The value to be returned is not representable.

31540 The *wcstol()* function may fail if:

31541 [EINVAL] No conversion could be performed.

31542 **EXAMPLES**

31543 None.

31544 **APPLICATION USAGE**

31545 None.

31546 **FUTURE DIRECTIONS**

31547 None.

31548 **SEE ALSO**

31549 *iswalph()*, *scanf()*, *wcstod()*, <wchar.h>.

31550 **CHANGE HISTORY**

31551 First released in Issue 4.

31552 Derived from the MSE working draft.

31553 **Issue 5**

31554 The DESCRIPTION is updated to indicate that **errno** will not be changed if the function is
 31555 successful.

31556 **NAME**

31557 wcstombs — convert a wide-character string to a character string

31558 **SYNOPSIS**

31559 #include <stdlib.h>

31560 size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);

31561 **DESCRIPTION**

31562 The *wcstombs()* function converts the sequence of wide-character codes that are in the array
 31563 pointed to by *pwcs* into a sequence of characters that begins in the initial shift state and stores
 31564 these characters into the array pointed to by *s*, stopping if a character would exceed the limit of *n*
 31565 total bytes or if a null byte is stored. Each wide-character code is converted as if by a call to
 31566 *wctomb()*, except that the shift state of *wctomb()* is not affected.

31567 The behaviour of this function is affected by the LC_CTYPE category of the current locale.

31568 No more than *n* bytes will be modified in the array pointed to by *s*. If copying takes place
 31569 EX between objects that overlap, the behaviour is undefined. If *s* is a null pointer, *wcstombs()* returns
 31570 the length required to convert the entire array regardless of the value of *n*, but no values are
 31571 stored. function returns the number of bytes required for the character array.

31572 **RETURN VALUE**

31573 If a wide-character code is encountered that does not correspond to a valid character (of one or
 31574 more bytes each), *wcstombs()* returns (size_t)−1. Otherwise, *wcstombs()* returns the number of
 31575 bytes stored in the character array, not including any terminating null byte. The array will not
 31576 be null-terminated if the value returned is *n*.

31577 **ERRORS**31578 The *wcstombs()* function may fail if:

31579 EX [EILSEQ] A wide-character code does not correspond to a valid character.

31580 **EXAMPLES**

31581 None.

31582 **APPLICATION USAGE**

31583 None.

31584 **FUTURE DIRECTIONS**

31585 None.

31586 **SEE ALSO**31587 *mblen()*, *mbtowc()*, *mbstowcs()*, *wctomb()*, <stdlib.h>.31588 **CHANGE HISTORY**

31589 First released in Issue 4.

31590 Derived from the ISO C standard.

31591 NAME

31592 wcstoul — convert a wide-character string to an unsigned long

31593 SYNOPSIS

31594 #include <wchar.h>

31595 unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr,
31596 int base);

31597 DESCRIPTION

31598 The *wcstoul()* function converts the initial portion of the wide-character string pointed to by *nptr*
31599 to **unsigned long int** representation. First it decomposes the input wide-character string into
31600 three parts: an initial, possibly empty, sequence of white-space wide-character codes (as
31601 specified by *iswspace()*); a subject sequence interpreted as an integer represented in some radix
31602 determined by the value of *base*; and a final wide-character string of one or more unrecognised
31603 wide-character codes, including the terminating null wide-character code of the input wide-
31604 character string. Then it attempts to convert the subject sequence to an unsigned integer, and
31605 returns the result.

31606 If *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant
31607 or hexadecimal constant, any of which may be preceded by a + or – sign. A decimal constant
31608 begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant
31609 consists of the prefix 0 optionally followed by a sequence of the digits 0 to 7 only. A
31610 hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits
31611 and letters a (or A) to f (or F) with values 10 to 15 respectively.

31612 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
31613 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
31614 by a + or – sign, but not including an integer suffix. The letters from a (or A) to z (or Z) inclusive
31615 are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are
31616 permitted. If the value of *base* is 16, the wide-character codes 0x or 0X may optionally precede
31617 the sequence of letters and digits, following the sign if present.

31618 The subject sequence is defined as the longest initial subsequence of the input wide-character
31619 string, starting with the first wide-character code that is not white space and is of the expected
31620 form. The subject sequence contains no wide-character codes if the input wide-character string
31621 is empty or consists entirely of white-space wide-character codes, or if the first wide-character
31622 code that is not white space is other than a sign or a permissible letter or digit.

31623 If the subject sequence has the expected form and *base* is 0, the sequence of wide-character codes
31624 starting with the first digit is interpreted as an integer constant. If the subject sequence has the
31625 expected form and the value of *base* is between 2 and 36, it is used as the base for conversion,
31626 ascribing to each letter its value as given above. If the subject sequence begins with a minus
31627 sign, the value resulting from the conversion is negated. A pointer to the final wide-character
31628 string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

31629 In other than the POSIX locale, additional implementation-dependent subject sequence forms
31630 may be accepted.

31631 If the subject sequence is empty or does not have the expected form, no conversion is performed;
31632 the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null
31633 pointer.

31634 The *wcstoul()* function will not change the setting of **errno** if successful.

31635 Because 0 and {ULONG_MAX} are returned on error and 0 is also a valid return on success, an
31636 application wishing to check for error situations should set *errno* to 0, then call *wcstoul()*, then
31637 check *errno*.

31638 **RETURN VALUE**

31639 Upon successful completion, *wcstoul()* returns the converted value, if any. If no conversion
 31640 could be performed, 0 is returned and *errno* may be set to indicate the error. If the correct value
 31641 is outside the range of representable values, {ULONG_MAX} is returned and *errno* is set to
 31642 [ERANGE].

31643 **ERRORS**

31644 The *wcstoul()* function will fail if:

31645 [EINVAL] The value of *base* is not supported.

31646 [ERANGE] The value to be returned is not representable.

31647 The *wcstoul()* function may fail if:

31648 [EINVAL] No conversion could be performed.

31649 **EXAMPLES**

31650 None.

31651 **APPLICATION USAGE**

31652 Unlike *wcstod()* and *wcstol()*, *wcstoul()* must always return a non-negative number; so, using the
 31653 return value of *wcstoul()* for out-of-range numbers with *wcstoul()* could cause more severe
 31654 problems than just loss of precision if those numbers can ever be negative.

31655 **FUTURE DIRECTIONS**

31656 None.

31657 **SEE ALSO**

31658 *iswalph()*, *scanf()*, *wcstod()*, *wcstol()*, <wchar.h>.

31659 **CHANGE HISTORY**

31660 First released in Issue 4.

31661 Derived from the MSE working draft.

31662 **Issue 5**

31663 The DESCRIPTION is updated to indicate that **errno** will not be changed if the function is
 31664 successful.

31665 **NAME**

31666 wcswcs — find a wide substring

31667 **SYNOPSIS**

31668 EX #include <wchar.h>

31669 wchar_t *wcswcs(const wchar_t *ws1, const wchar_t *ws2);

31670

31671 **DESCRIPTION**

31672 The *wcswcs()* function locates the first occurrence in the wide-character string pointed to by *ws1*
31673 of the sequence of wide-character codes (excluding the terminating null wide-character code) in
31674 the wide-character string pointed to by *ws2*.

31675 **RETURN VALUE**

31676 Upon successful completion, *wcswcs()* returns a pointer to the located wide-character string or a
31677 null pointer if the wide-character string is not found.

31678 If *ws2* points to a wide-character string with zero length, the function returns *ws1*.

31679 **ERRORS**

31680 No errors are defined.

31681 **EXAMPLES**

31682 None.

31683 **APPLICATION USAGE**

31684 This function was not included in the final ISO/IEC 9899:1990/Amendment 1:1994 (E).
31685 Application developers are strongly encouraged to use the *wcsstr()* function instead.

31686 **FUTURE DIRECTIONS**

31687 None.

31688 **SEE ALSO**31689 *wcschr()*, *wcsstr()*, <wchar.h>.31690 **CHANGE HISTORY**

31691 First released in Issue 4.

31692 Derived from the MSE working draft.

31693 **Issue 5**

31694 Marked EX.

31695 **NAME**31696 `wcswidth` — number of column positions of a wide-character string31697 **SYNOPSIS**31698 `#include <wchar.h>`31699 `int wcswidth(const wchar_t *pwcs, size_t n);`31700 **DESCRIPTION**

31701 The `wcswidth()` function determines the number of column positions required for *n* wide-
31702 character codes (or fewer than *n* wide-character codes if a null wide-character code is
31703 encountered before *n* wide-character codes are exhausted) in the string pointed to by *pwcs*.

31704 **RETURN VALUE**

31705 The `wcswidth()` function either returns 0 (if *pwcs* points to a null wide-character code), or returns
31706 the number of column positions to be occupied by the wide-character string pointed to by *pwcs*,
31707 or returns -1 (if any of the first *n* wide-character codes in the wide-character string pointed to by
31708 *pwcs* is not a printing wide-character code).

31709 **ERRORS**

31710 No errors are defined.

31711 **EXAMPLES**

31712 None.

31713 **APPLICATION USAGE**

31714 None.

31715 **FUTURE DIRECTIONS**

31716 None.

31717 **SEE ALSO**

31718 `wcwidth()`, `<wchar.h>`, the definition of **Column Position** in the XBD specification, **Chapter 2**,
31719 **Glossary**.

31720 **CHANGE HISTORY**

31721 First released in Issue 4.

31722 Derived from the MSE working draft.

31723 **NAME**

31724 wcsxfrm — wide-character string transformation

31725 **SYNOPSIS**

31726 #include <wchar.h>

31727 size_t wcsxfrm(wchar_t *ws1, const wchar_t *ws2, size_t n);

31728 **DESCRIPTION**

31729 The *wcsxfrm()* function transforms the wide-character string pointed to by *ws2* and places the
 31730 resulting wide-character string into the array pointed to by *ws1*. The transformation is such that
 31731 if *wscmp()* is applied to two transformed wide strings, it returns a value greater than, equal to
 31732 or less than 0, corresponding to the result of *wscoll()* applied to the same two original wide-
 31733 character strings. No more than *n* wide-character codes are placed into the resulting array
 31734 pointed to by *ws1*, including the terminating null wide-character code. If *n* is 0, *ws1* is permitted
 31735 to be a null pointer. If copying takes place between objects that overlap, the behaviour is
 31736 undefined.

31737 The *wcsxfrm()* function will not change the setting of **errno** if successful.31738 **RETURN VALUE**

31739 The *wcsxfrm()* function returns the length of the transformed wide-character string (not
 31740 including the terminating null wide-character code). If the value returned is *n* or more, the
 31741 contents of the array pointed to by *ws1* are indeterminate.

31742 On error, the *wcsxfrm()* function returns (**size_t**)−1, and sets *errno* to indicate the error.31743 **ERRORS**31744 The *wcsxfrm()* function may fail if:

31745 [EINVAL] The wide-character string pointed to by *ws2* contains wide-character codes
 31746 outside the domain of the collating sequence.

31747 **EXAMPLES**

31748 None.

31749 **APPLICATION USAGE**

31750 The transformation function is such that two transformed wide-character strings can be ordered
 31751 by *wscmp()* as appropriate to collating sequence information in the program's locale (category
 31752 LC_COLLATE).

31753 The fact that when *n* is 0, *ws1* is permitted to be a null pointer, is useful to determine the size of
 31754 the *ws1* array prior to making the transformation.

31755 Because no return value is reserved to indicate an error, an application wishing to check for error
 31756 situations should set *errno* to 0, then call *wcsxfrm()*, then check *errno*.

31757 **FUTURE DIRECTIONS**

31758 None.

31759 **SEE ALSO**31760 *wscmp()*, *wscoll()*, <wchar.h>.31761 **CHANGE HISTORY**

31762 First released in Issue 4.

31763 Derived from the MSE working draft.

31764 **Issue 5**

31765 Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

31766 The DESCRIPTION is updated to indicate that **errno** will not be changed if the function is
31767 successful.

31768 **NAME**

31769 wctob — wide-character to single-byte conversion

31770 **SYNOPSIS**

31771 #include <stdio.h>

31772 #include <wchar.h>

31773 int wctob(wint_t c);

31774 **DESCRIPTION**31775 The *wctob()* function determines whether *c* corresponds to a member of the extended character set whose character representation is a single byte when in the initial shift state.

31777 The behaviour of this function is affected by the LC_CTYPE category of the current locale.

31778 **RETURN VALUE**31779 The *wctob()* function returns EOF if *c* does not correspond to a character with length one in the initial shift state. Otherwise, it returns the single-byte representation of that character.31781 **ERRORS**

31782 No errors are defined.

31783 **EXAMPLES**

31784 None.

31785 **APPLICATION USAGE**

31786 None.

31787 **FUTURE DIRECTIONS**

31788 None.

31789 **SEE ALSO**31790 ***btowc()***, <wchar.h>.31791 **CHANGE HISTORY**

31792 First released in Issue 5.

31793 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

31794 **NAME**

31795 wctomb — convert a wide-character code to a character

31796 **SYNOPSIS**

31797 #include <stdlib.h>

31798 int wctomb(char *s, wchar_t wchar);

31799 **DESCRIPTION**

31800 The *wctomb()* function determines the number of bytes needed to represent the character
 31801 corresponding to the wide-character code whose value is *wchar* (including any change in the
 31802 shift state). It stores the character representation (possibly multiple bytes and any special bytes
 31803 to change shift state) in the array object pointed to by *s* (if *s* is not a null pointer). At most
 31804 {MB_CUR_MAX} bytes are stored. If *wchar* is 0, *wctomb()* is left in the initial shift state.

31805 The behaviour of this function is affected by the LC_CTYPE category of the current locale. For a
 31806 state-dependent encoding, this function is placed into its initial state by a call for which its
 31807 character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null
 31808 pointer cause the internal state of the function to be altered as necessary. A call with *s* as a null
 31809 pointer causes this function to return a non-zero value if encodings have state dependency, and
 31810 0 otherwise. Changing the LC_CTYPE category causes the shift state of this function to be
 31811 indeterminate.

31812 The implementation will behave as if no function defined in this document calls *wctomb()*.

31813 **RETURN VALUE**

31814 If *s* is a null pointer, *wctomb()* returns a non-zero or 0 value, if character encodings, respectively,
 31815 do or do not have state-dependent encodings. If *s* is not a null pointer, *wctomb()* returns -1 if the
 31816 value of *wchar* does not correspond to a valid character, or returns the number of bytes that
 31817 constitute the character corresponding to the value of *wchar*.

31818 In no case will the value returned be greater than the value of the MB_CUR_MAX macro.

31819 **ERRORS**

31820 No errors are defined.

31821 **EXAMPLES**

31822 None.

31823 **APPLICATION USAGE**

31824 None.

31825 **FUTURE DIRECTIONS**

31826 None.

31827 **SEE ALSO**31828 *mblen()*, *mbtowc()*, *mbstowcs()*, *wcstombs()*, <stdlib.h>.31829 **CHANGE HISTORY**

31830 First released in Issue 4.

31831 Derived from the ANSI C standard.

31832 NAME

31833 wctrans — define character mapping

31834 SYNOPSIS

31835 #include <wctype.h>

31836 wctrans_t wctrans(const char *charclass);

31837 DESCRIPTION

31838 The *wctrans()* function is defined for valid character mapping names identified in the current
31839 locale. The *charclass* is a string identifying a generic character mapping name for which codeset-
31840 specific information is required. The following character mapping names are defined in all
31841 locales — "tolower" and "toupper".

31842 The function returns a value of type **wctrans_t**, which can be used as the second argument to
31843 subsequent calls of *towctrans()*. The *wctrans()* function determines values of **wctrans_t**
31844 according to the rules of the coded character set defined by character mapping information in
31845 the program's locale (category LC_CTYPE). The values returned by *wctrans()* are valid until a
31846 call to *setlocale()* that modifies the category LC_CTYPE.

31847 RETURN VALUE

31848 The *wctrans()* function returns 0 if the given character mapping name is not valid for the current
31849 locale (category LC_CTYPE), otherwise it returns a non-zero object of type **wctrans_t** that can be
31850 used in calls to *towctrans()*.

31851 ERRORS

31852 The *wctrans()* function may fail if:

31853 [EINVAL] The character mapping name pointed to by *charclass* is not valid in the current
31854 locale.

31855 EXAMPLES

31856 None.

31857 APPLICATION USAGE

31858 None.

31859 FUTURE DIRECTIONS

31860 None.

31861 SEE ALSO

31862 *towctrans()*, <wctype.h>.

31863 CHANGE HISTORY

31864 First released in Issue 5.

31865 Derived from ISO/IEC 9899:1990/Amendment 1:1994 (E).

31866 **NAME**

31867 wctype — define character class

31868 **SYNOPSIS**

31869 #include <wctype.h>

31870 wctype_t wctype(const char *property);

31871 **DESCRIPTION**

31872 The *wctype()* function is defined for valid character class names as defined in the current locale.
 31873 The *property* is a string identifying a generic character class for which codeset-specific type
 31874 information is required. The following character class names are defined in all locales —
 31875 "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper" and
 31876 "xdigit".

31877 Additional character class names defined in the locale definition file (category LC_CTYPE) can
 31878 also be specified.

31879 The function returns a value of type **wctype_t**, which can be used as the second argument to
 31880 subsequent calls of *iswctype()*. The *wctype()* function determines values of **wctype_t** according
 31881 to the rules of the coded character set defined by character type information in the program's
 31882 locale (category LC_CTYPE). The values returned by *wctype()* are valid until a call to *setlocale()*
 31883 that modifies the category LC_CTYPE.

31884 **RETURN VALUE**

31885 The *wctype()* function returns 0 if the given character class name is not valid for the current
 31886 locale (category LC_CTYPE), otherwise it returns an object of type **wctype_t** that can be used in
 31887 calls to *iswctype()*.

31888 **ERRORS**

31889 No errors are defined.

31890 **EXAMPLES**

31891 None.

31892 **APPLICATION USAGE**

31893 None.

31894 **FUTURE DIRECTIONS**

31895 None.

31896 **SEE ALSO**31897 *iswctype()*, <wctype.h>, <wchar.h>.31898 **CHANGE HISTORY**

31899 First released in Issue 4.

31900 **Issue 5**

31901 The following change has been made in this issue for alignment with ISO/IEC
 31902 9899:1990/Amendment 1:1994 (E).

- 31903 • The SYNOPSIS has been changed to indicate that this function and associated data types are
 31904 now made visible by inclusion of the header <**wctype.h**> rather than <**wchar.h**>.

31905 NAME

31906 wcwidth — number of column positions of a wide-character code

31907 SYNOPSIS

31908 #include <wchar.h>

31909 int wcwidth(wchar_t *wc*);

31910 DESCRIPTION

31911 The *wcwidth()* function determines the number of column positions required for the wide
31912 character *wc*. The value of *wc* must be a character representable as a **wchar_t**, and must be a
31913 wide-character code corresponding to a valid character in the current locale.

31914 RETURN VALUE

31915 The *wcwidth()* function either returns 0 (if *wc* is a null wide-character code), or returns the
31916 number of column positions to be occupied by the wide-character code *wc*, or returns -1 (if *wc*
31917 does not correspond to a printing wide-character code).

31918 ERRORS

31919 No errors are defined.

31920 EXAMPLES

31921 None.

31922 APPLICATION USAGE

31923 None.

31924 FUTURE DIRECTIONS

31925 None.

31926 SEE ALSO

31927 *wcswidth()*, <wchar.h>.

31928 CHANGE HISTORY

31929 First released as a World-wide Portability Interface in Issue 4.

31930 Derived from MSE working draft.

31931 **NAME**

31932 wmemchr — find a wide-character in memory

31933 **SYNOPSIS**

31934 #include <wchar.h>

31935 wchar_t *wmemchr(const wchar_t *ws, wchar_t wc, size_t n);

31936 **DESCRIPTION**

31937 The *wmemchr()* function locates the first occurrence of *wc* in the initial *n* wide-characters of the
31938 object pointed to be *ws*. This function is not affected by locale and all **wchar_t** values are treated
31939 identically. The null wide-character and **wchar_t** values not corresponding to valid characters
31940 are not treated specially.

31941 If *n* is zero, *ws* must be a valid pointer and the function behaves as if no valid occurrence of *wc* is
31942 found.

31943 **RETURN VALUE**

31944 The *wmemchr()* function returns a pointer to the located wide-character, or a null pointer if the
31945 wide-character does not occur in the object.

31946 **ERRORS**

31947 No errors are defined.

31948 **EXAMPLES**

31949 None.

31950 **APPLICATION USAGE**

31951 None.

31952 **FUTURE DIRECTIONS**

31953 None.

31954 **SEE ALSO**31955 <wchar.h>, *wmemcmp()*, *wmemcpy()*, *wmemmove()*, *wmemset()*.31956 **CHANGE HISTORY**

31957 First released in Issue 5.

31958 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

31959 NAME

31960 wmemcmp — compare wide-characters in memory

31961 SYNOPSIS

31962 #include <wchar.h>

31963 int wmemcmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);

31964 DESCRIPTION

31965 The *wmemcmp()* function compares the first *n* wide-characters of the object pointed to by *ws1* to
31966 the first *n* wide-characters of the object pointed to by *ws2*. This function is not affected by locale
31967 and all **wchar_t** values are treated identically. The null wide-character and **wchar_t** values not
31968 corresponding to valid characters are not treated specially.

31969 If *n* is zero, *ws1* and *ws2* must be a valid pointers and the function behaves as if the two objects
31970 compare equal.

31971 RETURN VALUE

31972 The *wmemcmp()* function returns an integer greater than, equal to, or less than zero, accordingly
31973 as the object pointed to by *ws1* is greater than, equal to, or less than the object pointed to by *ws2*.

31974 ERRORS

31975 No errors are defined.

31976 EXAMPLES

31977 None.

31978 APPLICATION USAGE

31979 None.

31980 FUTURE DIRECTIONS

31981 None.

31982 SEE ALSO

31983 wchar.hwmemchr(), *wmemcpy()*, *wmemmove()*, *wmemset()*.

31984 CHANGE HISTORY

31985 First released in Issue 5.

31986 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

31987 **NAME**

31988 wmemcpy — copy wide-characters in memory

31989 **SYNOPSIS**

31990 #include <wchar.h>

31991 wchar_t *wmemcpy(wchar_t *ws1, const wchar_t *ws2, size_t n);

31992 **DESCRIPTION**

31993 The *wmemcpy()* function copies *n* wide-characters from the object pointed to by *ws2* to the object
31994 pointed to by *ws1*. This function is not affected by locale and all **wchar_t** values are treated
31995 identically. The null wide-character and **wchar_t** values not corresponding to valid characters
31996 are not treated specially.

31997 If *n* is zero, *ws1* and *ws2* must be valid pointers, and the function copies zero wide-characters.

31998 **RETURN VALUE**31999 The *wmemcpy()* function returns the value of *ws1*.32000 **ERRORS**

32001 No errors are defined.

32002 **EXAMPLES**

32003 None.

32004 **APPLICATION USAGE**

32005 None.

32006 **FUTURE DIRECTIONS**

32007 None.

32008 **SEE ALSO**32009 <wchar.h>, *wmemchr()*, *wmemcmp()*, *wmemmove()*, *wmemset()*.32010 **CHANGE HISTORY**

32011 First released in Issue 5.

32012 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

32013 NAME

32014 **wmemmove** — copy wide-characters in memory with overlapping areas

32015 SYNOPSIS

32016 `#include <wchar.h>`

32017 `wchar_t *wmemmove(wchar_t *ws1, const wchar_t *ws2, size_t n);`

32018 DESCRIPTION

32019 The *wmemmove()* function copies *n* wide-characters from the object pointed to by *ws2* to the
32020 object pointed to by *ws1*. Copying takes place as if the *n* wide-characters from the object pointed
32021 to by *ws2* are first copied into a temporary array of *n* wide-characters that does not overlap the
32022 objects pointed to by *ws1* or *ws2*, and then the *n* wide-characters from the temporary array are
32023 copied into the object pointed to by *ws1*.

32024 This function is not affected by locale and all **wchar_t** values are treated identically. The null
32025 wide-character and **wchar_t** values not corresponding to valid characters are not treated
32026 specially.

32027 If *n* is zero, *ws1* and *ws2* must be a valid pointers, and the function copies zero wide-characters.

32028 RETURN VALUE

32029 The *wmemmove* function returns the value of *ws1*.

32030 ERRORS

32031 No errors are defined

32032 EXAMPLES

32033 None.

32034 APPLICATION USAGE

32035 None.

32036 FUTURE DIRECTIONS

32037 None.

32038 SEE ALSO

32039 `<wchar.h>`, *wmemchr()*, *wmemcmp()*, *wmemcpy()*, *wmemset()*.

32040 CHANGE HISTORY

32041 First released in Issue 5.

32042 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

32043 **NAME**

32044 wmemset — set wide-characters in memory

32045 **SYNOPSIS**

32046 #include <wchar.h>

32047 wchar_t *wmemset(wchar_t *ws, wchar_t wc, size_t n);

32048 **DESCRIPTION**

32049 The *wmemset()* function copies the value of *wc* into each of the first *n* wide-characters of the
32050 object pointed to by *ws*. This function is not affected by locale and all **wchar_t** values are treated
32051 identically. The null wide-character and **wchar_t** values not corresponding to valid characters
32052 are not treated specially.

32053 If *n* is zero, *ws* must be a valid pointer and the function copies zero wide-characters.

32054 **RETURN VALUE**32055 The *wmemset()* functions returns the value of *ws*.32056 **ERRORS**

32057 No errors are defined.

32058 **EXAMPLES**

32059 None.

32060 **APPLICATION USAGE**

32061 None.

32062 **FUTURE DIRECTIONS**

32063 None.

32064 **SEE ALSO**32065 <wchar.h>, *wmemchr()*, *wmemcmp()*, *wmemcpy()*, *wmemmove()*.32066 **CHANGE HISTORY**

32067 First released in Issue 5.

32068 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

32069 **NAME**

32070 wordexp, wordfree — perform word expansions

32071 **SYNOPSIS**

32072 #include <wordexp.h>

32073 int wordexp(const char *words, wordexp_t *pwordexp, int flags);

32074 void wordfree(wordexp_t *pwordexp);

32075 **DESCRIPTION**

32076 The *wordexp()* function performs word expansions as described in the **XCU** specification, **Section 2.6, Word Expansions**, subject to quoting as in the **XCU** specification, **Section 2.2, Quoting**, and places the list of expanded words into the structure pointed to by *pwordexp*.

32079 The *words* argument is a pointer to a string containing one or more words to be expanded. The expansions will be the same as would be performed by the shell if *words* were the part of a command line representing the arguments to a utility. Therefore, *words* must not contain an unquoted newline or any of the unquoted shell special characters:

32083 | & ; < >

32084 except in the context of command substitution as specified in the **XCU** specification, **Section 2.6.3, Command Substitution**. It also must 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()* may treat the comment character as a regular character, or may interpret it as a comment indicator and ignore the remainder of *words*.

32090 The structure type **wordexp_t** is defined in the header **<wordexp.h>** and includes at least the following members:

32092

32093

Member Type	Member Name	Description
size_t	we_wordc	Count of words matched by <i>words</i> .
char **	we_wordv	Pointer to list of expanded words.
size_t	we_offs	Slots to reserve at the beginning of <i>pwordexp->we_wordv</i> .

32094

32095

32096

32097 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*. Each individual field created during field splitting (see the **XCU** specification, **Section 2.6.5, Field Splitting**) or pathname expansion (see the **XCU** specification, **Section 2.6.6, Pathname Expansion**) is a separate word in the *pwordexp->we_wordv* list. The words are in order as described in the **XCU** specification, **Section 2.6, Word Expansions**. The first pointer after the last word pointer will be a null pointer. The expansion of special parameters described in the **XCU** specification, **Section 2.5.2, Special Parameters** is unspecified.

32105 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()*.

32109 The *flags* argument is used to control the behaviour 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>**:

32112 WRDE_APPEND Append words generated to the ones from a previous call to *wordexp()*.

32113	WRDE_DOOFFS	Make use of <i>pwordexp->we_offs</i> . If this flag is set, <i>pwordexp->we_offs</i> is used to specify how many null pointers to add to the beginning of <i>pwordexp->we_wordv</i> . In other words, <i>pwordexp->we_wordv</i> will point to <i>pwordexp->we_offs</i> null pointers, followed by <i>pwordexp->we_wordc</i> word pointers, followed by a null pointer.
32114		
32115		
32116		
32117		
32118	WRDE_NOCMD	Fail if command substitution, as specified in the XCU specification, Section 2.6.3, Command Substitution , is requested.
32119		
32120	WRDE_REUSE	The <i>pwordexp</i> argument was passed to a previous successful call to <i>wordexp()</i> , and has not been passed to <i>wordfree()</i> . The result will be the same as if the application had called <i>wordfree()</i> and then called <i>wordexp()</i> without WRDE_REUSE.
32121		
32122		
32123		
32124	WRDE_SHOWERR	Do not redirect <i>stderr</i> to /dev/null .
32125	WRDE_UNDEF	Report error on an attempt to expand an undefined shell variable.
32126		The WRDE_APPEND flag can be used to append a new set of words to those generated by a previous call to <i>wordexp()</i> . The following rules apply when two or more calls to <i>wordexp()</i> are made with the same value of <i>pwordexp</i> and without intervening calls to <i>wordfree()</i> :
32127		
32128		
32129		1. The first such call must not set WRDE_APPEND. All subsequent calls must set it.
32130		2. All of the calls must set WRDE_DOOFFS, or all must not set it.
32131		3. After the second and each subsequent call, <i>pwordexp->we_wordv</i> will point to a list containing the following:
32132		
32133		a. zero or more null pointers, as specified by WRDE_DOOFFS and <i>pwordexp->we_offs</i>
32134		b. pointers to the words that were in the <i>pwordexp->we_wordv</i> list before the call, in the same order as before
32135		
32136		c. pointers to the new words generated by the latest call, in the specified order
32137		4. The count returned in <i>pwordexp->we_wordc</i> will be the total number of words from all of the calls.
32138		
32139		5. The application can change any of the fields after a call to <i>wordexp()</i> , but if it does it must reset them to the original value before a subsequent call, using the same <i>pwordexp</i> value, to <i>wordfree()</i> or <i>wordexp()</i> with the WRDE_APPEND or WRDE_REUSE flag.
32140		
32141		
32142		If <i>words</i> contains an unquoted:
32143		<newline> & ; < > () { }
32144		in an inappropriate context, <i>wordexp()</i> will fail, and the number of expanded words will be 0.
32145		Unless WRDE_SHOWERR is set in <i>flags</i> , <i>wordexp()</i> will redirect <i>stderr</i> to /dev/null for any utilities executed as a result of command substitution while expanding <i>words</i> . If WRDE_SHOWERR is set, <i>wordexp()</i> may write messages to <i>stderr</i> if syntax errors are detected while expanding <i>words</i> .
32146		
32147		
32148		
32149		If WRDE_DOOFFS is set, then <i>pwordexp->we_offs</i> must have the same value for each <i>wordexp()</i> call and <i>wordfree()</i> call using a given <i>pwordexp</i> .
32150		
32151		The following constants are defined as error return values:
32152	WRDE_BADCHAR	One of the unquoted characters:
32153		<newline> & ; < > () { }

32154 appears in *words* in an inappropriate context.

32155 WRDE_BADVAL Reference to undefined shell variable when WRDE_UNDEF is set in *flags*.

32156 WRDE_CMDSUB Command substitution requested when WRDE_NOCMD was set in *flags*.

32157 WRDE_NOSPACE Attempt to allocate memory failed.

32158 WRDE_SYNTAX Shell syntax error, such as unbalanced parentheses or unterminated

32159 string.

32160 RETURN VALUE

32161 On successful completion, *wordexp()* returns 0.

32162 Otherwise, a non-zero value as described in <**wordexp.h**> is returned to indicate an error. If

32163 *wordexp()* returns the value WRDE_NOSPACE, then *pwordexp->we_wordc* and

32164 *pwordexp->we_wordv* will be updated to reflect any words that were successfully expanded. In

32165 other cases, they will not be modified.

32166 The *wordfree()* function returns no value.

32167 ERRORS

32168 No errors are defined.

32169 EXAMPLES

32170 None.

32171 APPLICATION USAGE

32172 This function is intended to be used by an application that wants to do all of the shell's

32173 expansions on a word or words obtained from a user. For example, if the application prompts

32174 for a filename (or list of filenames) and then uses *wordexp()* to process the input, the user could

32175 respond with anything that would be valid as input to the shell.

32176 The WRDE_NOCMD flag is provided for applications that, for security or other reasons, want to

32177 prevent a user from executing shell commands. Disallowing unquoted shell special characters

32178 also prevents unwanted side effects such as executing a command or writing a file.

32179 FUTURE DIRECTIONS

32180 None.

32181 SEE ALSO

32182 *fnmatch()*, *glob()*, <**wordexp.h**>, the XCU specification.

32183 CHANGE HISTORY

32184 First released in Issue 4.

32185 Derived from the ISO POSIX-2 standard.

32186 Issue 5

32187 Moved from POSIX2 C-language Binding to BASE.

32188 **NAME**

32189 wprintf — print formatted wide-character output

32190 **SYNOPSIS**

32191 #include <stdio.h>

32192 #include <wchar.h>

32193 int wprintf(const wchar_t **format*, ...);32194 **DESCRIPTION**32195 Refer to *fwprintf()*.32196 **CHANGE HISTORY**

32197 First released in Issue 5.

32198 Include for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

32199 NAME

32200 write, writev, pwrite — write on a file

32201 SYNOPSIS

32202 #include <unistd.h>

32203 ssize_t write(int *fildes*, const void **buf*, size_t *nbyte*);32204 EX ssize_t pwrite(int *fildes*, const void **buf*, size_t *nbyte*,32205 off_t *offset*);

32206 #include <sys/uio.h>

32207 ssize_t writev(int *fildes*, const struct iovec **iov*, int *iovcnt*);

32208

32209 DESCRIPTION

32210 The *write()* function attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file
32211 associated with the open file descriptor, *fildes*.32212 If *nbyte* is 0, *write()* will return 0 and have no other results if the file is a regular file; otherwise,
32213 the results are unspecified.32214 On a regular file or other file capable of seeking, the actual writing of data proceeds from the
32215 position in the file indicated by the file offset associated with *fildes*. Before successful return
32216 from *write()*, the file offset is incremented by the number of bytes actually written. On a regular
32217 file, if this incremented file offset is greater than the length of the file, the length of the file will be
32218 set to this file offset.32219 On a file not capable of seeking, writing always takes place starting at the current position. The
32220 value of a file offset associated with such a device is undefined.32221 If the O_APPEND flag of the file status flags is set, the file offset will be set to the end of the file
32222 prior to each write and no intervening file modification operation will occur between changing
32223 the file offset and the write operation.32224 EX If a *write()* requests that more bytes be written than there is room for (for example, the *ulimit* or
32225 the physical end of a medium), only as many bytes as there is room for will be written. For
32226 example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512
32227 bytes will return 20. The next write of a non-zero number of bytes will give a failure return
32228 EX (except as noted below) and the implementation will generate a SIGXFSZ signal for the thread.32229 If *write()* is interrupted by a signal before it writes any data, it will return -1 with *errno* set to
32230 [EINTR].32231 FIPS If *write()* is interrupted by a signal after it successfully writes some data, it will return the
32232 number of bytes written.32233 If the value of *nbyte* is greater than {SSIZE_MAX}, the result is implementation-dependent.32234 After a *write()* to a regular file has successfully returned:32235 • Any successful *read()* from each byte position in the file that was modified by that write will
32236 return the data specified by the *write()* for that position until such byte positions are again
32237 modified.32238 • Any subsequent successful *write()* to the same byte position in the file will overwrite that file
32239 data.32240 Write requests to a pipe or FIFO will be handled the same as a regular file with the following
32241 exceptions:

32242		
32243	•	There is no file offset associated with a pipe, hence each write request will append to the end of the pipe.
32244	•	Write requests of {PIPE_BUF} bytes or less will 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.
32245		
32246		
32247		
32248	•	If the O_NONBLOCK flag is clear, a write request may cause the thread to block, but on normal completion it will return <i>nbyte</i> .
32249		
32250	•	If the O_NONBLOCK flag is set, <i>write()</i> requests will be handled differently, in the following ways:
32251		
32252	—	The <i>write()</i> function will not block the thread.
32253	—	A write request for {PIPE_BUF} or fewer bytes will have the following effect: If there is sufficient space available in the pipe, <i>write()</i> will transfer all the data and return the number of bytes requested. Otherwise, <i>write()</i> will transfer no data and return <i>-1</i> with <i>errno</i> set to [EAGAIN].
32254		
32255		
32256		
32257	—	A write request for more than {PIPE_BUF} bytes will case one of the following:
32258	a.	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 will transfer at least {PIPE_BUF} bytes.
32259		
32260		
32261	b.	When no data can be written, transfer no data and return <i>-1</i> with <i>errno</i> set to [EAGAIN].
32262		
32263		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:
32264		
32265	•	If the O_NONBLOCK flag is clear, <i>write()</i> will block the calling thread until the data can be accepted.
32266		
32267	•	If the O_NONBLOCK flag is set, <i>write()</i> will not block the process. If some data can be written without blocking the process, <i>write()</i> will write what it can and return the number of bytes written. Otherwise, it will return <i>-1</i> and <i>errno</i> will be set to [EAGAIN].
32268		
32269		
32270		Upon successful completion, where <i>nbyte</i> is greater than 0, <i>write()</i> will mark for update the <i>st_ctime</i> and <i>st_mtime</i> 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.
32271		
32272		
32273	EX	If <i>fildev</i> refers to a STREAM, the operation of <i>write()</i> is determined by the values of the minimum and maximum <i>nbyte</i> range ("packet size") accepted by the STREAM. These values are determined by the topmost STREAM module. If <i>nbyte</i> falls within the packet size range, <i>nbyte</i> bytes will be written. If <i>nbyte</i> does not fall within the range and the minimum packet size value is 0, <i>write()</i> will 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 <i>nbyte</i> does not fall within the range and the minimum value is non-zero, <i>write()</i> will fail with <i>errno</i> set to [ERANGE]. Writing a zero-length buffer (<i>nbyte</i> 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>I_SWROPT ioctl()</i> to enable zero-length messages to be sent across the pipe or FIFO.
32274		
32275		
32276		
32277		
32278		
32279		
32280		
32281		
32282		
32283		
32284		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:
32285		

32286		• If O_NONBLOCK is clear, and the STREAM cannot accept data (the STREAM write queue is full due to internal flow control conditions), <i>write()</i> will block until data can be accepted.
32287		
32288		• If O_NONBLOCK is set and the STREAM cannot accept data, <i>write()</i> will return -1 and set <i>errno</i> to [EAGAIN].
32289		
32290		• 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, <i>write()</i> will terminate and return the number of bytes written.
32291		
32292		
32293		In addition, <i>write()</i> and <i>writen()</i> will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of <i>errno</i> does not reflect the result of <i>write()</i> or <i>writen()</i> but reflects the prior error.
32294		
32295		
32296		The <i>writen()</i> function is equivalent to <i>write()</i> , but gathers the output data from the <i>iovcnt</i> buffers specified by the members of the <i>iov</i> array: <i>iov</i> [0], <i>iov</i> [1], ..., <i>iov</i> [<i>iovcnt</i> - 1]. <i>iovcnt</i> is valid if greater than 0 and less than or equal to {IOV_MAX}, defined in <limits.h>.
32297		
32298		
32299		Each <i>iovec</i> entry specifies the base address and length of an area in memory from which data should be written. The <i>writen()</i> function will always write a complete area before proceeding to the next.
32300		
32301		
32302		If <i>fildev</i> refers to a regular file and all of the <i>iov_len</i> members in the array pointed to by <i>iov</i> are 0, <i>writen()</i> will return 0 and have no other effect. For other file types, the behaviour is unspecified.
32303		
32304		If the sum of the <i>iov_len</i> values is greater than SSIZE_MAX, the operation fails and no data is transferred.
32305		
32306	RT	If the Synchronized Input and Output option is supported:
32307		If the O_DSYNC bit has been set, write I/O operations on the file descriptor complete as defined by synchronised I/O data integrity completion.
32308		
32309		If the O_SYNC bit has been set, write I/O operations on the file descriptor complete as defined by synchronised I/O file integrity completion.
32310		
32311	RT	If the Shared Memory Objects option is supported:
32312		If <i>fildev</i> refers to a shared memory object, the result of the <i>write()</i> function is unspecified.
32313	EX	For regular files, no data transfer will occur past the offset maximum established in the open file description associated with <i>fildev</i> .
32314		
32315		The <i>pwrite()</i> function performs the same action as <i>write()</i> , except that it writes into a given position without changing the file pointer. The first three arguments to <i>pwrite()</i> are the same as <i>write()</i> with the addition of a fourth argument offset for the desired position inside the file.
32316		
32317		
32318		RETURN VALUE
32319	EX	Upon successful completion, <i>write()</i> and <i>pwrite()</i> will return the number of bytes actually written to the file associated with <i>fildev</i> . This number will never be greater than <i>nbyte</i> . Otherwise, -1 is returned and <i>errno</i> is set to indicate the error.
32320		
32321		
32322	EX	Upon successful completion, <i>writen()</i> returns the number of bytes actually written. Otherwise, it returns a value of -1, the file-pointer remains unchanged, and <i>errno</i> is set to indicate an error.
32323		
32324		ERRORS
32325	EX	The <i>write()</i> , <i>writen()</i> and <i>pwrite()</i> functions will fail if:
32326		[EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the thread would be delayed in the <i>write()</i> operation.
32327		

32328	[EBADF]	The <i>fdes</i> argument is not a valid file descriptor open for writing.
32329	[EFBIG]	An attempt was made to write a file that exceeds the implementation-dependent maximum file size or the process' file size limit.
32330 EX		
32331 EX	[EFBIG]	The file is a regular file, <i>nbyte</i> is greater than 0 and the starting position is greater than or equal to the offset maximum established in the open file description associated with <i>fdes</i> .
32332		
32333		
32334	[EINTR]	The write operation was terminated due to the receipt of a signal, and no data was transferred.
32335		
32336 EX	[EIO]	A physical I/O error has occurred.
32337	[EIO]	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-dependent conditions.
32338		
32339		
32340		
32341	[ENOSPC]	There was no free space remaining on the device containing the file.
32342	[EPIPE]	An attempt is made to write to a pipe or FIFO that is not open for reading by any process, or that only has one end open. A SIGPIPE signal will also be sent to the thread.
32343 EX		
32344		
32345 EX	[ERANGE]	The transfer request size was outside the range supported by the STREAMS file associated with <i>fdes</i> .
32346		
32347		The <i>writv()</i> function will fail if:
32348	[EINVAL]	The sum of the <i>iov_len</i> values in the <i>iov</i> array would overflow an <i>ssize_t</i> .
32349 EX		The <i>write()</i> , <i>writv()</i> and <i>pwrite()</i> functions may fail if:
32350 EX	[EINVAL]	The STREAM or multiplexer referenced by <i>fdes</i> is linked (directly or indirectly) downstream from a multiplexer.
32351		
32352 EX	[ENXIO]	A request was made of a non-existent device, or the request was outside the capabilities of the device.
32353		
32354 EX	[ENXIO]	A hangup occurred on the STREAM being written to.
32355 EX		A write to a STREAMS file may fail if an error message has been received at the STREAM head. In this case, <i>errno</i> is set to the value included in the error message.
32356		
32357		The <i>writv()</i> function may fail and set <i>errno</i> to:
32358	[EINVAL]	The <i>iovcnt</i> argument was less than or equal to 0, or greater than {IOV_MAX}.
32359		The <i>pwrite()</i> function fails and the file pointer remains unchanged if:
32360	[EINVAL]	The <i>offset</i> argument is invalid. The value is negative.
32361	[ESPIPE]	<i>fdes</i> is associated with a pipe or FIFO.
32362		
32363	EXAMPLES	
32364	None.	
32365	APPLICATION USAGE	
32366	None.	

32367 **FUTURE DIRECTIONS**

32368 None.

32369 **SEE ALSO**

32370 *chmod()*, *creat()*, *dup()*, *fcntl()*, *getrlimit()*, *lseek()*, *open()*, *pipe()*, *ulimit()*, **<limits.h>**,
 32371 **<stropts.h>**, **<sys/uio.h>**, **<unistd.h>**.

32372 **CHANGE HISTORY**

32373 First released in Issue 1.

32374 Derived from Issue 1 of the SVID.

32375 **Issue 4**

32376 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- 32377 • The type of the argument *buf* is changed from **char *** to **const void***, and the type of the
 32378 argument *nbyte* is changed from **unsigned** to **size_t**.
- 32379 • The DESCRIPTION is changed:
 - 32380 — to indicate that writing at end-of-file is atomic
 - 32381 — to identify that {SSIZE_MAX} is now used to determine the maximum value of *nbyte*
 - 32382 — to indicate the consequences of activities after a call to the *write()* function
 - 32383 — To improve clarity, the text describing operations on pipes or FIFOs when
 32384 O_NONBLOCK is set is restructured.

32385 Other changes are incorporated as follows:

- 32386 • The **<unistd.h>** header is added to the SYNOPSIS section.
- 32387 • Reference to *ulimit* in the DESCRIPTION is marked as an extension.
- 32388 • Reference to the process' file size limit and the *ulimit()* function are marked as extensions in
 32389 the description of the [EFBIG] error.
- 32390 • The [ENXIO] error is marked as an extension.
- 32391 • The APPLICATION USAGE section is removed.
- 32392 • The description of [EINTR] is amended.

32393 **Issue 4, Version 2**

32394 The following changes are incorporated for X/OPEN UNIX conformance:

- 32395 • The *writew()* function is added to the SYNOPSIS.
- 32396 • The DESCRIPTION is updated to describe the writing of data to STREAMS files, an
 32397 operational description of the *writew()* function is included, and a statement is added
 32398 indicating that SIGXFSZ will be generated if an attempted write operation would cause the
 32399 maximum file size to be exceeded.
- 32400 • The RETURN VALUE section is updated to describe values returned by the *writew()* function.
- 32401 • The ERRORS section has been restructured to describe errors that apply to both *write()* and
 32402 *writew()* apart from those that apply to *writew()* specifically. The [EIO], [ERANGE] and
 32403 [EINVAL] errors are also added.

32404 **Issue 5**

32405 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
32406 Threads Extension.

32407 Large File Summit extensions added.

32408 The *pwrite()* function is added.

32409 **NAME**

32410 wscanf — convert formatted wide-character input

32411 **SYNOPSIS**

32412 #include <stdio.h>

32413 #include <wchar.h>

32414 int wscanf(const wchar_t **format*, ...);32415 **DESCRIPTION**32416 Refer to *fwscanf()*.32417 **CHANGE HISTORY**

32418 First released in Issue 5.

32419 Included for alignment with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

32420 **NAME**

32421 y0, y1, yn — Bessel functions of the second kind

32422 **SYNOPSIS**32423 EX `#include <math.h>`32424 `double y0(double x);`32425 `double y1 (double x);`32426 `double yn (int n, double x);`

32427

32428 **DESCRIPTION**32429 The *y0()*, *y1()* and *yn()* functions compute Bessel functions of *x* of the second kind of orders 0, 1
32430 and *n* respectively. The value of *x* must be positive.32431 An application wishing to check for error situations should set *errno* to 0 before calling *y0()*, *y1()*
32432 or *yn()*. If *errno* is non-zero on return, or the return value is NaN, an error has occurred.32433 **RETURN VALUE**32434 Upon successful completion, *y0()*, *y1()* and *yn()* will return the relevant Bessel value of *x* of the
32435 second kind.32436 If *x* is NaN, NaN is returned and *errno* may be set to [EDOM].32437 If the *x* argument to *y0()*, *y1()* or *yn()* is negative, `-HUGE_VAL` or NaN is returned, and *errno*
32438 may be set to [EDOM].32439 If *x* is 0.0, `-HUGE_VAL` is returned and *errno* may be set to [ERANGE] or [EDOM].32440 If the correct result would cause underflow, 0.0 is returned and *errno* may be set to [ERANGE].32441 If the correct result would cause overflow, `-HUGE_VAL` or 0.0 is returned and *errno* may be set
32442 to [ERANGE].32443 **ERRORS**32444 The *y0()*, *y1()* and *yn()* functions may fail if:32445 [EDOM] The value of *x* is negative or NaN.32446 [ERANGE] The value of *x* is too large in magnitude, or *x* is 0.0, or the correct result would
32447 cause overflow or underflow.

32448 No other errors will occur.

32449 **EXAMPLES**

32450 None.

32451 **APPLICATION USAGE**

32452 None.

32453 **FUTURE DIRECTIONS**

32454 None.

32455 **SEE ALSO**32456 *isnan()*, *j0()*, `<math.h>`.32457 **CHANGE HISTORY**

32458 First released in Issue 1.

32459 Derived from Issue 1 of the SVID.

32460 **Issue 4**

32461 The following changes are incorporated in this issue:

- 32462 • Removed references to *matherr()*. |
- 32463 • The RETURN VALUE and ERRORS sections are substantially rewritten to rationalise error |
- 32464 handling in the mathematics functions.

32465 **Issue 5**

32466 The DESCRIPTION is updated to indicate how an application should check for an error. This |

32467 text was previously published in the APPLICATION USAGE section.

Headers

32468

32469 This chapter describes the contents of headers used by the X/Open functions, macros and
32470 external variables.

32471 Headers contain function prototypes, the definition of symbolic constants, common structures,
32472 preprocessor macros and defined types. Each function in Chapter 3 specifies the headers that an
32473 application must include in order to use that function. In most cases only one header is required.
32474 These headers are present on an application development system; they do not have to be present
32475 on the target execution system.

32476 NAME

32477 aio.h — asynchronous input and output (**REALTIME**)

32478 SYNOPSIS

32479 RT #include <aio.h>

32480

32481 DESCRIPTION

32482 The <aio.h> header defines the **aio_cb** structure which includes at least the following members:

32483	int	aio_fildes	file descriptor
32484	off_t	aio_offset	file offset
32485	volatile void*	aio_buf	location of buffer
32486	size_t	aio_nbytes	length of transfer
32487	int	aio_reqprio	request priority offset
32488	struct sigevent	aio_sigevent	signal number and value
32489	int	aio_lio_opcode	operation to be performed

32490 This header also includes the following constants:

32491 AIO_CANCELED
 32492 AIO_NOTCANCELED
 32493 AIO_ALLDONE
 32494 LIO_WAIT
 32495 LIO_NOWAIT
 32496 LIO_READ
 32497 LIO_WRITE
 32498 LIO_NOP

32499 The following are declared as functions and may also be declared as macros. Function
 32500 prototypes must be provided for use with an ISO C compiler.

```

32501 int      aio_cancel(int, struct aiocb *);
32502 int      aio_error(const struct aiocb *);
32503 int      aio_fsync(int, struct aiocb *);
32504 int      aio_read(struct aiocb *);
32505 ssize_t  aio_return(struct aiocb *);
32506 int      aio_suspend(const struct aiocb *const[], int,
32507                     const struct timespec *);
32508 int      aio_write(struct aiocb *);
32509 int      lio_listio(int, struct aiocb *const[], int,
32510                    struct sigevent *);
  
```

32511 Inclusion of the <aio.h> header may make visible symbols defined in the headers <fcntl.h>,
 32512 <signal.h>, <sys/types.h> and <time.h>.

32513 APPLICATION USAGE

32514 None.

32515 FUTURE DIRECTIONS

32516 None.

32517 SEE ALSO

32518 *fsync()*, *lseek()*, *read()*, *write()*, <fcntl.h>, <signal.h>, <sys/types.h>, <time.h>.

32519	CHANGE HISTORY	
32520	First released in Issue 5.	
32521	Included for alignment with the POSIX Realtime Extension.	

32522 NAME

32523 assert.h — verify program assertion

32524 SYNOPSIS

32525 #include <assert.h>

32526 DESCRIPTION

32527 The **<assert.h>** header defines the *assert()* macro. It refers to the macro *NDEBUG* which is not
32528 defined in the header. If *NDEBUG* is defined as a macro name before the inclusion of this
32529 header, the *assert()* macro is defined simply as:

32530 #define assert(ignore)((void) 0)

32531 otherwise the macro behaves as described in *assert()*.

32532 The *assert()* macro is implemented as a macro, not as a function. If the macro definition is
32533 suppressed in order to access an actual function, the behaviour is undefined.

32534 APPLICATION USAGE

32535 None.

32536 FUTURE DIRECTIONS

32537 None.

32538 SEE ALSO

32539 *assert()*.

32540 CHANGE HISTORY

32541 First released in Issue 1.

32542 Derived from Issue 1 of the SVID.

32543 **NAME**

32544 cpio.h — cpio archive values

32545 **SYNOPSIS**

32546 EX #include <cpio.h>

32547

32548 **DESCRIPTION**

32549 Values needed by the *c_mode* field of the *cpio* archive format are described by:

32550

32551	Name	Description	Value (octal)
32552	C_IRUSR	read by owner	0000400
32553	C_IWUSR	write by owner	0000200
32554	C_IXUSR	execute by owner	0000100
32555	C_IRGRP	read by group	0000040
32556	C_IWGRP	write by group	0000020
32557	C_IXGRP	execute by group	0000010
32558	C_IROTH	read by others	0000004
32559	C_IWOTH	write by others	0000002
32560	C_IXOTH	execute by others	0000001
32561	C_ISUID	set user ID	0004000
32562	C_ISGID	set group ID	0002000
32563	C_ISVTX	on directories, restricted deletion flag	0001000
32564	C_ISDIR	directory	0040000
32565	C_ISFIFO	FIFO	0010000
32566	C_ISREG	regular file	0100000
32567	C_ISBLK	block special	0060000
32568	C_ISCHR	character special	0020000
32569	C_ISCTG	reserved	0110000
32570 EX	C_ISLNK	symbolic link	0120000
32571	C_ISSOCK	socket	0140000

32572 The header defines the symbolic constant:

32573 MAGIC "070707"

32574 **APPLICATION USAGE**

32575 None.

32576 **FUTURE DIRECTIONS**

32577 None.

32578 **SEE ALSO**

32579 *cpio*, the XCU specification.

32580 **CHANGE HISTORY**

32581 First released in Issue 3 of the referenced **Headers** specification.

32582 Derived from the POSIX.1-1988 standard.

32583 **Issue 4, Version 2**

32584 Descriptions for C_ISLNK and C_ISSOCK are provided; formerly, these were listed as
32585 “Reserved”.

32586 **NAME**

32587 ctype.h — character types

32588 **SYNOPSIS**

32589 #include <ctype.h>

32590 **DESCRIPTION**

32591 The <ctype.h> header declares the following as functions and may also define them as macros.
 32592 Function prototypes must be provided for use with an ISO C compiler.

```

32593       int    isalnum(int);
32594       int    isalpha(int);
32595 EX       int   isascii(int);
32596       int    iscntrl(int);
32597       int    isdigit(int);
32598       int    isgraph(int);
32599       int    islower(int);
32600       int    isprint(int);
32601       int    ispunct(int);
32602       int    isspace(int);
32603       int    isupper(int);
32604       int    isxdigit(int);
32605 EX       int   toascii(int);
32606       int    tolower(int);
32607       int    toupper(int);

```

32608 The following are defined as macros:

```

32609 EX       int    _toupper(int);
32610       int    _tolower(int);
32611

```

32612 **APPLICATION USAGE**

32613 None.

32614 **FUTURE DIRECTIONS**

32615 None.

32616 **SEE ALSO**

32617 *isalnum()*, *isalpha()*, *isascii()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*,
 32618 *isupper()*, *isxdigit()*, *mblen()*, *mbstowcs()*, *mbtowc()*, *setlocale()*, *toascii()*, *tolower()*, *_tolower()*,
 32619 *toupper()*, *_toupper()*, *wcstombs()*, *wctomb()*, <locale.h>.

32620 **CHANGE HISTORY**

32621 First released in Issue 1.

32622 Derived from Issue 1 of the SVID.

32623 **Issue 4**

32624 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 32625
 - The function declarations in this header are expanded to full ISO C prototypes.

32626 NAME

32627 dirent.h — format of directory entries

32628 SYNOPSIS

32629 #include <dirent.h>

32630 DESCRIPTION

32631 The internal format of directories is unspecified.

32632 The <dirent.h> header defines the following data type through **typedef**:

32633 DIR A type representing a directory stream.

32634 It also defines the structure **dirent** which includes the following members:

32635 EX ino_t d_ino file serial number

32636 char d_name[] name of entry

32637 EX The type **ino_t** is defined as described in <sys/types.h>.

32638 The character array **d_name** is of unspecified size, but the number of bytes preceding the
32639 terminating null byte will not exceed {NAME_MAX}.

32640 The following are declared as functions and may also be defined as macros. Function prototypes
32641 must be provided for use with an ISO C compiler.

32642 int closedir(DIR *);

32643 DIR *opendir(const char *);

32644 struct dirent *readdir(DIR *);

32645 int readdir_r(DIR *, struct direct *, struct dirent **);

32646 void rewinddir(DIR *);

32647 EX void seekdir(DIR *, long int);

32648 long int telldir(DIR *);

32650 APPLICATION USAGE

32651 None.

32652 FUTURE DIRECTIONS

32653 None.

32654 SEE ALSO

32655 *closedir()*, *opendir()*, *readdir()*, *rewinddir()*, *seekdir()*, *telldir()*, <sys/types.h>.

32656 CHANGE HISTORY

32657 First released in Issue 2.

32658 Issue 4

32659 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- 32660 • The function declarations in this header are expanded to full ISO C prototypes.
- 32661 • A statement is added to the DESCRIPTION indicating that the internal format of directories
- 32662 is unspecified. Also in the description of the *d_name* field, the text is changed to indicate
- 32663 “bytes” rather than (possibly multi-byte) “characters”.

32664 Another change is incorporated as follows:

- 32665 • Reference to type **ino_t** is marked as an extension, as are references to the *seekdir()* and
- 32666 *telldir()* functions.

32667 **Issue 5**

32668 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

|

32669 **NAME**

32670 dlfcn.h — dynamic linking

32671 **SYNOPSIS**

32672 EX #include <dlfcn.h>

32673

32674 **DESCRIPTION**

32675 The <dlfcn.h> header defines at least the following macros for use in the construction of a
32676 *dlopen()* mode argument:

32677	RTLD_LAZY	Relocations are performed at an implementation-dependent time.
32678	RTLD_NOW	Relocations are performed when the object is loaded.
32679	RTLD_GLOBAL	All symbols are available for relocation processing of other modules.
32680	RTLD_LOCAL	All symbols are not made available for relocation processing by other
32681		modules.

32682 The header <dlfcn.h> declares the following functions which may also be defined as macros.
32683 Function prototypes must be provided for use with an ISO C compiler.

32684	void	*dlopen(const char *, int);
32685	void	*dlsym(void *, const char *);
32686	int	dlclose(void *);
32687	char	*dlerror(void);

32688 **APPLICATION USAGE**

32689 None.

32690 **FUTURE DIRECTIONS**

32691 None.

32692 **SEE ALSO**

32693 *dlopen()*, *dlclose()*, *dlsym()*, *dlerror()*.

32694 **CHANGE HISTORY**

32695 First released in Issue 5.

32696 NAME

32697 errno.h — system error numbers

32698 SYNOPSIS

32699 #include <errno.h>

32700 DESCRIPTION

32701 The <errno.h> header provides a declaration for *errno* and gives non-zero values for the
 32702 EX following symbolic constants. Their values are unique except as noted below:

32703	E2BIG	Argument list too long.
32704	EACCES	Permission denied.
32705 EX	EADDRINUSE	Address in use.
32706	EADDRNOTAVAIL	Address not available.
32707	EAFNOSUPPORT	Address family not supported.
32708 EX	EAGAIN	Resource unavailable, try again (may be the same value as
32709		EWouldBlock).
32710 EX	EALREADY	Connection already in progress.
32711	EBADF	Bad file descriptor.
32712 EX	EBADMSG	Bad message.
32713	EBUSY	Device or resource busy.
32714 RT	ECANCELED	Operation canceled.
32715	ECHILD	No child processes.
32716 EX	ECONNABORTED	Connection aborted.
32717	ECONNREFUSED	Connection refused.
32718	ECONNRESET	Connection reset.
32719	EDEADLK	Resource deadlock would occur.
32720 EX	EDESTADDRREQ	Destination address required.
32721	EDOM	Mathematics argument out of domain of function.
32722 EX	EDQUOT	Reserved.
32723	EEXIST	File exists.
32724	EFAULT	Bad address.
32725	EFBIG	File too large.
32726 EX	EHOSTUNREACH	Host is unreachable.
32727	EIDRM	Identifier removed.
32728	EILSEQ	Illegal byte sequence.
32729	EINPROGRESS	Operation in progress.
32730	EINTR	Interrupted function.
32731	EINVAL	Invalid argument.
32732	EIO	I/O error.
32733 EX	EISCONN	Socket is connected.
32734	EISDIR	Is a directory.
32735 EX	ELOOP	Too many levels of symbolic links.
32736	EMFILE	Too many open files.
32737	EMLINK	Too many links.
32738 EX	EMSGSIZE	Message too large.
32739	EMULTIHOP	Reserved.
32740	ENAMETOOLONG	Filename too long.
32741 EX	ENETDOWN	Network is down.
32742	ENETUNREACH	Network unreachable.
32743	ENFILE	Too many files open in system.
32744 EX	ENOBUFS	No buffer space available.
32745	ENODATA	No message is available on the STREAM head read queue.
32746	ENODEV	No such device.

32747	ENOENT	No such file or directory.
32748	ENOEXEC	Executable file format error.
32749	ENOLCK	No locks available.
32750 EX	ENOLINK	Reserved.
32751	ENOMEM	Not enough space.
32752 EX	ENOMSG	No message of the desired type.
32753	ENOPROTOOPT	Protocol not available.
32754	ENOSPC	No space left on device.
32755 EX	ENOSR	No STREAM resources.
32756	ENOSTR	Not a STREAM.
32757	ENOSYS	Function not supported.
32758 EX	ENOTCONN	The socket is not connected.
32759	ENOTDIR	Not a directory.
32760	ENOTEMPTY	Directory not empty.
32761 EX	ENOTSOCK	Not a socket.
32762	ENOTSUP	Not supported.
32763	ENOTTY	Inappropriate I/O control operation.
32764	ENXIO	No such device or address.
32765 EX	EOPNOTSUPP	Operation not supported on socket.
32766	EOVERFLOW	Value too large to be stored in data type.
32767 FIPS	EPERM	Operation not permitted.
32768	EPIPE	Broken pipe.
32769 EX	EPROTO	Protocol error.
32770	EPROTONOSUPPORT	Protocol not supported.
32771	EPROTOTYPE	Socket type not supported.
32772	ERANGE	Result too large.
32773	EROFS	Read-only file system.
32774	ESPIPE	Invalid seek.
32775	ESRCH	No such process.
32776 EX	ESTALE	Reserved.
32777	ETIME	Stream <i>ioctl()</i> timeout.
32778	ETIMEDOUT	Connection timed out.
32779	ETXTBSY	Text file busy.
32780	EWOULDBLOCK	Operation would block (may be the same value as [EAGAIN]).
32781	EXDEV	Cross-device link.

32782 APPLICATION USAGE

32783 Additional error numbers may be defined on XSI-conformant systems. See Section 2.3.1 on page
 32784 29.

32785 FUTURE DIRECTIONS

32786 None.

32787 SEE ALSO

32788 Section 2.3 on page 22.

32789 CHANGE HISTORY

32790 First released in Issue 1.

32791 Derived from Issue 1 of the SVID.

32792 Issue 4

32793 The following changes are incorporated in this issue:

- 32794 • The [EILSEQ] error is added and marked as an EX interface.

32795 • The [ENOTBLK] error is withdrawn.

32796 **Issue 4, Version 2**

32797 The EADDRINUSE, EADDRNOTAVAIL, EAFNOSUPPORT, EALREADY, EBADMSG,
32798 ECONNABORTED, ECONNREFUSED, ECONNRESET, EDESTADDRREQ, EDQUOT,
32799 EHOSTUNREACH, EINPROGRESS, EISCONN, ELOOP, EMSGSIZE, EMULTIHOP,
32800 ENETDOWN, ENETUNREACH, ENOBUFS, ENODATA, ENOLINK, ENOPROTOOPT, ENOSR,
32801 ENOSTR, ENOTCONN, ENOTSOCK, EOPNOTSUPP, EOVERFLOW, EPROTO,
32802 EPROTONOSUPPORT, EPROTOTYPE, ESTALE, ETIME, ETIMEDOUT and EWOULDBLOCK
32803 errors are added in the UX context.

32804 **Issue 5**

32805 Updated for alignment with the POSIX Realtime Extension.

32806 NAME

32807 fcntl.h — file control options

32808 SYNOPSIS

32809 #include <fcntl.h>

32810 DESCRIPTION

32811 The <fcntl.h> header defines the following requests and arguments for use by the functions
32812 *fcntl()* and *open()*.

32813 Values for *cmd* used by *fcntl()* (the following values are unique):

32814	F_DUPFD	Duplicate file descriptor.
32815	F_GETFD	Get file descriptor flags.
32816	F_SETFD	Set file descriptor flags.
32817	F_GETFL	Get file status flags and file access modes.
32818	F_SETFL	Set file status flags.
32819	F_GETLK	Get record locking information.
32820	F_SETLK	Set record locking information.
32821	F_SETLKW	Set record locking information; wait if blocked.

32822 File descriptor flags used for *fcntl()*:

32823	FD_CLOEXEC	Close the file descriptor upon execution of an <i>exec</i> family function.
-------	------------	---

32824 Values for *l_type* used for record locking with *fcntl()* (the following values are unique):

32825	F_RDLCK	Shared or read lock.
32826	F_UNLCK	Unlock.
32827	F_WRLCK	Exclusive or write lock.

32828 EX The values used for *l_whence*, *SEEK_SET*, *SEEK_CUR* and *SEEK_END* are defined as described
32829 in <unistd.h>.

32830 The following four sets of values for *oflag* used by *open()* are bitwise distinct:

32831	O_CREAT	Create file if it does not exist.
32832	O_EXCL	Exclusive use flag.
32833	O_NOCTTY	Do not assign controlling terminal.
32834	O_TRUNC	Truncate flag.

32835 File status flags used for *open()* and *fcntl()*:

32836	O_APPEND	Set append mode.
32837 RT	O_DSYNC	Write according to synchronised I/O data integrity completion.
32838	O_NONBLOCK	Non-blocking mode.
32839 RT	O_RSYNC	Synchronised read I/O operations.
32840	O_SYNC	Write according to synchronised I/O file integrity completion.

32841 Mask for use with file access modes:

32842	O_ACCMODE	Mask for file access modes.
-------	-----------	-----------------------------

32843 File access modes used for *open()* and *fcntl()*:

32844	O_RDONLY	Open for reading only.
32845	O_RDWR	Open for reading and writing.
32846	O_WRONLY	Open for writing only.

32847 EX The symbolic names for file modes for use as values of **mode_t** are defined as described in
32848 <sys/stat.h>.

32849 The structure **flock** describes a file lock. It includes the following members:

32850 short l_type type of lock; F_RDLCK, F_WRLCK, F_UNLCK

32851 short l_whence flag for starting offset

32852 off_t l_start relative offset in bytes

32853 off_t l_len size; if 0 then until EOF

32854 pid_t l_pid process ID of the process holding the lock; returned with F_GETLK

32855 EX The **mode_t**, **off_t** and **pid_t** types are defined as described in <sys/types.h>.

32856 The following are declared as functions and may also be defined as macros. Function prototypes

32857 must be provided for use with an ISO C compiler.

32858 int creat(const char *, mode_t);

32859 int fcntl(int, int, ...);

32860 int open(const char *, int, ...);

32861 EX Inclusion of the <fcntl.h> header may also make visible all symbols from <sys/stat.h> and

32862 <unistd.h>.

32863 **APPLICATION USAGE**

32864 None.

32865 **FUTURE DIRECTIONS**

32866 None.

32867 **SEE ALSO**

32868 *creat()*, *exec*, *fcntl()*, *open()*, <sys/stat.h>, <sys/types.h>, <unistd.h>.

32869 **CHANGE HISTORY**

32870 First released in Issue 1.

32871 Derived from Issue 1 of the SVID.

32872 **Issue 4**

32873 The following change is incorporated for alignment with the ISO POSIX-1 standard:

32874 • The function declarations in this header are expanded to full ISO C prototypes.

32875 Other changes are incorporated as follows:

32876 • A reference to <unistd.h> is added for the definition of *l_whence*, SEEK_SET, SEEK_CUR and

32877 SEEK_END, and marked as an extension.

32878 • A reference to <sys/stat.h> is added for the symbolic names of file modes used as values of

32879 **mode_t**, and marked as an extension.

32880 • A reference to <sys/types.h> is added for the definition of **mode_t**, **off_t** and **pid_t**, and

32881 marked as an extension.

32882 • A warning is added indicating that inclusion of <fcntl.h> may also make visible all symbols

32883 from <sys/stat.h> and <unistd.h>. This is marked as an extension.

32884 **Issue 5**

32885 The DESCRIPTION is updated for alignment with POSIX Realtime Extension.

32886 **NAME**

32887 float.h — floating types

32888 **SYNOPSIS**

32889 #include <float.h>

32890 **DESCRIPTION**

32891 The characteristics of floating types are defined in terms of a model that describes a
32892 representation of floating-point numbers and values that provide information about an
32893 implementation's floating-point arithmetic.

32894 The following parameters are used to define the model for each floating-point type:

- 32895 s sign (± 1)
- 32896 b base or radix of exponent representation (an integer > 1)
- 32897 e exponent (an integer between a minimum e_{\min} and a maximum e_{\max})
- 32898 p precision (the number of base- b digits in the significand)
- 32899 f_k non-negative integers less than b (the significand digits)

32900 A normalised floating-point number x ($f_1 > 0$ if $x \neq 0$) is defined by the following model:

$$32901 \quad x = s \times b^e \times \sum_{k=1}^p f_k \times b^{-k}, \quad e_{\min} \leq e \leq e_{\max}$$

32902

32903 FLT_RADIX will be a constant expression suitable for use in the **#if** preprocessing directives. All
32904 except FLT_RADIX and FLT_ROUNDSD have separate names for all three floating-point types.
32905 The floating-point model representation is provided for all macro names except FLT_ROUNDSD.

32906 The rounding mode for floating-point addition is characterised by the value of FLT_ROUNDSD:

- 32907 -1 indeterminable
- 32908 0 toward 0.0
- 32909 1 to nearest
- 32910 2 toward positive infinity
- 32911 3 toward negative infinity

32912 All other values for FLT_ROUNDSD characterise implementation-dependent rounding behaviour.

32913 The macro names given in the following list will be defined as expressions with values that are
32914 equal or greater in magnitude (absolute value) to those shown, with the same sign.

32915

Name	Description	Value
FLT_RADIX	radix of exponent representation, b	2
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	number of base-FLT_RADIX digits in the floating-point significand, p	
FLT_DIG DBL_DIG LDBL_DIG	number of decimal digits, q , such that any floating-point number with q decimal digits can be rounded into a floating-point number with p radix b digits and back again without change to the q decimal digits, $\left\lfloor (p-1) \times \log_{10} b \right\rfloor + \begin{cases} 1 & \text{if } b \text{ is a power of } 10 \\ 0 & \text{otherwise} \end{cases}$	6 10 10
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalised floating-point number, e_{\min}	
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	minimum negative integer such that 10 raised to that power is in the range of normalised floating point numbers, $\left\lceil \log_{10} b^{e_{\min}^{-1}} \right\rceil$	-37 -37 -37
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	maximum integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number, e_{\max}	
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, $\left\lfloor \log_{10} ((1 - b^{-p}) \times b^{e_{\max}}) \right\rfloor$	37 37 37

32916

The macro names given in the following list will be defined as expressions with values that will be equal to or greater than those shown.

32917

32918

FLT_MAX DBL_MAX LDBL_MAX	maximum representable finite floating-point number, $(1 - b^{-p}) \times b^{e_{\max}}$	1E+37 1E+37 1E+37
--------------------------------	---	-------------------------

32919

The macro names given in the following list will be defined as expressions with values that will be equal to or less than those shown.

32920

32921

FLT_EPSILON	the difference between 1.0 and the least value greater than 1.0 that is representable in the given floating-point type, $b^{(1-p)}$	1E-5
DBL_EPSILON		1E-9
LDBL_EPSILON		1E-9
FLT_MIN	minimum normalised positive floating-point number, $b^{(e_{\min}-1)}$	1E-37
DBL_MIN		1E-37
LDBL_MIN		1E-37

32922 **APPLICATION USAGE**

32923 None.

32924 **FUTURE DIRECTIONS**

32925 None.

32926 **SEE ALSO**

32927 None.

32928 **CHANGE HISTORY**

32929 First released in Issue 4.

32930 Derived from the ISO C standard.

32931 NAME

32932 fmtmsg.h — message display structures

32933 SYNOPSIS

32934 EX #include <fmtmsg.h>

32935

32936 DESCRIPTION

32937 The <fmtmsg.h> header defines the following macros, which expand to constant integral
32938 expressions:

32939	MM_HARD	Source of the condition is hardware.
32940	MM_SOFT	Source of the condition is software.
32941	MM_FIRM	Source of the condition is firmware.
32942	MM_APPL	Condition detected by application.
32943	MM_UTIL	Condition detected by utility.
32944	MM_OPSYS	Condition detected by operating system.
32945	MM_RECOVER	Recoverable error.
32946	MM_NRECOV	Non-recoverable error.
32947	MM_HALT	Error causing application to halt.
32948	MM_ERROR	Application has encountered a non-fatal fault.
32949	MM_WARNING	Application has detected unusual non-error condition.
32950	MM_INFO	Informative message.
32951	MM_NOSEV	No severity level provided for the message.
32952	MM_PRINT	Display message on standard error.
32953	MM_CONSOLE	Display message on system console.

32954 The table below indicates the null values and identifiers for *fmtmsg()* arguments. The
32955 <fmtmsg.h> header defines the macros in the **Identifier** column, which expand to constant
32956 expressions that expand to expressions of the type indicated in the **Type** column:

32957

32958

32959

32960

32961

32962

32963

32964

Argument	Type	Null-Value	Identifier
<i>label</i>	char*	(char*)0	MM_NULLLBL
<i>severity</i>	int	0	MM_NULLSEV
<i>class</i>	long int	0L	MM_NULLMC
<i>text</i>	char*	(char*)0	MM_NULLTXT
<i>action</i>	char*	(char*)0	MM_NULLACT
<i>tag</i>	char*	(char*)0	MM_NULLTAG

32965 The <fmtmsg.h> header also defines the following macros for use as return values for *fmtmsg()*:

32966	MM_OK	The function succeeded.
32967	MM_NOTOK	The function failed completely.
32968	MM_NOMSG	The function was unable to generate a message on standard error, but
32969		otherwise succeeded.
32970	MM_NOCON	The function was unable to generate a console message, but otherwise
32971		succeeded.

32972 The following is declared as a function and may also be defined as a macro. A function
32973 prototype must be provided for use with an ISO C compiler.

```
32974 int fmtmsg(long, const char*, int,
32975            const char*, const char*, const char*);
```


32976 **APPLICATION USAGE**

32977 None.

32978 **FUTURE DIRECTIONS**

32979 None.

32980 **SEE ALSO**

32981 *fmtmsg()*.

32982 **CHANGE HISTORY**

32983 First released in Issue 4, Version 2.

32984 **NAME**

32985 fnmatch.h — filename-matching types

32986 **SYNOPSIS**

32987 #include <fnmatch.h>

32988 **DESCRIPTION**32989 The <fnmatch.h> header defines the flags and return value used by the *fnmatch()* function. The
32990 following constants are defined:

32991 FNM_NOMATCH The string does not match the specified pattern.
 32992 FNM_PATHNAME Slash in *string* only matches slash in *pattern*.
 32993 FNM_PERIOD Leading period in *string* must be exactly matched by period in *pattern*.
 32994 FNM_NOESCAPE Disable backslash escaping.
 32995 FNM_NOSYS The implementation does not support this function.

32996 The following is declared as a function and may also be declared as a macro. Function
 32997 prototypes must be provided for use with an ISO C compiler.

32998 int fnmatch(const char *, const char *, int);

32999 **APPLICATION USAGE**

33000 None.

33001 **FUTURE DIRECTIONS**

33002 None.

33003 **SEE ALSO**33004 *fnmatch()*, the XCU specification.33005 **CHANGE HISTORY**

33006 First released in Issue 4.

33007 Derived from the ISO POSIX-2 standard.

33008 NAME

33009 ftw.h — file tree traversal

33010 SYNOPSIS

33011 EX `#include <ftw.h>`

33012

33013 DESCRIPTION

33014 The <ftw.h> header defines the **FTW** structure that includes at least the following members:

33015 int base
33016 int level

33017 The <ftw.h> header defines macros for use as values of the third argument to the application-
33018 supplied function that is passed as the second argument to *ftw()* and *nftw()*:

33019 **FTW_F** File.
33020 **FTW_D** Directory.
33021 **FTW_DNR** Directory without read permission.
33022 **FTW_DP** Directory with subdirectories visited.
33023 **FTW_NS** Unknown type, *stat()* failed.
33024 **FTW_SL** Symbolic link.
33025 **FTW_SLN** Symbolic link that names a non-existent file.

33026 The <ftw.h> header defines macros for use as values of the fourth argument to *nftw()*:

33027 **FTW_PHYS** Physical walk, does not follow symbolic links. Otherwise, *nftw()* will
33028 follow links but will not walk down any path that crosses itself.
33029 **FTW_MOUNT** The walk will not cross a mount point.
33030 **FTW_DEPTH** All subdirectories will be visited before the directory itself.
33031 **FTW_CHDIR** The walk will change to each directory before reading it.

33032 The following are declared as functions and may also be defined as macros. Function prototypes
33033 must be provided for use with an ISO C compiler.

33034 int ftw(const char *,
33035 int (*)(const char *, const struct stat *, int), int);
33036 int nftw(const char *, int (*)(
33037 const char *, const struct stat *, int, struct FTW*),
33038 int, int);

33039 The <ftw.h> header defines the **stat** structure and the symbolic names for **st_mode** and the file
33040 type test macros as described in <sys/stat.h>.

33041 Inclusion of the <ftw.h> header may also make visible all symbols from <sys/stat.h>.

33042 APPLICATION USAGE

33043 None.

33044 FUTURE DIRECTIONS

33045 None.

33046 SEE ALSO

33047 *ftw()*, *nftw()*, <sys/stat.h>.

33048 CHANGE HISTORY

33049 First released in Issue 1.

33050 Derived from Issue 1 of the SVID.

33051 **Issue 4**

33052 The following changes are incorporated in this issue:

- 33053 • The function declarations in this header are expanded to full ISO C prototypes.
- 33054 • A reference to <sys/stat.h> is added for the definition of the **stat** structure, the symbolic
33055 names for **st_mode** and the file type test macros.
- 33056 • A warning is added indicating that inclusion of <ftw.h> may also make visible all symbols
33057 from <sys/stat.h>.

33058 **Issue 4, Version 2**33059 The following changes are incorporated in the *DESCRIPTION* for X/OPEN UNIX conformance:

- 33060 • The **FTW** structure is defined.
- 33061 • The *nftw()* function is declared by the header and is mentioned as one of the functions to
33062 which the first list of macros applies.
- 33063 • FTW_SL and FTW_SLN are added to the first list of macros to handle symbolic links.
- 33064 • Macros for use as values of the fourth argument to *nftw()* are defined.

33065 **Issue 5**

33066 A description of FTW_DP is added.

33067 **NAME**

33068 glob.h — pathname pattern-matching types

33069 **SYNOPSIS**

33070 #include <glob.h>

33071 **DESCRIPTION**

33072 The <glob.h> header defines the structures and symbolic constants used by the *glob()* function.

33073 The structure type **glob_t** contains at least the following members:

33074 size_t gl_pathc count of paths matched by *pattern*
 33075 char **gl_pathv pointer to a list of matched pathnames
 33076 size_t gl_offs slots to reserve at the beginning of **gl_pathv**

33077 The following constants are provided as values for the *flags* argument:

33078 GLOB_APPEND Append generated pathnames to those previously obtained.
 33079 GLOB_DOOFFS Specify how many null pointers to add to the beginning of
 33080 *pglob->gl_pathv*.
 33081 GLOB_ERR Cause *glob()* to return on error.
 33082 GLOB_MARK Each pathname that is a directory that matches *pattern* has a slash
 33083 appended.
 33084 GLOB_NOCHECK If *pattern* does not match any pathname, then return a list consisting of
 33085 only *pattern*.
 33086 GLOB_NOESCAPE Disable backslash escaping.
 33087 GLOB_NOSORT Do not sort the pathnames returned.

33088 The following constants are defined as error return values:

33089 GLOB_ABORTED The scan was stopped because GLOB_ERR was set or (*errfunc)()
 33090 returned non-zero.
 33091 GLOB_NOMATCH The pattern does not match any existing pathname, and
 33092 GLOB_NOCHECK was not set in flags.
 33093 GLOB_NOSPACE An attempt to allocate memory failed.
 33094 GLOB_NOSYS The implementation does not support this function.

33095 The following are declared as functions and may also be declared as macros. Function
 33096 prototypes must be provided for use with an ISO C compiler.

33097 int glob(const char *, int,
 33098 int (*)(const char *, int), glob_t *);
 33099 void globfree (glob_t *);

33100 The implementation may define additional macros or constants using names beginning with
 33101 GLOB_.

33102 **APPLICATION USAGE**

33103 None.

33104 **FUTURE DIRECTIONS**

33105 None.

33106 **SEE ALSO**

33107 *glob()*, the XCU specification.

33108 **CHANGE HISTORY**

33109 First released in Issue 4.

33110 Derived from the ISO POSIX-2 standard.

33111 NAME

33112 grp.h — group structure

33113 SYNOPSIS

33114 #include <grp.h>

33115 DESCRIPTION

33116 The <grp.h> header declares the structure **group** which includes the following members:

33117 char *gr_name the name of the group
33118 gid_t gr_gid numerical group ID
33119 char **gr_mem pointer to a null-terminated array of character
33120 pointers to member names

33121 EX The **gid_t** type is defined as described in <sys/types.h>.

33122 The following are declared as functions and may also be defined as macros. Function prototypes
33123 must be provided for use with an ISO C compiler.

33124 struct group *getgrgid(gid_t);
33125 struct group *getgrnam(const char *);
33126 int getgrgid_r(gid_t, struct group *, char *,
33127 size_t, struct group **);
33128 int getgrnam_r(const char *, struct group *, char *,
33129 size_t, struct group **);

33130 EX struct group *getgrent(void);
33131 void endgrent(void);
33132 void setgrent(void);

33133

33134 APPLICATION USAGE

33135 None.

33136 FUTURE DIRECTIONS

33137 None.

33138 SEE ALSO

33139 endgrent(), getgrgid(), getgrgid_r(), getgrnam(), <sys/types.h>.

33140 CHANGE HISTORY

33141 First released in Issue 1.

33142 Issue 4

33143 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 33144 • The function declarations in this header are expanded to full ISO C prototypes.

33145 Another change is incorporated as follows:

- 33146 • A reference to <sys/types.h> is added for the definition of **gid_t** and marked as an extension.

33147 Issue 4, Version 2

33148 For X/OPEN UNIX conformance, the *getgrent()*, *endgrent()* and *setgrent()* functions are added to
33149 the list of functions declared in this header.

33150 Issue 5

33151 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

33152 **NAME**33153 **iconv.h** — codeset conversion facility33154 **SYNOPSIS**33155 EX `#include <iconv.h>`

33156

33157 **DESCRIPTION**33158 The **<iconv.h>** header defines the following data type through **typedef**:33159 **iconv_t** Identifies the conversion from one codeset to another.33160 The following are declared as functions and may also be declared as macros. Function
33161 prototypes must be provided for use with an ISO C compiler.33162 `iconv_t iconv_open(const char *, const char *);`33163 `size_t iconv(iconv_t, char **, size_t *, char **, size_t *);`33164 `int iconv_close(iconv_t);`33165 **APPLICATION USAGE**

33166 None.

33167 **FUTURE DIRECTIONS**

33168 None.

33169 **SEE ALSO**33170 *iconv_open()*, *iconv()*, *iconv_close()*.33171 **CHANGE HISTORY**

33172 First released in Issue 4.

33173 **NAME**

33174 inttypes.h — fixed size integral types

33175 **SYNOPSIS**

33176 EX #include <inttypes.h>

33177

33178 **DESCRIPTION**

33179 The <inttypes.h> header includes definitions of at least the following types:

33180	int8_t	8-bit signed integral type.
33181	int16_t	16-bit signed integral type.
33182	int32_t	32-bit signed integral type.
33183	int64_t	64-bit signed integral type.
33184	uint8_t	8-bit unsigned integral type.
33185	uint16_t	16-bit unsigned integral type.
33186	uint32_t	32-bit unsigned integral type.
33187	uint64_t	64-bit unsigned integral type.
33188	intptr_t	Signed integral type large enough to hold any pointer.
33189	uintptr_t	Unsigned integral type large enough to hold any pointer.

33190 **APPLICATION USAGE**

33191 None.

33192 **FUTURE DIRECTIONS**

33193 None.

33194 **SEE ALSO**

33195 None.

33196 **CHANGE HISTORY**

33197 First released in Issue 5.

33198 NAME

33199 iso646.h — alternative spellings

33200 SYNOPSIS

33201 #include <iso646.h>

33202 DESCRIPTION

33203 The **<iso646.h>** header defines the following eleven macros (on the left) that expand to the
33204 corresponding tokens (on the right):

33205 and &&

33206 and_eq &=

33207 bitand &

33208 bitor |

33209 compl ~

33210 not !

33211 not_eq !=

33212 or | |

33213 or_eq |=

33214 xor ^

33215 xor_eq ^=

33216 APPLICATION USAGE

33217 None.

33218 FUTURE DIRECTIONS

33219 None.

33220 SEE ALSO

33221 None.

33222 CHANGE HISTORY

33223 First released in Issue 5.

33224 Derived from ISO/IEC 9899:1990/Amendment 1:1994 (E).

33225 **NAME**

33226 langinfo.h — language information constants

33227 **SYNOPSIS**33228 EX

```
#include <langinfo.h>
```

33229

33230 **DESCRIPTION**

33231 The <langinfo.h> header contains the constants used to identify items of *langinfo* data (see
 33232 *nl_langinfo()*). The type of the constants, **nl_item**, is defined as described in <nl_types.h>. The
 33233 following constants are defined on all XSI-conformant systems.

33234 The entries under **Category** indicate in which *setlocale()* category each item is defined.

33235

33236	Constant	Category	Meaning
33237	CODESET	LC_CTYPE	codeset name
33238	D_T_FMT	LC_TIME	string for formatting date and time
33239	D_FMT	LC_TIME	date format string
33240	T_FMT	LC_TIME	time format string
33241	T_FMT_AMPM	LC_TIME	a.m. or p.m. time format string
33242	AM_STR	LC_TIME	Ante Meridian affix
33243	PM_STR	LC_TIME	Post Meridian affix
33244	DAY_1	LC_TIME	name of the first day of the week (for example, Sunday)
33245	DAY_2	LC_TIME	name of the second day of the week (for example, Monday)
33246	DAY_3	LC_TIME	name of the third day of the week (for example, Tuesday)
33247	DAY_4	LC_TIME	name of the fourth day of the week
33248			(for example, Wednesday)
33249	DAY_5	LC_TIME	name of the fifth day of the week (for example, Thursday)
33250	DAY_6	LC_TIME	name of the sixth day of the week (for example, Friday)
33251	DAY_7	LC_TIME	name of the seventh day of the week
33252			(for example, Saturday)
33253	ABDAY_1	LC_TIME	abbreviated name of the first day of the week
33254	ABDAY_2	LC_TIME	abbreviated name of the second day of the week
33255	ABDAY_3	LC_TIME	abbreviated name of the third day of the week
33256	ABDAY_4	LC_TIME	abbreviated name of the fourth day of the week
33257	ABDAY_5	LC_TIME	abbreviated name of the fifth day of the week
33258	ABDAY_6	LC_TIME	abbreviated name of the sixth day of the week
33259	ABDAY_7	LC_TIME	abbreviated name of the seventh day of the week
33260	MON_1	LC_TIME	name of the first month of the year
33261	MON_2	LC_TIME	name of the second month
33262	MON_3	LC_TIME	name of the third month
33263	MON_4	LC_TIME	name of the fourth month
33264	MON_5	LC_TIME	name of the fifth month
33265	MON_6	LC_TIME	name of the sixth month
33266	MON_7	LC_TIME	name of the seventh month
33267	MON_8	LC_TIME	name of the eighth month
33268	MON_9	LC_TIME	name of the ninth month
33269	MON_10	LC_TIME	name of the tenth month
33270	MON_11	LC_TIME	name of the eleventh month
33271	MON_12	LC_TIME	name of the twelfth month

33272

33273

33274

33275

33276

33277

33278

33279

33280

33281

33282

33283

33284

33285

33286

33287

33288

33289

33290

33291

33292

33293

33294

33295

33296

33297

33298

33299

33300

33301

33302

Constant	Category	Meaning
ABMON_1	LC_TIME	abbreviated name of the first month
ABMON_2	LC_TIME	abbreviated name of the second month
ABMON_3	LC_TIME	abbreviated name of the third month
ABMON_4	LC_TIME	abbreviated name of the fourth month
ABMON_5	LC_TIME	abbreviated name of the fifth month
ABMON_6	LC_TIME	abbreviated name of the sixth month
ABMON_7	LC_TIME	abbreviated name of the seventh month
ABMON_8	LC_TIME	abbreviated name of the eighth month
ABMON_9	LC_TIME	abbreviated name of the ninth month
ABMON_10	LC_TIME	abbreviated name of the tenth month
ABMON_11	LC_TIME	abbreviated name of the eleventh month
ABMON_12	LC_TIME	abbreviated name of the twelfth month
ERA	LC_TIME	era description segments
ERA_D_FMT	LC_TIME	era date format string
ERA_D_T_FMT	LC_TIME	era date and time format string
ERA_T_FMT	LC_TIME	era time format string
ALT_DIGITS	LC_TIME	alternative symbols for digits
RADIXCHAR	LC_NUMERIC	radix character
THOUSEP	LC_NUMERIC	separator for thousands
YESEXPR	LC_MESSAGES	affirmative response expression
NOEXPR	LC_MESSAGES	negative response expression
YESSTR	LC_MESSAGES	affirmative response for yes/no queries (LEGACY)
NOSTR	LC_MESSAGES	negative response for yes/no queries (LEGACY)
CRNCYSTR	LC_MONETARY	currency symbol, preceded by – if the symbol should appear before the value, + if the symbol should appear after the value, or . if the symbol should replace the radix character

33303

33304

If the locale's value for **p_cs_precedes** and **n_cs_precedes** do not match, the value of *nl_langinfo*(CRNCYSTR) is unspecified.

33305

The <langinfo.h> header declares the following as a function:

33306

```
char *nl_langinfo(nl_item);
```

33307

Inclusion of the <langinfo.h> header may also make visible all symbols from <nl_types.h>.

33308 APPLICATION USAGE

33309

33310

33311

33312

33313

Wherever possible, users are advised to use functions compatible with those in the ISO C standard to access items of *langinfo* data. In particular, the *strftime*() function should be used to access date and time information defined in category LC_TIME. The *localeconv*() function should be used to access information corresponding to RADIXCHAR, THOUSEP and CRNCYSTR.

33314 FUTURE DIRECTIONS

33315

None.

33316 SEE ALSO

33317

nl_langinfo(), *localeconv*(), *strfmon*(), *strftime*(), the XBD specification, Chapter 5, Locale.

33318 CHANGE HISTORY

33319

First released in Issue 2.

33320 **Issue 4**

33321 The following changes are incorporated in this issue:

- 33322 • The function declarations in this header are expanded to full ISO C prototypes.
- 33323 • The constants CODESET, T_FMT_AMPM, ERA, ERA_D_FMT, ALT_DIGITS, YESEXPR and
33324 NOEXPR are added.
- 33325 • The constants YESSTR and NOSTR are marked TO BE WITHDRAWN.
- 33326 • Reference to the Gregorian calendar is removed.
- 33327 • Constants YESSTR and NOSTR are now defined as belonging to category LC_MESSAGES.
33328 Previously they were defined as constants in category LC_ALL.
- 33329 • A warning is added indicating that inclusion of <langinfo.h> may also make visible all
33330 symbols from <nl_types.h>.
- 33331 • The APPLICATION USAGE section is expanded to recommend use of the *localeconv()* |
33332 function.

33333 **Issue 5**

33334 The constants YESSTR and NOSTR are marked LEGACY. |

33335 **NAME**

33336 libgen.h — definitions for pattern matching functions

33337 **SYNOPSIS**

33338 EX #include <libgen.h>

33339

33340 **DESCRIPTION**

33341 The <libgen.h> header declares the following external variable:

33342 extern char* __loc1 (**LEGACY**)33343 (Used by *regex()* to report pattern location.)33344 The following are declared as functions and may also be defined as macros. Function prototypes
33345 must be provided for use with an ISO C compiler.

33346 char *basename(char *);

33347 char *dirname(char *);

33348 char *regcomp(const char *, ...);

33349 char *regex(const char *, const char *, ...);

33350 **APPLICATION USAGE**33351 The function prototypes for *regcomp()* and *regex()* are included in this header for historical
33352 reasons. New applications should use the *regcomp()*, *regexexec()*, *regerror()* and *regfree()* functions,
33353 and the <regex.h> header, which provide full internationalised regular expression functionality
33354 compatible with the ISO POSIX-2 standard, as described in the **XBD** specification, **Chapter 7**,
33355 **Regular Expressions**.33356 **FUTURE DIRECTIONS**

33357 None.

33358 **SEE ALSO**33359 *basename()*, *dirname()*.33360 **CHANGE HISTORY**

33361 First released in Issue 4, Version 2.

33362 **Issue 5**33363 The function prototypes for *basename()* and *dirname()* are changed to indicate that the first
33364 argument is of type **char*** rather than **const char***.

33365 **NAME**

33366 limits.h — implementation-dependent constants

33367 **SYNOPSIS**

33368 #include <limits.h>

33369 **DESCRIPTION**

33370 The <limits.h> header defines various symbolic names. Different categories of names are
33371 described below.

33372 The names represent various limits on resources that the system imposes on applications.

33373 Implementations may choose any appropriate value for each limit, provided it is not more
33374 restrictive than the Minimum Acceptable Values listed below. Symbolic constant names
33375 beginning with _POSIX may be found in <unistd.h>.

33376 Applications should not assume any particular value for a limit. To achieve maximum
33377 portability, an application should not require more resource than the Minimum Acceptable
33378 Value quantity. However, an application wishing to avail itself of the full amount of a resource
33379 available on an implementation may make use of the value given in <limits.h> on that
33380 particular system, by using the symbolic names listed below. It should be noted, however, that
33381 many of the listed limits are not invariant, and at run time, the value of the limit may differ from
33382 those given in this header, for the following reasons:

- 33383 • The limit is pathname-dependent.
- 33384 • The limit differs between the compile and run-time machines.

33385 For these reasons, an application may use the *fpathconf()*, *pathconf()* and *sysconf()* functions to
33386 determine the actual value of a limit at run time.

33387 The items in the list ending in _MIN give the most negative values that the mathematical types
33388 are guaranteed to be capable of representing. Numbers of a more negative value may be
33389 supported on some systems, as indicated by the <limits.h> header on the system, but
33390 applications requiring such numbers are not guaranteed to be portable to all systems.

33391 The Minimum Acceptable Value symbol * indicates that there is no guaranteed value across all
33392 XSI-conformant systems.

33393 **Run-time Invariant Values (Possibly Indeterminate)**

33394 A definition of one of the symbolic names in the following list will be omitted from <limits.h>
33395 on specific implementations where the corresponding value is equal to or greater than the stated
33396 minimum, but is indeterminate.

33397 This might depend on the amount of available memory space on a specific instance of a specific
33398 implementation. The actual value supported by a specific instance will be provided by the
33399 *sysconf()* function.

33400 RT **AIO_LISTIO_MAX**

33401 Maximum number of I/O operations in a single list I/O call supported by the
33402 implementation.
33403 Minimum Acceptable Value: _POSIX_AIO_LISTIO_MAX

33404 **AIO_MAX**

33405 Maximum number of outstanding asynchronous I/O operations supported by the
33406 implementation.
33407 Minimum Acceptable Value: _POSIX_AIO_MAX

33408	AIO_PRIO_DELTA_MAX
33409	The maximum amount by which a process can decrease its asynchronous I/O priority level
33410	from its own scheduling priority.
33411	Minimum Acceptable Value: 0
33412	ARG_MAX
33413	Maximum length of argument to the <i>exec</i> functions including environment data.
33414	Minimum Acceptable Value: _POSIX_ARG_MAX
33415 EX	ATEXIT_MAX
33416	Maximum number of functions that may be registered with <i>atexit()</i> .
33417	Minimum Acceptable Value: 32
33418	CHILD_MAX
33419	Maximum number of simultaneous processes per real user ID.
33420 FIPS	Minimum Acceptable Value: 25
33421 RT	DELAYTIMER_MAX
33422	Maximum number of timer expiration overruns.
33423	Minimum Acceptable Value: _POSIX_DELAYTIMER_MAX
33424 EX	IOV_MAX
33425	Maximum number of iovec structures that one process has available for use with <i>readv()</i> or
33426	<i>writev()</i> .
33427	Minimum Acceptable Value: _XOPEN_IOV_MAX
33428	LOGIN_NAME_MAX
33429	Maximum length of a login name.
33430	Minimum Acceptable Value: _POSIX_LOGIN_NAME_MAX
33431 RT	MQ_OPEN_MAX
33432	The maximum number of open message queue descriptors a process may hold.
33433	Minimum Acceptable Value: _POSIX_MQ_OPEN_MAX
33434	MQ_PRIO_MAX
33435	The maximum number of message priorities supported by the implementation.
33436	Minimum Acceptable Value: _POSIX_MQ_PRIO_MAX
33437	OPEN_MAX
33438	Maximum number of files that one process can have open at any one time.
33439 FIPS	Minimum Acceptable Value: 20
33440 EX	PAGESIZE
33441	Size in bytes of a page.
33442	Minimum Acceptable Value: 1
33443	PAGE_SIZE
33444	Same as PAGESIZE. If either PAGESIZE or PAGE_SIZE is defined, the other will be defined
33445	with the same value.
33446	PASS_MAX
33447	Maximum number of significant bytes in a password (not including terminating null).
33448	(LEGACY)
33449	Minimum Acceptable Value: 8
33450	PTHREAD_DESTRUCTOR_ITERATIONS
33451	Maximum number of attempts made to destroy a thread's thread-specific data values on
33452	thread exit.
33453	Minimum Acceptable Value: _POSIX_THREAD_DESTRUCTOR_ITERATIONS

33454	PTHREAD_KEYS_MAX	
33455	Maximum number of data keys that can be created by a process.	
33456	Minimum Acceptable Value: _POSIX_THREAD_KEYS_MAX	
33457	PTHREAD_STACK_MIN	
33458	Minimum size in bytes of thread stack storage.	
33459	Minimum Acceptable Value: 0	
33460	PTHREAD_THREADS_MAX	
33461	Maximum number of threads that that can be created per process.	
33462	Minimum Acceptable Value: _POSIX_THREAD_THREADS_MAX	
33463 RT	RTSIG_MAX	
33464	Maximum number of realtime signals reserved for application use in this implementation.	
33465	Minimum Acceptable Value: _POSIX_RTSIG_MAX	
33466	SEM_NSEMS_MAX	
33467	Maximum number of semaphores that a process may have.	
33468	Minimum Acceptable Value: _POSIX_SEM_NSEMS_MAX	
33469	SEM_VALUE_MAX	
33470	The maximum value a semaphore may have.	
33471	Minimum Acceptable Value: _POSIX_SEM_VALUE_MAX	
33472	SIGQUEUE_MAX	
33473	Maximum number of queued signals that a process may send and have pending at the	
33474	receiver(s) at any time.	
33475	Minimum Acceptable Value: _POSIX_SIGQUEUE_MAX	
33476	STREAM_MAX	
33477	The number of streams that one process can have open at one time. If defined, it has the	
33478	same value as {FOPEN_MAX} (see <stdio.h>).	
33479	Minimum Acceptable Value: _POSIX_STREAM_MAX	
33480 RT	TIMER_MAX	
33481	Maximum number of timers per-process supported by the implementation.	
33482	Minimum Acceptable Value: _POSIX_TIMER_MAX	
33483	TTY_NAME_MAX	
33484	Maximum length of terminal device name.	
33485	Minimum Acceptable Value: _POSIX_TTY_NAME_MAX	
33486	TZNAME_MAX	
33487	Maximum number of bytes supported for the name of a time zone (not of the TZ variable).	
33488	Minimum Acceptable Value: _POSIX_TZNAME_MAX	
33489	Pathname Variable Values	
33490	The values in the following list may be constants within an implementation or may vary from	
33491	one pathname to another. For example, file systems or directories may have different	
33492	characteristics.	
33493	A definition of one of the values will be omitted from the <limits.h> header on specific	
33494	implementations where the corresponding value is equal to or greater than the stated minimum,	
33495	but where the value can vary depending on the file to which it is applied. The actual value	
33496	supported for a specific pathname will be provided by the <i>pathconf()</i> function.	
33497 EX	FILESIZEBITS	
33498	Minimum number of bits needed to represent, as a signed integer value, the maximum size	

33499	of a regular file allowed in the specified directory.
33500	Minimum Acceptable Value: 32
33501	LINK_MAX
33502	Maximum number of links to a single file.
33503	Minimum Acceptable Value: _POSIX_LINK_MAX
33504	MAX_CANON
33505	Maximum number of bytes in a terminal canonical input line.
33506	Minimum Acceptable Value: _POSIX_MAX_CANON
33507	MAX_INPUT
33508	Minimum number of bytes for which space will be available in a terminal input queue;
33509	therefore, the maximum number of bytes a portable application may require to be typed as
33510	input before reading them.
33511	Minimum Acceptable Value: _POSIX_MAX_INPUT
33512	NAME_MAX
33513	Maximum number of bytes in a filename (not including terminating null).
33514	Minimum Acceptable Value: _POSIX_NAME_MAX
33515	PATH_MAX
33516	Maximum number of bytes in a pathname, including the terminating null character.
33517	Minimum Acceptable Value: _POSIX_PATH_MAX
33518	PIPE_BUF
33519	Maximum number of bytes that is guaranteed to be atomic when writing to a pipe.
33520	Minimum Acceptable Value: _POSIX_PIPE_BUF
33521	Run-time Increasable Values
33522	The magnitude limitations in the following list will be fixed by specific implementations. An
33523	application should assume that the value supplied by <limits.h> in a specific implementation is
33524	the minimum that pertains whenever the application is run under that implementation. A
33525	specific instance of a specific implementation may increase the value relative to that supplied by
33526	<limits.h> for that implementation. The actual value supported by a specific instance will be
33527	provided by the <i>sysconf()</i> function.
33528	BC_BASE_MAX
33529	Maximum <i>obase</i> values allowed by the <i>bc</i> utility.
33530	Minimum Acceptable Value: _POSIX2_BC_BASE_MAX
33531	BC_DIM_MAX
33532	Maximum number of elements permitted in an array by the <i>bc</i> utility.
33533	Minimum Acceptable Value: _POSIX2_BC_DIM_MAX
33534	BC_SCALE_MAX
33535	Maximum <i>scale</i> value allowed by the <i>bc</i> utility.
33536	Minimum Acceptable Value: _POSIX2_BC_SCALE_MAX
33537	BC_STRING_MAX
33538	Maximum length of a string constant accepted by the <i>bc</i> utility.
33539	Minimum Acceptable Value: _POSIX2_BC_STRING_MAX
33540	COLL_WEIGHTS_MAX
33541	Maximum number of weights that can be assigned to an entry of the LC_COLLATE order
33542	keyword in the locale definition file; see the XBD specification, Chapter 5, Locale .
33543	Minimum Acceptable Value: _POSIX2_COLL_WEIGHTS_MAX

33544	EXPR_NEST_MAX	
33545	Maximum number of expressions that can be nested within parentheses by the <i>expr</i> utility.	
33546	Minimum Acceptable Value: _POSIX2_EXPR_NEST_MAX	
33547	LINE_MAX	
33548	Unless otherwise noted, the maximum length, in bytes, of a utility's input line (either	
33549	standard input or another file), when the utility is described as processing text files. The	
33550	length includes room for the trailing newline.	
33551	Minimum Acceptable Value: _POSIX2_LINE_MAX	
33552 FIPS	NGROUPS_MAX	
33553	Maximum number of simultaneous supplementary group IDs per process.	
33554	Minimum Acceptable Value: 8	
33555	RE_DUP_MAX	
33556	Maximum number of repeated occurrences of a regular expression permitted when using	
33557	the interval notation $\{m, n\}$; see the XBD specification, Chapter 7, Regular Expressions .	
33558	Minimum Acceptable Value: _POSIX2_RE_DUP_MAX	
33559	Maximum Values	
33560 RT	The symbolic constants in the following list are defined in <limits.h> with the values shown.	
33561	These are symbolic names for the most restrictive value for certain features on a system	
33562	supporting the Realtime Feature Group. A conforming implementation will provide values no	
33563	larger than these values. A portable application will not require a smaller value for correct	
33564	operation.	
33565	_POSIX_CLOCKRES_MIN	
33566	The CLOCK_REALTIME clock resolution, in nanoseconds	
33567	Value: 20 000 000	
33568		
33569	Minimum Values	
33570	The symbolic constants in the following list are defined in <limits.h> with the values shown.	
33571	These are symbolic names for the most restrictive value for certain features on a system	
33572	conforming to this specification. Related symbolic constants are defined elsewhere in this	
33573	specification which reflect the actual implementation and which need not be as restrictive. A	
33574	conforming implementation will provide values at least this large. A portable application must	
33575	not require a larger value for correct operation.	
33576 RT	_POSIX_AIO_LISTIO_MAX	
33577	The number of I/O operations that can be specified in a list I/O call.	
33578	Value: 2	
33579	_POSIX_AIO_MAX	
33580	The number of outstanding asynchronous I/O operations.	
33581	Value: 1	
33582	_POSIX_ARG_MAX	
33583	Maximum length of argument to the <i>exec</i> functions including environment data.	
33584	Value: 4 096	
33585	_POSIX_CHILD_MAX	
33586	Maximum number of simultaneous processes per real user ID.	
33587	Value: 6	

33588	RT	_POSIX_DELAYTIMER_MAX
33589		The number of timer expiration overruns.
33590		Value: 32
33591		_POSIX_LINK_MAX
33592		Maximum number of links to a single file.
33593		Value: 8
33594		_POSIX_LOGIN_NAME_MAX
33595		The size of the storage required for a login name, in bytes, including the terminating null.
33596		Value: 9
33597		_POSIX_MAX_CANON
33598		Maximum number of bytes in a terminal canonical input queue.
33599		Value: 255
33600		_POSIX_MAX_INPUT
33601		Maximum number of bytes allowed in a terminal input queue.
33602		Value: 255
33603	RT	_POSIX_MQ_OPEN_MAX
33604		The number of message queues that can be open for a single process.
33605		Value: 8
33606		_POSIX_MQ_PRIO_MAX
33607		The maximum number of message priorities supported by the implementation.
33608		Value: 32
33609		_POSIX_NAME_MAX
33610		Maximum number of bytes in a filename (not including terminating null).
33611		Value: 14
33612		_POSIX_NGROUPS_MAX
33613		Maximum number of simultaneous supplementary group IDs per process.
33614		Value: 0
33615		_POSIX_OPEN_MAX
33616		Maximum number of files that one process can have open at any one time.
33617		Value: 16
33618		_POSIX_PATH_MAX
33619		Maximum number of bytes in a pathname.
33620		Value: 255
33621		_POSIX_PIPE_BUF
33622		Maximum number of bytes that is guaranteed to be atomic when writing to a pipe.
33623		Value: 512
33624	RT	_POSIX_RTSIG_MAX
33625		The number of realtime signal numbers reserved for application use.
33626		Value: 8
33627		_POSIX_SEM_NSEMS_MAX
33628		The number of semaphores that a process may have.
33629		Value: 256
33630		_POSIX_SEM_VALUE_MAX
33631		The maximum value a semaphore may have.
33632		Value: 32 767

33633	_POSIX_SIGQUEUE_MAX	
33634	The number of queued signals that a process may send and have pending at the receiver(s)	
33635	at any time.	
33636	Value: 32	
33637	_POSIX_SSIZE_MAX	
33638	The value that can be stored in an object of type ssize_t .	
33639	Value: 32 767	
33640	_POSIX_STREAM_MAX	
33641	The number of streams that one process can have open at one time.	
33642	Value: 8	
33643	_POSIX_THREAD_DESTRUCTOR_ITERATIONS	
33644	The number of attempts made to destroy a thread's thread-specific data values on thread	
33645	exit.	
33646	Value: 4	
33647	_POSIX_THREAD_KEYS_MAX	
33648	The number of data keys per process.	
33649	Value: 128	
33650	_POSIX_THREAD_THREADS_MAX	
33651	The number of threads per process.	
33652	Value: 64	
33653 RT	_POSIX_TIMER_MAX	
33654	The per process number of timers.	
33655	Value: 32	
33656	_POSIX_TTY_NAME_MAX	
33657	The size of the storage required for a terminal device name, in bytes, including the	
33658	terminating null.	
33659	Value: 9	
33660	_POSIX_TZNAME_MAX	
33661	Maximum number of bytes supported for the name of a time zone (not of TZ variable).	
33662	Value: 3	
33663	_POSIX2_BC_BASE_MAX	
33664	Maximum <i>obase</i> values allowed by the <i>bc</i> utility.	
33665	Value: 99	
33666	_POSIX2_BC_DIM_MAX	
33667	Maximum number of elements permitted in an array by the <i>bc</i> utility.	
33668	Value: 2 048	
33669	_POSIX2_BC_SCALE_MAX	
33670	Maximum <i>scale</i> value allowed by the <i>bc</i> utility.	
33671	Value: 99	
33672	_POSIX2_BC_STRING_MAX	
33673	Maximum length of a string constant accepted by the <i>bc</i> utility.	
33674	Value: 1 000	
33675	_POSIX2_COLL_WEIGHTS_MAX	
33676	Maximum number of weights that can be assigned to an entry of the LC_COLLATE order	
33677	keyword in the locale definition file; see the XBD specification, Chapter 5, Locale .	
33678	Value: 2	

33679	<code>_POSIX2_EXPR_NEST_MAX</code>	
33680	Maximum number of expressions that can be nested within parentheses by the <i>expr</i> utility.	
33681	Value: 32	
33682	<code>_POSIX2_LINE_MAX</code>	
33683	Unless otherwise noted, the maximum length, in bytes, of a utility's input line (either	
33684	standard input or another file), when the utility is described as processing text files. The	
33685	length includes room for the trailing newline.	
33686	Value: 2 048	
33687	<code>_POSIX2_RE_DUP_MAX</code>	
33688	Maximum number of repeated occurrences of a regular expression permitted when using	
33689	the interval notation $\{m, n\}$; see the XBD specification, Chapter 7, Regular Expressions .	
33690	Value: 255	
33691 EX	<code>_XOPEN_IOV_MAX</code>	
33692	Maximum number of iovec structures that one process has available for use with <i>readv()</i> or	
33693	<i>writev()</i> .	
33694	Value: 16	
33695		
33696	Numerical Limits	
33697	The values in the following lists are defined in <limits.h> and will be constant expressions	
33698 EX	suitable for use in #if preprocessing directives. Moreover, except for CHAR_BIT, DBL_DIG,	
33699	DBL_MAX, FLT_DIG, FLT_MAX, LONG_BIT, WORD_BIT and MB_LEN_MAX, the symbolic	
33700	names will be defined as expressions of the correct type.	
33701	If the value of an object of type char is treated as a signed integer when used in an expression,	
33702	the value of CHAR_MIN is the same as that of SCHAR_MIN and the value of CHAR_MAX is	
33703	the same as that of SCHAR_MAX. Otherwise, the value of CHAR_MIN is 0 and the value of	
33704	CHAR_MAX is the same as that of UCHAR_MAX.	
33705	CHAR_BIT	
33706	Number of bits in a type char .	
33707	Minimum Acceptable Value: 8	
33708	CHAR_MAX	
33709	Maximum value of a type char .	
33710	Minimum Acceptable Value: UCHAR_MAX or SCHAR_MAX	
33711 EX	DBL_DIG	
33712	Digits of precision of a type double . (LEGACY)	
33713	Minimum Acceptable Value: 10	
33714	DBL_MAX	
33715	Maximum value of a type double . (LEGACY)	
33716	Minimum Acceptable Value: 1E +37	
33717	FLT_DIG	
33718	Digits of precision of a type float . (LEGACY)	
33719	Minimum Acceptable Value: 6	
33720	FLT_MAX	
33721	Maximum value of a float . (LEGACY)	
33722	Minimum Acceptable Value: 1E+37	

33723	INT_MAX	
33724	Maximum value of an int .	
33725	Minimum Acceptable Value: 2 147 483 647	
33726 EX	LONG_BIT	
33727	Number of bits in a long int .	
33728	Minimum Acceptable Value: 32	
33729	LONG_MAX	
33730	Maximum value of a long int .	
33731	Minimum Acceptable Value: +2 147 483 647	
33732	MB_LEN_MAX	
33733	Maximum number of bytes in a character, for any supported locale.	
33734	Minimum Acceptable Value: 1	
33735	SCHAR_MAX	
33736	Maximum value of a type signed char .	
33737	Minimum Acceptable Value: +127	
33738	SHRT_MAX	
33739	Maximum value of a type short .	
33740	Minimum Acceptable Value: +32 767	
33741	SSIZE_MAX	
33742	Maximum value of an object of type ssize_t .	
33743	Minimum Acceptable Value: _POSIX_SSIZE_MAX	
33744	UCHAR_MAX	
33745	Maximum value of a type unsigned char .	
33746	Minimum Acceptable Value: 255	
33747	UINT_MAX	
33748	Maximum value of a type unsigned int .	
33749	Minimum Acceptable Value: 4 294 967 295	
33750	ULONG_MAX	
33751	Maximum value of a type unsigned long int .	
33752	Minimum Acceptable Value: 4 294 967 295	
33753	USHRT_MAX	
33754	Maximum value for a type unsigned short int .	
33755	Minimum Acceptable Value: 65 535	
33756 EX	WORD_BIT	
33757	Number of bits in a word or type int .	
33758	Minimum Acceptable Value: 16	
33759	CHAR_MIN	
33760	Minimum value of a type char .	
33761	Minimum Acceptable Value: SCHAR_MIN or 0	
33762	INT_MIN	
33763	Minimum value of a type int .	
33764	Minimum Acceptable Value: -2 147 483 647	
33765	LONG_MIN	
33766	Minimum value of a type long int .	
33767	Minimum Acceptable Value: -2 147 483 647	

33768	SCHAR_MIN	
33769	Minimum value of a type signed char .	
33770	Minimum Acceptable Value: -127	
33771	SHRT_MIN	
33772	Minimum value of a type short .	
33773	Minimum Acceptable Value: -32 767	
33774	Other Invariant Values	
33775	The following constants are defined on all systems in <limits.h>.	
33776 EX	CHARCLASS_NAME_MAX	
33777	Maximum number of bytes in a character class name.	
33778	Minimum Acceptable Value: 14	
33779	NL_ARGMAX	
33780	Maximum value of <i>digit</i> in calls to the <i>printf()</i> and <i>scanf()</i> functions.	
33781	Minimum Acceptable Value: 9	
33782	NL_LANGMAX	
33783	Maximum number of bytes in a <i>LANG</i> name.	
33784	Minimum Acceptable Value: 14	
33785	NL_MSGMAX	
33786	Maximum message number.	
33787	Minimum Acceptable Value: 32 767	
33788	NL_NMAX	
33789	Maximum number of bytes in an N-to-1 collation mapping.	
33790	Minimum Acceptable Value: *	
33791	NL_SETMAX	
33792	Maximum set number.	
33793	Minimum Acceptable Value: 255	
33794	NL_TEXTMAX	
33795	Maximum number of bytes in a message string.	
33796	Minimum Acceptable Value: _POSIX2_LINE_MAX	
33797	NZERO	
33798	Default process priority.	
33799	Minimum Acceptable Value: 20	
33800	TMP_MAX	
33801	Minimum number of unique pathnames generated by <i>tmpnam()</i> . Maximum number of	
33802	times an application can call <i>tmpnam()</i> reliably. (LEGACY)	
33803	Minimum Acceptable Value: 10 000	
33804		
33805	APPLICATION USAGE	
33806	None.	
33807	FUTURE DIRECTIONS	
33808	None.	
33809	SEE ALSO	
33810	<i>fpathconf()</i> , <i>pathconf()</i> , <i>sysconf()</i> .	

33811 CHANGE HISTORY

33812 First released in Issue 1.

33813 Issue 4

33814 This entry is largely restructured to improve symbol grouping. A great many symbols, too
33815 numerous to mention, have also been added for alignment with the ISO POSIX-2 standard.

33816 The following changes are incorporated for alignment with the ISO C standard:

- 33817 • The constants INT_MIN, LONG_MIN and SHRT_MIN are changed from values ending in 8
33818 to ones ending in 7.
- 33819 • The DBL_DIG, DBL_MAX, FLT_DIG and FLT_MAX symbols are marked both as extensions
33820 and **LEGACY**.
- 33821 • The LONG_BIT and WORD_BIT symbols are marked as extensions.
- 33822 • The DBL_MIN and FLT_MIN symbols are withdrawn.
- 33823 • Text introducing numerical limits now indicates that they will be constant expressions
33824 suitable for use in #if preprocessing directives.

33825 The following change is incorporated for alignment with the FIPS requirements:

- 33826 • The minimum acceptable value for NGROUPS_MAX is changed from
33827 _POSIX_NGROUPS_MAX to 8. This is marked as as extension.

33828 Other changes are incorporated as follows:

- 33829 • A sentence is added to the DESCRIPTION indicating that names beginning with _POSIX can
33830 be found in <unistd.h>.
- 33831 • The PASS_MAX and TMP_MAX symbols are marked **LEGACY**.
- 33832 • Use of the terms “bytes” and “characters” is rationalised to make it clear when the
33833 description is referring to either single-byte values or possibly multi-byte characters.
- 33834 • CHARCLASS_NAME_MAX is added to the list of **Other Invariant Values** and marked as an
33835 extension.

33836 Issue 4, Version 2

33837 The DESCRIPTION is revised for X/OPEN UNIX conformance as follows:

- 33838 • Under **Run-time Invariant Values**, ATEXT_MAX, IOV_MAX, PAGESIZE and PAGE_SIZE
33839 are added.
- 33840 • Under **Minimum Values**, _XOPEN_IOV_MAX is added.

33841 Issue 5

33842 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
33843 Threads Extension.

33844 FILESIZEBITS added for the Large File Summit extensions.

33845 The minimum acceptable values for INT_MAX, INT_MIN and UINT_MAX are changed to make
33846 32-bit values the minimum requirement.

33847 The entry is restructured to improve readability.

33848 **NAME**

33849 locale.h — category macros

33850 **SYNOPSIS**

33851 #include <locale.h>

33852 **DESCRIPTION**

33853 The <locale.h> header provides a definition for structure **lconv**, which includes at least the
 33854 following members. (See the definitions of LC_MONETARY in the **XBD** specification, **Section**
 33855 **5.3.3, LC_MONETARY**, and the **XBD** specification, **Section 5.3.4, LC_NUMERIC**.)

```

33856     char    *currency_symbol
33857     char    *decimal_point
33858     char    frac_digits
33859     char    *grouping
33860     char    *int_curr_symbol
33861     char    int_frac_digits
33862     char    *mon_decimal_point
33863     char    *mon_grouping
33864     char    *mon_thousands_sep
33865     char    *negative_sign
33866     char    n_cs_precedes
33867     char    n_sep_by_space
33868     char    n_sign_posn
33869     char    *positive_sign
33870     char    p_cs_precedes
33871     char    p_sep_by_space
33872     char    p_sign_posn
33873     char    *thousands_sep

```

33874 The <locale.h> header defines NULL (as defined in <stddef.h>) and at least the following as
 33875 macros:

```

33876     LC_ALL
33877     LC_COLLATE
33878     LC_CTYPE
33879     LC_MESSAGES
33880     LC_MONETARY
33881     LC_NUMERIC
33882     LC_TIME

```

33883 which expand to distinct integral-constant expressions, for use as the first argument to the
 33884 *setlocale()* function.

33885 Additional macro definitions, beginning with the characters LC_ and an upper-case letter, may
 33886 also be given here.

33887 The following are declared as functions and may also be defined as macros. Function prototypes
 33888 must be provided for use with an ISO C compiler.

```

33889     struct lconv *localeconv (void);
33890     char    setlocale(int, const char *);

```

33891 **APPLICATION USAGE**

33892 None.

33893 **FUTURE DIRECTIONS**

33894 None.

33895 **SEE ALSO**

33896 *localeconv()*, *setlocale()*, the **XBD** specification, **Chapter 6, Environment Variables**.

33897 **CHANGE HISTORY**

33898 First released in Issue 3.

33899 Entry included for alignment with the ISO C standard.

33900 **Issue 4**

33901 The following changes are incorporated for alignment with the ISO C standard:

- 33902 • The function declarations in this header are expanded to full ISO C prototypes.
- 33903 • The definition of **struct lconv** is added.
- 33904 • A reference to <**stddef.h**> is added for the definition of NULL.

33905 NAME

33906 math.h — mathematical declarations

33907 SYNOPSIS

33908 #include <math.h>

33909 DESCRIPTION

33910 The <math.h> header provides for the following constants. The values are of type **double** and
 33911 are accurate within the precision of the **double** type.

33912 EX	M_E	Value of e
33913	M_LOG2E	Value of $\log_2 e$
33914	M_LOG10E	Value of $\log_{10} e$
33915	M_LN2	Value of $\log_e 2$
33916	M_LN10	Value of $\log_e 10$
33917	M_PI	Value of π
33918	M_PI_2	Value of $\pi/2$
33919	M_PI_4	Value of $\pi/4$
33920	M_1_PI	Value of $1/\pi$
33921	M_2_PI	Value of $2/\pi$
33922	M_2_SQRTPI	Value of $2/\sqrt{\pi}$
33923	M_SQRT2	Value of $\sqrt{2}$
33924	M_SQRT1_2	Value of $1/\sqrt{2}$

33925 The header defines the following symbolic constants:

33926 EX	MAXFLOAT	Value of maximum non-infinite single-precision floating point number.
33927	HUGE_VAL	A positive double expression, not necessarily representable as a float . Used
33928		as an error value returned by the mathematics library. HUGE_VAL evaluates
33929		to $+\infty$ on systems supporting the ANSI/IEEE Std 754:1985 standard.

33930 The following are declared as functions and may also be defined as macros. Function prototypes
 33931 must be provided for use with an ISO C compiler.

```

33932 double acos(double);
33933 double asin(double);
33934 double atan(double);
33935 double atan2(double, double);
33936 double ceil(double);
33937 double cos(double);
33938 double cosh(double);
33939 double exp(double);
33940 double fabs(double);
33941 double floor(double);
33942 double fmod(double, double);
33943 double frexp(double, int *);
33944 double ldexp(double, int);
33945 double log(double);
33946 double log10(double);
33947 double modf(double, double *);
33948 double pow(double, double);
33949 double sin(double);
33950 double sinh(double);
33951 double sqrt(double);
33952 double tan(double);
33953 double tanh(double);

```

```

33954 EX    double erf(double);
33955        double erfc(double);
33956        double gamma(double);
33957        double hypot(double, double);
33958        double j0(double);
33959        double j1(double);
33960        double jn(int, double);
33961        double lgamma(double);
33962        double y0(double);
33963        double y1(double);
33964        double yn(int, double);
33965        int isnan(double);
33966        double acosh(double);
33967        double asinh(double);
33968        double atanh(double);
33969        double cbrt(double);
33970        double expm1(double);
33971        int ilogb(double);
33972        double log1p(double);
33973        double logb(double);
33974        double nextafter(double, double);
33975        double remainder(double, double);
33976        double rint(double);
33977        double scalb(double, double);
33978

```

33979 The following external variable is defined:

```

33980 EX    extern int signgam;
33981

```

33982 APPLICATION USAGE

33983 None.

33984 FUTURE DIRECTIONS

33985 None.

33986 SEE ALSO

33987 *acos()*, *acosh()*, *asin()*, *atan()*, *atan2()*, *cbrt()*, *ceil()*, *cos()*, *cosh()*, *erf()*, *exp()*, *expm1()*, *fabs()*,
33988 *floor()*, *fmod()*, *frexp()*, *hypot()*, *ilogb()*, *isnan()*, *j0()*, *ldexp()*, *lgamma()*, *log()*, *log10()*, *log1p()*,
33989 *logb()*, *modf()*, *nextafter()*, *pow()*, *remainder()*, *rint()*, *scalb()*, *sin()*, *sinh()*, *sqrt()*, *tan()*, *tanh()*,
33990 *y0()*.

33991 CHANGE HISTORY

33992 First released in Issue 1.

33993 Issue 4

33994 The following changes are incorporated for alignment with the ISO C standard:

- 33995 • The description of HUGE_VAL is changed to indicate that this value is not necessarily
- 33996 representable as a **float**.
- 33997 • The function declarations in this header are expanded to full ISO C prototypes.

33998 Other changes are incorporated as follows:

- 33999 • The constants M_E and MAXFLOAT are marked as extensions.

34000 • The functions declared in this header are subdivided into those defined in the ISO C
34001 standard, and those defined only by X/Open. Functions in the latter group are marked as
34002 extensions, as is the external variable *signgam*.

34003 **Issue 4, Version 2**

34004 The following change is incorporated for X/OPEN UNIX conformance:

34005 • The *acosh()*, *asinh()*, *atanh()*, *cbrt()*, *expm1()*, *ilogb()*, *log1p()*, *logb()*, *nextafter()*, *remainder()*,
34006 *rint()* and *scalb()* functions are added to the list of functions declared in this header.

34007 **NAME**

34008 monetary.h — monetary types

34009 **SYNOPSIS**

34010 EX #include <monetary.h>

34011

34012 **DESCRIPTION**

34013 The <monetary.h> header defines the following data types through typedef:

34014 **size_t** As described in <stddef.h>.

34015 **ssize_t** As described in <sys/types.h>.

34016 The following is declared as a function and may also be defined as a macro. Function prototypes
34017 must be provided for use with an ISO C compiler.

34018 ssize_t strfmon(char *, size_t, const char *, ...);

34019 **APPLICATION USAGE**

34020 None.

34021 **FUTURE DIRECTIONS**

34022 None.

34023 **SEE ALSO**

34024 strfmon().

34025 **CHANGE HISTORY**

34026 First released in Issue 4.

34027 **NAME**34028 mqueue.h — message queues (**REALTIME**)34029 **SYNOPSIS**

34030 RT #include <mqueue.h>

34031

34032 **DESCRIPTION**

34033 The <mqueue.h> header defines the **mqd_t** type, which is used for message queue descriptors.
 34034 This will not be an array type. A message queue descriptor may be implemented using a file
 34035 descriptor, in which case applications can open up to at least {OPEN_MAX} file and message
 34036 queues.

34037 The <mqueue.h> header defines the **sigevent** structure (as described in <signal.h>) and the
 34038 **mq_attr** structure, which is used in getting and setting the attributes of a message queue.
 34039 Attributes are initially set when the message queue is created. A **mq_attr** structure will have at
 34040 least the following fields:

34041	long	mq_flags	message queue flags
34042	long	mq_maxmsg	maximum number of messages
34043	long	mq_msgsize	maximum message size
34044	long	mq_curmsgs	number of messages currently queued

34045 The following are declared as functions and may also be declared as macros. Function
 34046 prototypes must be provided for use with an ISO C compiler.

```

34047       int      mq_close(mqd_t);
34048       int      mq_getattr(mqd_t, struct mq_attr *);
34049       int      mq_notify(mqd_t, const struct sigevent *);
34050       mqd_t    mq_open(const char *, int, ...);
34051       ssize_t  mq_receive(mqd_t, char *, size_t, unsigned int *);
34052       int      mq_send(mqd_t, const char *, size_t, unsigned int);
34053       int      mq_setattr(mqd_t, const struct mq_attr *, struct mq_attr *);
34054       int      mq_unlink(const char *);

```

34055 Inclusion of the <mqueue.h> header may make visible symbols defined in the headers <fcntl.h>,
 34056 <signal.h>, <sys/types.h> and <time.h>.

34057 **APPLICATION USAGE**

34058 None.

34059 **FUTURE DIRECTIONS**

34060 None.

34061 **SEE ALSO**

34062 <fcntl.h>, <signal.h>, <sys/types.h>, <time.h>.

34063 **CHANGE HISTORY**

34064 First released in Issue 5.

34065 Included for alignment with the POSIX Realtime Extension.

34066 NAME

34067 ndbm.h — definitions for ndbm database operations

34068 SYNOPSIS

34069 EX `#include <ndbm.h>`

34070

34071 DESCRIPTION

34072 The <ndbm.h> header defines the **datum** type as a structure that includes at least the following
34073 members:

34074 **void *dptr** A pointer to the application's data
34075 **size_t dsize** The size of the object pointed to by **dptr**

34076 The **size_t** type is defined through **typedef** as described in <stddef.h>.

34077 The <ndbm.h> header defines the **DBM** type through **typedef**.

34078 The following constants are defined as possible values for the *store_mode* argument to
34079 *dbm_store()*:

34080 **DBM_INSERT** Insertion of new entries only
34081 **DBM_REPLACE** Allow replacing existing entries

34082 The following are declared as functions and may also be defined as macros. Function prototypes
34083 must be provided for use with an ISO C compiler.

```
34084 int      dbm_clearerr(DBM *);
34085 void     dbm_close(DBM *);
34086 int      dbm_delete(DBM *, datum);
34087 int      dbm_error(DBM *);
34088 datum    dbm_fetch(DBM *, datum);
34089 datum    dbm_firstkey(DBM *);
34090 datum    dbm_nextkey(DBM *);
34091 DBM      *dbm_open(const char *, int, mode_t);
34092 int      dbm_store(DBM *, datum, datum, int);
```

34093 The **mode_t** type is defined through **typedef** as described in <sys/types.h>.

34094 APPLICATION USAGE

34095 None.

34096 FUTURE DIRECTIONS

34097 None.

34098 SEE ALSO

34099 *dbm_clearerr()*.

34100 CHANGE HISTORY

34101 First released in Issue 4, Version 2.

34102 Issue 5

34103 References to the definitions of **size_t** and **mode_t** are added to the DESCRIPTION.

34104 **NAME**

34105 nl_types.h — data types

34106 **SYNOPSIS**

34107 EX #include <nl_types.h>

34108

34109 **DESCRIPTION**

34110 The <nl_types.h> header contains definitions of at least the following types:

34111 **nl_catd** Used by the message catalogue functions *catopen()*, *catgets()* and
34112 *catclose()* to identify a catalogue descriptor.34113 **nl_item** Used by *nl_langinfo()* to identify items of *langinfo* data. Values of objects
34114 of type **nl_item** are defined in <langinfo.h>.

34115 The <nl_types.h> header contains definitions of at least the following constants:

34116 **NL_SETD** Used by *gencat* when no *\$set* directive is specified in a message text source
34117 file, see the **Internationalisation Guide, Chapter 3, The Message System**.
34118 This constant can be passed as the value of *set_id* on subsequent calls to
34119 *catgets()* (that is, to retrieve messages from the default message set). The
34120 value of **NL_SETD** is implementation-dependent.34121 **NL_CAT_LOCALE** Value that must be passed as the *offlag* argument to *catopen()* to ensure
34122 that message catalogue selection depends on the **LC_MESSAGES** locale
34123 category, rather than directly on the *LANG* environment variable.34124 The following are declared as functions and may also be defined as macros. Function prototypes
34125 must be provided for use with an ISO C compiler.34126 int catclose(nl_catd);
34127 char *catgets(nl_catd, int, int, const char *);
34128 nl_catd catopen(const char *, int);34129 **APPLICATION USAGE**

34130 None.

34131 **FUTURE DIRECTIONS**

34132 None.

34133 **SEE ALSO**34134 *catclose()*, *catgets()*, *catopen()*, *nl_langinfo()*, <langinfo.h>, the **XCU** specification, *gencat*.34135 **CHANGE HISTORY**

34136 First released in Issue 2.

34137 **Issue 4**

34138 The following change is incorporated for alignment with the ISO C standard:

- 34139
- The function declarations in this header are expanded to full ISO C prototypes.

34140 **NAME**

34141 poll.h — definitions for the poll() function

34142 **SYNOPSIS**

34143 EX #include <poll.h>

34144

34145 **DESCRIPTION**34146 The <poll.h> header defines the **pollfd** structure that includes at least the following member:

34147 int fd the following descriptor being polled

34148 short int events the input event flags (see below)

34149 short int revents the output event flags (see below)

34150 The <poll.h> header defines the following type through typedef:

34151 **nfds_t** An unsigned integral type used for the number of file descriptors.34152 The following symbolic constants are defined, zero or more of which may be OR-ed together to
34153 form the **events** or **revents** members in the **pollfd** structure:

34154 POLLIN Same effect as POLLRDNORM | POLLRDBAND.

34155 POLLRDNORM Data on priority band 0 may be read.

34156 POLLRDBAND Data on priority bands greater than 0 may be read.

34157 POLLPRI High priority data may be read.

34158 POLLOUT Same value as POLLWRNORM.

34159 POLLWRNORM Data on priority band 0 may be written.

34160 POLLWRBAND Data on priority bands greater than 0 may be written. This event only
34161 examines bands that have been written to at least once.34162 POLLERR An error has occurred (**revents** only).34163 POLLHUP Device has been disconnected (**revents** only).34164 POLLNVAL Invalid **fd** member (**revents** only).34165 The <poll.h> header declares the following function which may also be defined as a macro.
34166 Function prototypes must be provided for use with an ISO C compiler.

34167 int poll(struct pollfd[], nfds_t, int);

34168 **APPLICATION USAGE**

34169 None.

34170 **FUTURE DIRECTIONS**

34171 None.

34172 **SEE ALSO**

34173 poll().

34174 **CHANGE HISTORY**

34175 First released in Issue 4, Version 2.

34176 NAME

34177 pthread.h — threads

34178 SYNOPSIS

34179 #include <pthread.h>

34180 DESCRIPTION

34181 The <pthread.h> header defines the following symbols:

34182 PTHREAD_CANCEL_ASYNCHRONOUS

34183 PTHREAD_CANCEL_ENABLE

34184 PTHREAD_CANCEL_DEFERRED

34185 PTHREAD_CANCEL_DISABLE

34186 PTHREAD_CANCELED

34187 PTHREAD_COND_INITIALIZER

34188 PTHREAD_CREATE_DETACHED

34189 PTHREAD_CREATE_JOINABLE

34190 PTHREAD_EXPLICIT_SCHED

34191 PTHREAD_INHERIT_SCHED

34192 EX PTHREAD_MUTEX_DEFAULT

34193 PTHREAD_MUTEX_ERRORCHECK

34194 PTHREAD_MUTEX_NORMAL

34195 PTHREAD_MUTEX_INITIALIZER

34196 PTHREAD_MUTEX_RECURSIVE

34197 PTHREAD_ONCE_INIT

34198 RTT PTHREAD_PRIO_INHERIT

34199 PTHREAD_PRIO_NONE

34200 PTHREAD_PRIO_PROTECT

34201 PTHREAD_PROCESS_SHARED

34202 PTHREAD_PROCESS_PRIVATE

34203 EX PTHREAD_RWLOCK_INITIALIZER

34204 RTT PTHREAD_SCOPE_PROCESS

34205 PTHREAD_SCOPE_SYSTEM

34206

34207 EX The `pthread_attr_t`, `pthread_cond_t`, `pthread_condattr_t`, `pthread_key_t`, `pthread_mutex_t`,
 34208 `pthread_mutexattr_t`, `pthread_once_t`, `pthread_rwlock_t`, `pthread_rwlockattr_t` and `pthread_t`
 34209 types are defined as described in <sys/types.h>.

34210 The following are declared as functions and may also be declared as macros. Function
 34211 prototypes must be provided for use with an ISO C compiler.

34212 int pthread_attr_destroy(pthread_attr_t *);

34213 int pthread_attr_getdetachstate(const pthread_attr_t *, int *);

34214 EX int pthread_attr_getguardsize(const pthread_attr_t *, size_t *);

34215 RTT int pthread_attr_getinheritsched(const pthread_attr_t *, int *);

34216 int pthread_attr_getschedparam(const pthread_attr_t *,
 34217 struct sched_param *);

34218 RTT int pthread_attr_getschedpolicy(const pthread_attr_t *, int *);

34219 RTT int pthread_attr_getscope(const pthread_attr_t *, int *);

34220 int pthread_attr_getstackaddr(const pthread_attr_t *, void **);

34221 int pthread_attr_getstacksize(const pthread_attr_t *, size_t *);

34222 int pthread_attr_init(pthread_attr_t *);

34223 int pthread_attr_setdetachstate(pthread_attr_t *, int);

34224 EX

```

34225 int pthread_attr_setguardsize(pthread_attr_t *, size_t);
34226 RTT int pthread_attr_setinheritsched(pthread_attr_t *, int);
34227 int pthread_attr_setschedparam(pthread_attr_t *,
34228     const struct sched_param *);
34229 RTT int pthread_attr_setschedpolicy(pthread_attr_t *, int);
34230 int pthread_attr_setscope(pthread_attr_t *, int);
34231 int pthread_attr_setstackaddr(pthread_attr_t *, void *);
34232 int pthread_attr_setstacksize(pthread_attr_t *, size_t);
34233 int pthread_cancel(pthread_t);
34234 void pthread_cleanup_push(void (*)(void*), void *);
34235 void pthread_cleanup_pop(int);
34236 int pthread_cond_broadcast(pthread_cond_t *);
34237 int pthread_cond_destroy(pthread_cond_t *);
34238 int pthread_cond_init(pthread_cond_t *, const pthread_condattr_t *);
34239 int pthread_cond_signal(pthread_cond_t *);
34240 int pthread_cond_timedwait(pthread_cond_t *,
34241     pthread_mutex_t *, const struct timespec *);
34242 int pthread_cond_wait(pthread_cond_t *);
34243 int pthread_condattr_destroy(pthread_condattr_t *);
34244 int pthread_condattr_getpshared(const pthread_condattr_t *, int *);
34245 int pthread_condattr_init(pthread_condattr_t *);
34246 int pthread_condattr_setpshared(pthread_condattr_t *, int);
34247 int pthread_create(pthread_t *, const pthread_attr_t *,
34248     void (*)(void*), void *);
34249 int pthread_detach(pthread_t);
34250 int pthread_equal(pthread_t, pthread_t);
34251 void pthread_exit(void *);
34252 EX int pthread_getconcurrency(void);
34253 RTT int pthread_getschedparam(pthread_t, int *, struct sched_param *);
34254 void *pthread_getspecific(pthread_key_t);
34255 int pthread_join(pthread_t, void **);
34256 int pthread_key_create(pthread_key_t *, void (*)(void*));
34257 int pthread_key_delete(pthread_key_t);
34258 int pthread_mutex_destroy(pthread_mutex_t *);
34259 RTT int pthread_mutex_getprioceiling(const pthread_mutex_t *, int *);
34260 int pthread_mutex_init(pthread_mutex_t *, const pthread_mutexattr_t *);
34261 int pthread_mutex_lock(pthread_mutex_t *);
34262 RTT int pthread_mutex_setprioceiling(pthread_mutex_t *, int, int *);
34263 int pthread_mutex_trylock(pthread_mutex_t *);
34264 int pthread_mutex_unlock(pthread_mutex_t *);
34265 int pthread_mutexattr_destroy(pthread_mutexattr_t *);
34266 RTT int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *,
34267     int *);
34268 int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *, int *);
34269 int pthread_mutexattr_getpshared(const pthread_mutexattr_t *, int *);
34270 EX int pthread_mutexattr_gettype(pthread_mutexattr_t *, int *);
34271 int pthread_mutexattr_init(pthread_mutexattr_t *);
34272 RTT int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *, int);
34273 int pthread_mutexattr_setprotocol(pthread_mutexattr_t *, int);
34274 int pthread_mutexattr_setpshared(pthread_mutexattr_t *, int);
34275 EX int pthread_mutexattr_settype(pthread_mutexattr_t *, int);
34276 int pthread_once(pthread_once_t *, void (*)(void));

```

```

34277 EX    int    pthread_rwlock_destroy(pthread_rwlock_t *);
34278    int    pthread_rwlock_init(pthread_rwlock_t *,
34279        const pthread_rwlockattr_t *);
34280    int    pthread_rwlock_rdlock(pthread_rwlock_t *);
34281    int    pthread_rwlock_tryrdlock(pthread_rwlock_t *);
34282    int    pthread_rwlock_trywrlock(pthread_rwlock_t *);
34283    int    pthread_rwlock_unlock(pthread_rwlock_t *);
34284    int    pthread_rwlock_wrlock(pthread_rwlock_t *);
34285    int    pthread_rwlockattr_destroy(pthread_rwlockattr_t *);
34286    int    pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *,
34287        int *);
34288    int    pthread_rwlockattr_init(pthread_rwlockattr_t *);
34289    int    pthread_rwlockattr_setpshared(pthread_rwlockattr_t *, int);
34290    pthread_t
34291        pthread_self(void);
34292    int    pthread_setcancelstate(int, int *);
34293    int    pthread_setcanceltype(int, int *);
34294 EX    int    pthread_setconcurrency(int);
34295 RTT    int    pthread_setschedparam(pthread_t, int *,
34296        const struct sched_param *);
34297    int    pthread_setspecific(pthread_key_t, const void *);
34298    void    pthread_testcancel(void);

```

34299 EX Inclusion of the <pthread.h> header will make visible symbols defined in the headers <sched.h>
34300 and <time.h>.

34301 APPLICATION USAGE

34302 An interpretation request has been filed with IEEE PASC concerning requirements for visibility
34303 of symbols in this header.

34304 FUTURE DIRECTIONS

34305 None.

34306 SEE ALSO

34307 *pthread_attr_init()*, *pthread_attr_getguardsize()*, *pthread_attr_setscope()*, *pthread_cancel()*,
34308 *pthread_cleanup_push()*, *pthread_cond_init()*, *pthread_cond_signal()*, *pthread_cond_wait()*,
34309 *pthread_condattr_init()*, *pthread_create()*, *pthread_detach()*, *pthread_equal()*, *pthread_exit()*,
34310 *pthread_getconcurrency()*, *pthread_getschedparam()*, *pthread_join()*, *pthread_key_create()*,
34311 *pthread_key_delete()*, *pthread_mutex_init()*, *pthread_mutex_lock()*, *pthread_mutex_setprioceiling()*,
34312 *pthread_mutexattr_init()*, *pthread_mutexattr_gettype()*, *pthread_mutexattr_setprotocol()*,
34313 *pthread_once()*, *pthread_self()*, *pthread_setcancelstate()*, *pthread_setspecific()*, *pthread_rwlock_init()*,
34314 *pthread_rwlock_rdlock()*, *pthread_rwlock_unlock()*, *pthread_rwlock_wrlock()*,
34315 *pthread_rwlockattr_init()*, <sched.h>, <time.h>.

34316 CHANGE HISTORY

34317 First released in Issue 5.

34318 Included for alignment with the POSIX Threads Extension.

34319 NAME

34320 pwd.h — password structure

34321 SYNOPSIS

34322 #include <pwd.h>

34323 DESCRIPTION

34324 The <pwd.h> header provides a definition for **struct passwd**, which includes at least the
34325 following members:

34326	char	*pw_name	user's login name
34327	uid_t	pw_uid	numerical user ID
34328	gid_t	pw_gid	numerical group ID
34329	char	*pw_dir	initial working directory
34330	char	*pw_shell	program to use as shell

34331 EX The **gid_t** and **uid_t** types are defined as described in <sys/types.h>.

34332 The following are declared as functions and may also be defined as macros. Function prototypes
34333 must be provided for use with an ISO C compiler.

```
34334 struct passwd *getpwnam(const char *);
34335 struct passwd *getpwuid(uid_t);
34336 int           getpwnam_r(const char *, struct passwd *, char *,
34337                          size_t, struct passwd **);
34338 int           getpwuid_r(uid_t, struct passwd *, char *,
34339                          size_t, struct passwd **);
```

34340 EX void endpwent(void);

34341 struct passwd *getpwent(void);

34342 void setpwent(void);

34343

34344 APPLICATION USAGE

34345 None.

34346 FUTURE DIRECTIONS

34347 None.

34348 SEE ALSO

34349 endpwent(), getpwnam(), getpwuid(), getpwuid_r(), <sys/types.h>.

34350 CHANGE HISTORY

34351 First released in Issue 1.

34352 Issue 4

34353 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 34354 • The function declarations in this header are expanded to full ISO C prototypes.

34355 Another change is incorporated as follows:

- 34356 • Reference to the <sys/types.h> header is added for the definitions of **gid_t** and **uid_t**. This is
34357 marked as an extension.

34358 Issue 4, Version 2

34359 For X/OPEN UNIX conformance, the *getpwent()*, *endpwent()* and *setpwent()* functions are added
34360 to the list of functions declared in this header.

34361 Issue 5

34362 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

34363 **NAME**

34364 regex.h — regular-expression-matching types

34365 **SYNOPSIS**

34366 #include <regex.h>

34367 **DESCRIPTION**

34368 The <regex.h> header defines the structures and symbolic constants used by the *regcomp()*,
34369 *regex()*, *regerror()* and *regfree()* functions.

34370 The structure type **regex_t** contains at least the following member:

34371 size_t re_nsub number of parenthesised subexpressions

34372 The type **regoff_t** is defined as a signed arithmetic type that can hold the largest value that can
34373 be stored in either a type **off_t** or type **ssize_t**. The structure type **regmatch_t** contains at least
34374 the following members:

34375	regoff_t	rm_so	byte offset from start of <i>string</i>
34376			to start of substring
34377	regoff_t	rm_eo	byte offset from start of <i>string</i>
34378			of the first character after the end of substring

34379 Values for the *cflags* parameter to the *regcomp()* function:

34380	REG_EXTENDED	Use Extended Regular Expressions.
34381	REG_ICASE	Ignore case in match.
34382	REG_NOSUB	Report only success or fail in <i>regex()</i> .
34383	REG_NEWLINE	Change the handling of newline.

34384 Values for the *eflags* parameter to the *regex()* function:

34385	REG_NOTBOL	The circumflex character (^), when taken as a special character, will not
34386		match the beginning of <i>string</i> .
34387	REG_NOTEOL	The dollar sign (\$), when taken as a special character, will not match the
34388		end of <i>string</i> .

34389 The following constants are defined as error return values:

34390	REG_NOMATCH	<i>regex()</i> failed to match.
34391	REG_BADPAT	Invalid regular expression.
34392	REG_ECOLLATE	Invalid collating element referenced.
34393	REG_ECTYPE	Invalid character class type referenced.
34394	REG_EESCAPE	Trailing \ in pattern.
34395	REG_ESUBREG	Number in \digit invalid or in error.
34396	REG_EBRACK	[] imbalance.
34397	REG_EPAREN	\(\) or () imbalance.
34398	REG_EBRACE	\{ \} imbalance.
34399	REG_BADBR	Content of \{ \} invalid: not a number, number too large, more than two
34400		numbers, first larger than second.
34401	REG_ERANGE	Invalid endpoint in range expression.
34402	REG_ESPACE	Out of memory.
34403	REG_BADRPT	?, * or + not preceded by valid regular expression.
34404	REG_ENOSYS	The implementation does not support the function.

34405 The following are declared as functions and may also be declared as macros. Function
34406 prototypes must be provided for use with an ISO C compiler.

34407 int regcomp(regex_t *, const char *, int);
34408 int regexexec(const regex_t *, const char *, size_t, regmatch_t[], int);
34409 size_t regerror(int, const regex_t *, char *, size_t);
34410 void regfree(regex_t *);

34411 The implementation may define additional macros or constants using names beginning with
34412 REG_.

34413 **APPLICATION USAGE**

34414 None.

34415 **FUTURE DIRECTIONS**

34416 None.

34417 **SEE ALSO**

34418 *regcomp()*, the XCU specification.

34419 **CHANGE HISTORY**

34420 First released in Issue 4.

34421 Originally derived from the ISO POSIX-2 standard.

34422 **NAME**

34423 re_comp.h — regular-expression-matching functions for re_comp() (**LEGACY**)

34424 **SYNOPSIS**

34425 EX #include <re_comp.h>

34426

34427 **DESCRIPTION**

34428 The following are declared as functions and may also be declared as macros:

34429 char *re_comp(const char *string);

34430 int re_exec(const char *string);

34431 **APPLICATION USAGE**

34432 This header is kept for historical reasons. New applications should use the *regcomp()*, *regexexec()*,
34433 *regerror()* and *regfree()* functions, and the <regex.h> header, which provide full internationalised
34434 regular expression functionality compatible with the ISO POSIX-2 standard and the **XBD**
34435 specification, **Chapter 7, Regular Expressions**.

34436 **FUTURE DIRECTIONS**

34437 None.

34438 **SEE ALSO**

34439 *re_comp()*, <regex.h>.

34440 **CHANGE HISTORY**

34441 First released in Issue 4, Version 2.

34442 **Issue 5**

34443 Marked **LEGACY**.

34444 **NAME**34445 regex.h — regular-expression declarations (**LEGACY**)34446 **SYNOPSIS**

34447 EX #include <regex.h>

34448

34449 **DESCRIPTION**34450 In the <regex.h> header, each of the following is declared as a function, or defined as a macro,
34451 or both:

```

34452        int    advance(const char *string, const char *expbuf);
34453        char *compile(char *instring, char *expbuf, const char *endbuf,
34454                      int eof);
34455        int    step(const char *string, const char *expbuf);

```

34456 and the following are declared as external variables:

```

34457        extern char *loc1;
34458        extern char *loc2;
34459        extern char *locs;

```

34460 **APPLICATION USAGE**

34461 This header is kept for historical reasons. New applications should use the *regcomp()*, *regex()*,
 34462 *regerror()* and *regfree()* functions, and the <regex.h> header, which provide full internationalised
 34463 regular expression functionality compatible with the ISO POSIX-2 standard and the **XBD**
 34464 specification, **Chapter 7, Regular Expressions**.

34465 **FUTURE DIRECTIONS**

34466 None.

34467 **SEE ALSO**34468 *regex()*, <regex.h>.34469 **CHANGE HISTORY**

34470 First released in Issue 3.

34471 Entry derived from System V Release 2.0.

34472 **Issue 4**

34473 The following changes are incorporated in this issue:

- 34474 • The function declarations in this header are expanded to full ISO C prototypes.
- 34475 • The interface is marked TO BE WITHDRAWN.

34476 **Issue 5**

34477 Marked LEGACY.

34478 **NAME**34479 sched.h — execution scheduling (**REALTIME**)34480 **SYNOPSIS**

34481 RT #include <sched.h>

34482

34483 **DESCRIPTION**

34484 The <sched.h> header defines the **sched_param** structure, which contains the scheduling
 34485 parameters required for implementation of each supported scheduling policy. This structure
 34486 contains at least the following member:

34487 int sched_priority process execution scheduling priority

34488 Each process is controlled by an associated scheduling policy and priority. Associated with each
 34489 policy is a priority range. Each policy definition specifies the minimum priority range for that
 34490 policy. The priority ranges for each policy may overlap the priority ranges of other policies.

34491 Three scheduling policies are defined; others may be defined by the implementation. The three
 34492 standard policies are indicated by the values of the following symbolic constants:

34493 SCHED_FIFO First in-first out (FIFO) scheduling policy.

34494 SCHED_RR Round robin scheduling policy.

34495 SCHED_OTHER Another scheduling policy.

34496 The values of these constants are distinct.

34497 The following are declared as functions and may also be declared as macros. Function
 34498 prototypes must be provided for use with an ISO C compiler.

34499 int sched_get_priority_max(int);

34500 int sched_get_priority_min(int);

34501 int sched_getparam(pid_t, struct sched_param *);

34502 int sched_getscheduler(pid_t);

34503 int sched_rr_get_interval(pid_t, struct timespec *);

34504 int sched_setparam(pid_t, const struct sched_param *);

34505 int sched_setscheduler(pid_t, int, const struct sched_param *);

34506 int sched_yield(void);

34507 Inclusion of the <sched.h> header will make visible symbols defined in the header <time.h>.

34508 **APPLICATION USAGE**

34509 None.

34510 **FUTURE DIRECTIONS**

34511 None.

34512 **SEE ALSO**

34513 <time.h>.

34514 **CHANGE HISTORY**

34515 First released in Issue 5.

34516 Included for alignment with the POSIX Realtime Extension.

34517 **NAME**

34518 search.h — search tables

34519 **SYNOPSIS**34520 EX `#include <search.h>`

34521

34522 **DESCRIPTION**

34523 The <search.h> header provides a type definition, **ENTRY**, for structure **entry** which includes
 34524 the following members:

34525 char *key
 34526 void *data

34527 and defines **ACTION** and **VISIT** as enumeration data types through type definitions as follows:

34528 enum { FIND, ENTER } ACTION;
 34529 enum { preorder, postorder, endorder, leaf } VISIT;

34530 The **size_t** type is defined as described in <sys/types.h>.

34531 Each of the following is declared as a function, or defined as a macro, or both. Function
 34532 prototypes must be provided for use with an ISO C compiler.

```
34533 int    hcreate(size_t);
34534 void    hdestroy(void);
34535 ENTRY *hsearch(ENTRY, ACTION);
34536 void    insque(void *, void *);
34537 void    *lfind(const void *, const void *, size_t *,
34538               size_t, int (*)(const void *, const void *));
34539 void    *lsearch(const void *, void *, size_t *,
34540                size_t, int (*)(const void *, const void *));
34541 void    remque(void *);
34542 void    *tdelete(const void *, void *,
34543                 int (*)(const void *, const void *));
34544 void    *tfind(const void *, void *const *,
34545               int (*)(const void *, const void *));
34546 void    *tsearch(const void *, void *,
34547                 int (*)(const void *, const void *));
34548 void    twalk(const void *,
34549              void (*)(const void *, VISIT, int ));
```

34550 **APPLICATION USAGE**

34551 None.

34552 **FUTURE DIRECTIONS**

34553 None.

34554 **SEE ALSO**34555 *hsearch()*, *insque()*, *lsearch()*, *remque()*, *tsearch()*, <sys/types.h>.34556 **CHANGE HISTORY**

34557 First released in Issue 1.

34558 Derived from Issue 1 of the SVID.

34559 **Issue 4**

34560 The following changes are incorporated in this issue:

- 34561 • The function declarations in this header are expanded to full ISO C prototypes.
- 34562 • Reference to the <sys/types.h> header is added for the definition of **size_t**.

34563 **Issue 4, Version 2**

34564 For X/OPEN UNIX conformance, the *insque()* and *remque()* functions are added to the list of
 34565 functions declared in this header.

34566 **NAME**34567 semaphore.h — semaphores (**REALTIME**)34568 **SYNOPSIS**34569 RT `#include <semaphore.h>`

34570

34571 **DESCRIPTION**34572 The <semaphore.h> header defines the **sem_t** type, used in performing semaphore operations.34573 The semaphore may be implemented using a file descriptor, in which case applications are able
34574 to open up at least a total of OPEN_MAX files and semaphores.34575 The following are declared as functions and may also be declared as macros. Function
34576 prototypes must be provided for use with an ISO C compiler.

```

34577     int      sem_close(sem_t *);
34578     int      sem_destroy(sem_t *);
34579     int      sem_getvalue(sem_t *, int *);
34580     int      sem_init(sem_t *, int, unsigned int);
34581     sem_t *sem_open(const char *, int, ...);
34582     int      sem_post(sem_t *);
34583     int      sem_trywait(sem_t *);
34584     int      sem_unlink(const char *);
34585     int      sem_wait(sem_t *);

```

34586 Inclusion of the <semaphore.h> header may make visible symbols defined in the headers
34587 <fcntl.h> and <sys/types.h>.34588 **APPLICATION USAGE**

34589 None.

34590 **FUTURE DIRECTIONS**

34591 None.

34592 **SEE ALSO**

34593 <fcntl.h>, <sys/types.h>.

34594 **CHANGE HISTORY**

34595 First released in Issue 5.

34596 Included for alignment with the POSIX Realtime Extension.

34597 **NAME**

34598 setjmp.h — stack environment declarations

34599 **SYNOPSIS**

34600 #include <setjmp.h>

34601 **DESCRIPTION**

34602 The <setjmp.h> header contains the type definitions for array types **jmp_buf** and **sigjmp_buf**.

34603 The following are declared as functions and may also be defined as macros. Function prototypes
34604 must be provided for use with an ISO C compiler.

34605 void longjmp(jmp_buf, int);
34606 void siglongjmp(sigjmp_buf, int);
34607 EX void _longjmp(jmp_buf, int);
34608

34609 Each of the following may be declared as a function, or defined as a macro, or both. Function
34610 prototypes must be provided for use with an ISO C compiler.

34611 int setjmp(jmp_buf);
34612 int sigsetjmp(sigjmp_buf, int);
34613 EX int _setjmp(jmp_buf);
34614

34615 **APPLICATION USAGE**

34616 None.

34617 **FUTURE DIRECTIONS**

34618 None.

34619 **SEE ALSO**

34620 *longjmp()*, *_longjmp()*, *setjmp()*, *siglongjmp()*, *sigsetjmp()*.

34621 **CHANGE HISTORY**

34622 First released in Issue 1.

34623 **Issue 4**

34624 The following changes are incorporated for alignment with the ISO C standard:

- 34625 • The function declarations in this header are expanded to full ISO C prototypes.
- 34626 • The DESCRIPTION is changed to indicate that all functions in this header can also be
34627 declared as macros.
- 34628 • The arguments *jmp_buf* and *sigjmp_buf* are specified as array types.

34629 **Issue 4, Version 2**

34630 For X/OPEN UNIX conformance, the *_longjmp()* and *_setjmp()* functions are added to the list of
34631 functions declared in this header.

34632 NAME

34633 signal.h — signals

34634 SYNOPSIS

34635 #include <signal.h>

34636 DESCRIPTION

34637 The <signal.h> header defines the following symbolic constants, each of which expands to a
 34638 distinct constant expression of the type:

34639 void (*)(int)

34640 whose value matches no declarable function.

34641 SIG_DFL Request for default signal handling.

34642 SIG_ERR Return value from *signal()* in case of error.

34643 SIG_HOLD Request that signal be held.

34644 SIG_IGN Request that signal be ignored.

34645 The following data types are defined through **typedef**:

34646 **sig_atomic_t** Integral type of an object that can be accessed as an atomic entity, even in the
 34647 presence of asynchronous interrupts

34648 **sigset_t** Integral or structure type of an object used to represent sets of signals.34649 EX **pid_t** As described in <sys/types.h>.34650 RT The <signal.h> header defines the **sigevent** structure, which has at least the following members:

34651	int	sigev_notify	notification type
34652	int	sigev_signo	signal number
34653	union sigval	sigev_value	signal value
34654	void(*)(unsigned sigval)	sigev_notify_function	notification function
34655	(pthread_attr_t*)	sigev_notify_attributes	notification attributes

34656 The following values of *sigev_notify* are defined:

34657 SIGEV_NONE No asynchronous notification will be delivered when the event of interest
 34658 occurs.

34659 SIGEV_SIGNAL A queued signal, with an application-defined value, will be generated
 34660 when the event of interest occurs.

34661 SIGEV_THREAD A notification function will be called to perform notification.

34662 The **sigval** union is defined as:

34663	int	sival_int	integer signal value
34664	void*	sival_ptr	pointer signal value

34665 This header also declares the macros SIGRTMIN and SIGRTMAX, which evaluate to integral
 34666 expressions and, if the Realtime Signals Extension option is supported, specify a range of signal
 34667 numbers that are reserved for application use and for which the realtime signal behaviour
 34668 specified in this specification is supported. The signal numbers in this range do not overlap any
 34669 of the signals specified in the following table.

34670 The range SIGRTMIN through SIGRTMAX inclusive includes at least RTSIG_MAX signal
 34671 numbers.

34672 It is implementation-dependent whether realtime signal behaviour is supported for other
 34673 signals.

34674 This header also declares the constants that are used to refer to the signals that occur in the
 34675 system. Signals defined here begin with the letters SIG. Each of the signals have distinct

positive integral values. The value 0 is reserved for use as the null signal (see *kill()*). Additional implementation-dependent signals may occur in the system.

The following signals are supported on all implementations (default actions are explained below the table):

Signal	Default Action	Description
SIGABRT	ii	Process abort signal.
SIGALRM	i	Alarm clock.
SIGFPE	ii	Erroneous arithmetic operation.
SIGHUP	i	Hangup.
SIGILL	ii	Illegal instruction.
SIGINT	i	Terminal interrupt signal.
SIGKILL	i	Kill (cannot be caught or ignored).
SIGPIPE	i	Write on a pipe with no one to read it.
SIGQUIT	ii	Terminal quit signal.
SIGSEGV	ii	Invalid memory reference.
SIGTERM	i	Termination signal.
SIGUSR1	i	User-defined signal 1.
SIGUSR2	i	User-defined signal 2.
SIGCHLD	iii	Child process terminated or stopped.
SIGCONT	v	Continue executing, if stopped.
SIGSTOP	iv	Stop executing (cannot be caught or ignored).
SIGTSTP	iv	Terminal stop signal.
SIGTTIN	iv	Background process attempting read.
SIGTTOU	iv	Background process attempting write.
SIGBUS	ii	Access to an undefined portion of a memory object.
SIGPOLL	i	Pollable event.
SIGPROF	i	Profiling timer expired.
SIGSYS	ii	Bad system call.
SIGTRAP	ii	Trace/breakpoint trap.
SIGURG	iii	High bandwidth data is available at a socket.
SIGVTALRM	i	Virtual timer expired.
SIGXCPU	ii	CPU time limit exceeded.
SIGXFSZ	ii	File size limit exceeded.

The default actions are as follows:

- i Abnormal termination of the process. The process is terminated with all the consequences of *_exit()* except that the status is made available to *wait()* and *waitpid()* indicates abnormal termination by the specified signal.
 - ii Abnormal termination of the process.
 - iii Ignore the signal.
 - iv Stop the process.
 - v Continue the process, if it is stopped; otherwise ignore the signal.
- Additionally, implementation-dependent abnormal termination actions, such as creation of a core file, may occur.

34720 The header provides a declaration of **struct sigaction**, including at least the following members:

34721	void	(*sa_handler)(int)	what to do on receipt of signal
34722	sigset_t	sa_mask	set of signals to be blocked during execution of the signal handling function
34723			
34724	int	sa_flags	special flags
34725	void (*)	(int, siginfo_t *, void *)	sa_sigaction
34726			pointer to signal handler function or one of the macros SIG_IGN or SIG_DFL
34727			

34728 EX The storage occupied by **sa_handler** and **sa_sigaction** may overlap, and a portable program must not use both simultaneously.

34730 The following are declared as constants:

34731	SA_NOCLDSTOP	Do not generate SIGCHLD when children stop.
34732	SIG_BLOCK	The resulting set is the union of the current set and the signal set pointed to by the argument <i>set</i> .
34733		
34734	SIG_UNBLOCK	The resulting set is the intersection of the current set and the complement of the signal set pointed to by the argument <i>set</i> .
34735		
34736	SIG_SETMASK	The resulting set is the signal set pointed to by the argument <i>set</i> .
34737 EX	SA_ONSTACK	Causes signal delivery to occur on an alternate stack.
34738	SA_RESETHAND	Causes signal dispositions to be set to SIG_DFL on entry to signal handlers.
34739		
34740	SA_RESTART	Causes certain functions to become restartable.
34741	SA_SIGINFO	Causes extra information to be passed to signal handlers at the time of receipt of a signal.
34742		
34743	SA_NOCLDWAIT	Causes implementations not to create zombie processes on child death.
34744	SA_NODEFER	Causes signal not to be automatically blocked on entry to signal handler.
34745	SS_ONSTACK	Process is executing on an alternate signal stack.
34746	SS_DISABLE	Alternate signal stack is disabled.
34747	MINSIGSTKSZ	Minimum stack size for a signal handler.
34748	SIGSTKSZ	Default size in bytes for the alternate signal stack.

34749 The **ucontext_t** structure is defined through typedef as described in <ucontext.h>.

34750 The <signal.h> header defines the **stack_t** type as a structure that includes at least the following members:

34752	void	*ss_sp	stack base or pointer
34753	size_t	ss_size	stack size
34754	int	ss_flags	flags

34755 The <signal.h> header defines the **sigstack** structure that includes at least the following members:

34757	int	ss_onstack	non-zero when signal stack is in use
34758	void	*ss_sp	signal stack pointer

34759 The <signal.h> header defines the **siginfo_t** type as a structure that includes at least the following members:

34761	int	si_signo	signal number
34762	int	si_errno	if non-zero, an <i>errno</i> value associated with this signal, as defined in <errno.h>
34763			
34764	int	si_code	signal code
34765	pid_t	si_pid	sending process ID
34766	uid_t	si_uid	real user ID of sending process

34767	void	*si_addr	address of faulting instruction	
34768	int	si_status	exit value or signal	
34769	long	si_band	band event for SIGPOLL	
34770 RT	union sigval	si_value	signal value	
34771				
34772 EX	The macros specified in the Code column of the following table are defined for use as values of			
34773	si_code that are signal-specific reasons why the signal was generated.			

34774

34775

34776

34777

34778

34779

34780

34781

34782

34783

34784

34785

34786

34787

34788

34789

34790

34791

34792

34793

34794

34795

34796

34797

34798

34799

34800

34801

34802

34803

34804

34805

34806

34807

34808

34809

34810

34811

34812

34813

34814

34815

34816

34817

Signal	Code	Reason
SIGILL	ILL_ILLOPC	illegal opcode
	ILL_ILLOPN	illegal operand
	ILL_ILLADR	illegal addressing mode
	ILL_ILTRP	illegal trap
	ILL_PRVOPC	privileged opcode
	ILL_PRVREG	privileged register
	ILL_COPROC	coprocessor error
	ILL_BADSTK	internal stack error
SIGFPE	FPE_INTDIV	integer divide by zero
	FPE_INTOVF	integer overflow
	FPE_FLTDIV	floating point divide by zero
	FPE_FLTOVF	floating point overflow
	FPE_FLTUND	floating point underflow
	FPE_FLTRES	floating point inexact result
	FPE_FLTINV	invalid floating point operation
	FPE_FLTSUB	subscript out of range
SIGSEGV	SEGV_MAPERR	address not mapped to object
	SEGV_ACCERR	invalid permissions for mapped object
SIGBUS	BUS_ADRALN	invalid address alignment
	BUS_ADRERR	non-existent physical address
	BUS_OBJERR	object specific hardware error
SIGTRAP	TRAP_BRKPT	process breakpoint
	TRAP_TRACE	process trace trap
SIGCHLD	CLD_EXITED	child has exited
	CLD_KILLED	child has terminated abnormally and did not create a core file
	CLD_DUMPED	child has terminated abnormally and created a core file
	CLD_TRAPPED	traced child has trapped
	CLD_STOPPED	child has stopped
	CLD_CONTINUED	stopped child has continued
SIGPOLL	POLL_IN	data input available
	POLL_OUT	output buffers available
	POLL_MSG	input message available
	POLL_ERR	I/O error
	POLL_PRI	high priority input available
	POLL_HUP	device disconnected
	SI_USER	signal sent by <i>kill()</i>
	SI_QUEUE	signal sent by the <i>sigqueue()</i>
	SI_TIMER	signal generated by expiration of a timer set by <i>timer_settime()</i>
	SI_ASYNCIO	signal generated by completion of an asynchronous I/O request
	SI_MESGQ	signal generated by arrival of a message on an empty message queue

Implementations may support additional **si_code** values not included in this list, may generate values included in this list under circumstances other than those described in this list, and may contain extensions or limitations that prevent some values from being generated. Implementations will not generate a different value from the ones described in this list for circumstances described in this list.

Signal	Member	Value
SIGILL SIGFPE	void * si_addr	address of faulting instruction
SIGSEGV SIGBUS	void * si_addr	address of faulting memory reference
SIGCHLD	pid_t si_pid int si_status uid_t si_uid	child process ID exit value or signal real user ID of the process that sent the signal
SIGPOLL	long si_band	band event for POLL_IN, POLL_OUT or POLL_MSG

In addition, the following signal-specific information will be available:

For some implementations, the value of *si_addr* may be inaccurate.

The following are declared as functions and may also be defined as macros.

```

void (*bsd_signal(int, void (*)(int)))(int);
int kill(pid_t, int);
int killpg(pid_t, int);
int pthread_kill(pthread_t, int);
int pthread_sigmask(int, const sigset_t *, sigset_t *);
int raise(int);
int sigaction(int, const struct sigaction *, struct sigaction *);
int sigaddset(sigset_t *, int);
int sigaltstack(const stack_t *, stack_t *);
int sigdelset(sigset_t *, int);
int sigemptyset(sigset_t *);
int sigfillset(sigset_t *);
int sighold(int);
int sigignore(int);
int siginterrupt(int, int);
int sigismember(const sigset_t *, int);
void (*signal(int, void (*)(int)))(int);
int sigpause(int);
int sigpending(sigset_t *);
int sigprocmask(int, const sigset_t *, sigset_t *);
int sigqueue(pid_t, int, const union sigval);
int sigrelse(int);
void *sigset(int, void (*)(int))(int);
int sigstack(struct sigstack *ss,
             struct sigstack *oss); (LEGACY)
int sigsuspend(const sigset_t *);
int sigtimedwait(const sigset_t *, siginfo_t *,
                const struct timespec *);
int sigwait(const sigset_t *set, int *sig);
int sigwaitinfo(const sigset_t *, siginfo_t *);

```

34867 **APPLICATION USAGE**

34868 None.

34869 **FUTURE DIRECTIONS**

34870 None.

34871 **SEE ALSO**

34872 *alarm()*, *bsd_signal()*, *ioctl()*, *kill()*, *killpg()*, *raise()*, *sigaction()*, *sigaddset()*, *sigaltstack()*,
 34873 *sigdelset()*, *sigemptyset()*, *sigfillset()*, *siginterrupt()*, *sigismember()*, *signal()*, *sigpending()*,
 34874 *sigprocmask()*, *sigqueue()*, *sigsuspend()*, *sigwaitinfo()*, *wait()*, *waitid()*, <errno.h>, <streams.h>,
 34875 <sys/types.h>, <ucontext.h>.

34876 **CHANGE HISTORY**

34877 First released in Issue 1.

34878 **Issue 4**

34879 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- 34880 • The function declarations in this header are expanded to full ISO C prototypes.
- 34881 • The DESCRIPTION is changed:
 - 34882 — to define the type **sig_atomic_t**
 - 34883 — to define the syntax of signal names and functions
 - 34884 — to combine the two tables of constants
 - 34885 — SIGFPE is no longer limited to floating-point exceptions, but covers all erroneous
 - 34886 arithmetic operations.

34887 The following change is incorporated for alignment with the ISO C standard:

- 34888 • The *raise()* function is added to the list of functions declared in this header.

34889 Other changes are incorporated as follows:

- 34890 • A reference to <sys/types.h> is added for the definition of **pid_t**. This is marked as an
- 34891 extension.
- 34892 • In the list of signals starting with SIGCHLD, the statement “but a system not supporting the
- 34893 job control option is not obliged to support the functionality of these signals” is removed.
- 34894 This is because job control is defined as mandatory on Issue 4 conforming implementations.
- 34895 • Reference to implementation-dependent abnormal termination routines, such as creation of a
- 34896 core file, in item ii in the defaults action list is marked as an extension.

34897 **Issue 4, Version 2**

34898 The following changes are incorporated for X/OPEN UNIX conformance:

- 34899 • The SIGTRAP, SIGBUS, SIGSYS, SIGPOLL, SIGPROF, SIGXCPU, SIGXFSZ, SIGURG and
- 34900 SIGVTALRM signals are added to the list of signals that will be supported on all conforming
- 34901 implementations.
- 34902 • The *sa_sigaction* member is added to the **sigaction** structure, and a note is added that the
- 34903 storage used by *sa_handler* and *sa_sigaction* may overlap.
- 34904 • The SA_ONSTACK, SA_RESETHAND, SA_RESTART, SA_SIGINFO, SA_NOCLDWAIT,
- 34905 SS_ONSTACK, SS_DISABLE, MINSIGSTKSZ and SIGSTKSZ constants are defined. The
- 34906 **stack_t**, **sigstack** and **siginfo** structures are defined.
- 34907 • Definitions are given for the **ucontext_t**, **stack_t**, **sigstack** and **siginfo_t** types.

34908	<ul style="list-style-type: none"> • A table is provided listing macros that are defined as signal-specific reasons why a signal was generated. Signal-specific additional information is specified. 	
34909		
34910	<ul style="list-style-type: none"> • The <i>bsd_signal()</i>, <i>killpg()</i>, <i>_longjmp()</i>, <i>_setjmp()</i>, <i>sigaltstack()</i>, <i>sighold()</i>, <i>sigignore()</i>, <i>siginterrupt()</i>, <i>sigpause()</i>, <i>sigrelse()</i>, <i>sigset()</i> and <i>sigstack()</i> functions are added to the list of functions declared in this header. 	
34911		
34912		
34913	Issue 5	
34914	The DESCRIPTION is updated for alignment with POSIX Realtime Extension and the POSIX Threads Extension.	
34915		
34916	The default action for SIGURG is changed for i to iii. The function prototype for <i>sigmask()</i> is removed.	
34917		

34918 NAME

34919 `stdarg.h` — handle variable argument list

34920 SYNOPSIS

34921 `#include <stdarg.h>`34922 `void va_start(va_list ap, argN);`34923 `type va_arg(va_list ap, type);`34924 `void va_end(va_list ap);`

34925 DESCRIPTION

34926 The `<stdarg.h>` header contains a set of macros which allows portable functions that accept
 34927 variable argument lists to be written. Functions that have variable argument lists (such as
 34928 `printf()`) but do not use these macros are inherently non-portable, as different systems use
 34929 different argument-passing conventions.

34930 The type `va_list` is defined for variables used to traverse the list.

34931 The `va_start()` macro is invoked to initialise `ap` to the beginning of the list before any calls to
 34932 `va_arg()`.

34933 The object `ap` may be passed as an argument to another function; if that function invokes the
 34934 `va_arg()` macro with parameter `ap`, the value of `ap` in the calling function is indeterminate and
 34935 must be passed to the `va_end()` macro prior to any further reference to `ap`. The parameter `argN` is
 34936 the identifier of the rightmost parameter in the variable parameter list in the function definition
 34937 (the one just before the `, ...`). If the parameter `argN` is declared with the **register** storage class,
 34938 with a function type or array type, or with a type that is not compatible with the type that results
 34939 after application of the default argument promotions, the behaviour is undefined.

34940 The `va_arg()` macro will return the next argument in the list pointed to by `ap`. Each invocation of
 34941 `va_arg()` modifies `ap` so that the values of successive arguments are returned in turn. The `type`
 34942 parameter is the type the argument is expected to be. This is the type name specified such that
 34943 the type of a pointer to an object that has the specified type can be obtained simply by suffixing
 34944 a `*` to `type`. Different types can be mixed, but it is up to the routine to know what type of
 34945 argument is expected.

34946 The `va_end()` macro is used to clean up; it invalidates `ap` for use (unless `va_start()` is invoked
 34947 again).

34948 Multiple traversals, each bracketed by `va_start()` ... `va_end()`, are possible.

34949 EXAMPLES

34950 This example is a possible implementation of `execl()`.

34951 `#include <stdarg.h>`34952 `#define MAXARGS 31`

```

34953        /*
34954        * execl is called by
34955        * execl(file, arg1, arg2, ..., (char *)(0));
34956        */
34957        int execl (const char *file, const char *args, ...)
34958        {
34959            va_list ap;
34960            char *array[MAXARGS];
34961            int argno = 0;
34962            va_start(ap, args);

```

```

34963         while (args != 0) {
34964             array[argno++] = args;
34965             args = va_arg(ap, const char *);
34966         }
34967         va_end(ap);
34968         return execv(file, array);
34969     }

```

34970 APPLICATION USAGE

34971 It is up to the calling routine to communicate to the called routine how many arguments there
 34972 are, since it is not always possible for the called routine to determine this in any other way. For
 34973 example, *exec()* is passed a null pointer to signal the end of the list. The *printf()* function can tell
 34974 how many arguments are there by the *format* argument.

34975 FUTURE DIRECTIONS

34976 None.

34977 SEE ALSO

34978 *exec*, *printf()*.

34979 CHANGE HISTORY

34980 First released in Issue 4.

34981 Derived from the ANSI C standard.

34982 **NAME**34983 **stddef.h** — standard type definitions34984 **SYNOPSIS**

34985 #include <stddef.h>

34986 **DESCRIPTION**34987 The <**stddef.h**> header defines the following:34988 **NULL** Null pointer constant.34989 offsetof(*type*, *member-designator*)34990 Integral constant expression of type **size_t**, the value of which is the offset in
34991 bytes to the structure member (*member-designator*), from the beginning of its
34992 structure (*type*).34993 The <**stddef.h**> header defines through **typedef**:34994 **ptrdiff_t** Signed integral type of the result of subtracting two pointers.34995 **wchar_t** Integral type whose range of values can represent distinct wide-character
34996 codes for all members of the largest character set specified among the locales
34997 supported by the compilation environment: the null character has the code
34998 value 0 and each member of the Portable Character Set has a code value equal
34999 to its value when used as the lone character in an integer character constant.35000 **size_t** Unsigned integral type of the result of the *sizeof* operator.35001 **APPLICATION USAGE**

35002 None.

35003 **FUTURE DIRECTIONS**

35004 None.

35005 **SEE ALSO**35006 <**wchar.h**>, <**sys/types.h**>.35007 **CHANGE HISTORY**

35008 First released in Issue 4.

35009 Derived from the ANSI C standard.

35010 **NAME**

35011 stdio.h — standard buffered input/output

35012 **SYNOPSIS**

35013 #include <stdio.h>

35014 **DESCRIPTION**35015 The <stdio.h> header defines the following macro names as positive integral constant
35016 expressions:

35017	BUFSIZ	Size of <stdio.h> buffers.
35018	FILENAME_MAX	Maximum size in bytes of the longest filename string that the 35019 implementation guarantees can be opened.
35020	FOPEN_MAX	Number of streams which the implementation guarantees can be open 35021 simultaneously. The value will be at least eight.
35022	_IOFBF	Input/output fully buffered.
35023	_IOLBF	Input/output line buffered.
35024	_IONBF	Input/output unbuffered.
35025	L_ctermid	Maximum size of character array to hold <i>ctermid()</i> output.
35026	L_tmpnam	Maximum size of character array to hold <i>tmpnam()</i> output.
35027	SEEK_CUR	Seek relative to current position.
35028	SEEK_END	Seek relative to end-of-file.
35029	SEEK_SET	Seek relative to start-of-file.
35030	TMP_MAX	Minimum number of unique filenames generated by <i>tmpnam()</i> .
35031 EX		Maximum number of times an application can call <i>tmpnam()</i> reliably. The 35032 value of TMP_MAX will be at least 10,000.

35033 The following macro name is defined as a negative integral constant expression:

35034 **EOF** End-of-file return value.

35035 The following macro name is defined as a null pointer constant:

35036 **NULL** Null pointer.

35037 The following macro name is defined as a string constant:

35038 EX **P_tmpdir** default directory prefix for *tmpnam()*.

35039 The following macro names are defined as expressions of type pointer to FILE:

35040 **stderr** Standard error output stream.35041 **stdin** Standard input stream.35042 **stdout** Standard output stream.35043 The following data types are defined through **typedef**:35044 **FILE** A structure containing information about a file.35045 **fpos_t** Type containing all information needed to specify uniquely every
35046 position within a file.35047 EX **va_list** As described in <stdarg.h>.35048 **size_t** As described in <stddef.h>.

35049 The following are declared as functions and may also be defined as macros. Function prototypes
 35050 must be provided for use with an ISO C compiler.

```

35051 void      clearerr(FILE *);
35052 char      *ctermid(char *);
35053 int        fclose(FILE *);
35054 FILE      *fdopen(int, const char *);
35055 int        feof(FILE *);
35056 int        ferror(FILE *);
35057 int        fflush(FILE *);
35058 int        fgetc(FILE *);
35059 int        fgetpos(FILE *, fpos_t *);
35060 char      *fgets(char *, int, FILE *);
35061 int        fileno(FILE *);
35062 void      flockfile(FILE *);
35063 FILE      *fopen(const char *, const char *);
35064 int        fprintf(FILE *, const char *, ...);
35065 int        fputc(int, FILE *);
35066 int        fputs(const char *, FILE *);
35067 size_t     fread(void *, size_t, size_t, FILE *);
35068 FILE      *freopen(const char *, const char *, FILE *);
35069 int        fscanf(FILE *, const char *, ...);
35070 int        fseek(FILE *, long int, int);
35071 EX int      fseeko(FILE *, off_t, int);
35072 int        fsetpos(FILE *, const fpos_t *);
35073 long int   ftell(FILE *);
35074 EX off_t     ftello(FILE *);
35075 int        ftrylockfile(FILE *);
35076 void      funlockfile(FILE *);
35077 size_t     fwrite(const void *, size_t, size_t, FILE *);
35078 int        getc(FILE *);
35079 int        getchar(void);
35080 int        getc_unlocked(FILE *);
35081 int        getchar_unlocked(void);
35082 EX int      getopt(int, char * const[], const char); (LEGACY)
35083 char      *gets(char *);
35084 EX int      getw(FILE *);
35085 int        pclose(FILE *);
35086 void      perror(const char *);
35087 FILE      *popen(const char *, const char *);
35088 int        printf(const char *, ...);
35089 int        putc(int, FILE *);
35090 int        putchar(int);
35091 int        putc_unlocked(int, FILE *);
35092 int        putchar_unlocked(int);
35093 int        puts(const char *);
35094 EX int      putw(int, FILE *);
35095 int        remove(const char *);
35096 int        rename(const char *, const char *);
35097 void      rewind(FILE *);
35098 int        scanf(const char *, ...);
35099 void      setbuf(FILE *, char *);
35100 int        setvbuf(FILE *, char *, int, size_t);

```

```

35101 EX    int      snprintf(char *, size_t, const char *, ...);
35102      int      sprintf(char *, const char *, ...);
35103      int      sscanf(const char *, const char *, int ...);
35104 EX    char      *tempnam(const char *, const char *);
35105      FILE      *tmpfile(void);
35106      char      *tmpnam(char *);
35107      int      ungetc(int, FILE *);
35108      int      vfprintf(FILE *, const char *, va_list);
35109      int      vprintf(const char *, va_list);
35110 EX    int      vsnprintf(char *, size_t, const char *, va_list);
35111      int      vsprintf(char *, const char *, va_list);

```

35112 The following external variables are defined:

```

35113 EX    extern char  *optarg;      )
35114      extern int    opterr;      )
35115      extern int    optind;      ) (LEGACY)
35116      extern int    optopt;      )
35117

```

35118 EX Inclusion of the <stdio.h> header may also make visible all symbols from <stddef.h>.

35119 APPLICATION USAGE

35120 None.

35121 FUTURE DIRECTIONS

35122 None.

35123 SEE ALSO

35124 *clearerr(), ctermid(), fclose(), fdopen(), fgetc(), fgetpos(), ferror(), feof(), fflush(), fgets(), fileno(),*
35125 *fopen(), fputc(), fputs(), fread(), freopen(), fseek(), fsetpos(), ftell(), fwrite(), getc(), getc_unlocked(),*
35126 *getwchar(), getws(), getchar(), getopt(), gets(), pclose(), perror(), popen(), printf(), putc(),*
35127 *putchar(), puts(), putwchar(), remove(), rename(), rewind(), scanf(), setbuf(), setvbuf(), sscanf(),*
35128 *stdin, system(), tempnam(), tmpfile(), tmpnam(), ungetc(), vprintf(), <sys/types.h>.*

35129 CHANGE HISTORY

35130 First released in Issue 1.

35131 Derived from Issue 1 of the SVID.

35132 Issue 4

35133 The following changes are incorporated for alignment with the ISO C standard:

- 35134 • The function declarations in this header are expanded to full ISO C prototypes.
- 35135 • The DESCRIPTION is restructured to group lists of macro names according to how they will
35136 be defined by an implementation (for example, whether they are integral constant
35137 expressions, pointer constants or string constants).
- 35138 • The constant FILENAME_MAX is added to the list of integral constant expressions. The text
35139 of FOPEN_MAX has also been changed for consistency with the ISO C standard.
- 35140 • The data type **fpos_t** is moved from the APPLICATION USAGE section to the
35141 DESCRIPTION.
- 35142 • The functions *fgetpos()* and *fsetpos()* are added to the list of functions declared in this header.

35143	Other changes are incorporated as follows:	
35144	• The constant <code>L_cuserid</code> and the external variables <i>optarg</i> , <i>opterr</i> , <i>optind</i> and <i>optopt</i> are marked	
35145	as extensions and TO BE WITHDRAWN.	
35146	• The minimum allowable value of <code>TMP_MAX</code> , 10,000 on XSI-conformant systems, has been	
35147	marked as an extension.	
35148	• The <code>P_tmpdir</code> constant is moved from the APPLICATION USAGE section to the	
35149	DESCRIPTION and marked as an extension. The remainder of the APPLICATION USAGE	
35150	section is removed.	
35151	• References to the va_list and size_t types are added to the DESCRIPTION.	
35152	• Function declarations of the <i>cuserid()</i> , <i>getopt()</i> , <i>getw()</i> , <i>putw()</i> and <i>tempnam()</i> functions, and	
35153	the va_list type are marked as extensions.	
35154	• The <i>cuserid()</i> and <i>getopt()</i> functions are marked TO BE WITHDRAWN.	
35155	• A warning is added indicating that inclusion of <stdio.h> may also make visible all symbols	
35156	from <stddef.h>.	
35157	Issue 5	
35158	The DESCRIPTION is updated for alignment with the POSIX Threads Extension.	
35159	Large File System extensions added.	
35160	The constant <code>L_cuserid</code> and the external variables <i>optarg</i> , <i>opterr</i> , <i>optind</i> and <i>optopt</i> are marked as	
35161	extensions and LEGACY.	
35162	The <i>cuserid()</i> and <i>getopt()</i> functions are marked LEGACY.	

35163 NAME

35164 stdlib.h — standard library definitions

35165 SYNOPSIS

35166 #include <stdlib.h>

35167 DESCRIPTION

35168 The <stdlib.h> header defines the following macro names:

35169 EXIT_FAILURE Unsuccessful termination for *exit()*, evaluates to a non-zero value.
 35170 EXIT_SUCCESS Successful termination for *exit()*, evaluates to 0.
 35171 NULL Null pointer.
 35172 RAND_MAX Maximum value returned by *rand()*, at least 32,767.
 35173 MB_CUR_MAX Integer expression whose value is the maximum number of bytes in a
 35174 character specified by the current locale.

35175 The following data types are defined through **typedef**:

35176 **div_t** Structure type returned by *div()* function.
 35177 **ldiv_t** Structure type returned by *ldiv()* function.
 35178 **size_t** As described in <stddef.h>.
 35179 **wchar_t** As described in <stddef.h>.

35180 In addition, the following symbolic names and macros are defined as in <sys/wait.h>, for use in
 35181 decoding the return value from *system()*:

35182 EX WNOHANG
 35183 WUNTRACED
 35184 WEXITSTATUS()
 35185 WIFEXITED()
 35186 WIFSIGNALED()
 35187 WIFSTOPPED()
 35188 WSTOPSIG()
 35189 WTERMSIG()
 35190

35191 The following are declared as functions and may also be defined as macros. Function prototypes
 35192 must be provided for use with an ISO C compiler.

35193 EX long a64l(const char *);
 35194 void abort(void);
 35195 int abs(int);
 35196 int atexit(void (*)(void));
 35197 double atof(const char *);
 35198 int atoi(const char *);
 35199 long int atol(const char *);
 35200 void *bsearch(const void *, const void *, size_t, size_t,
 35201 int (*)(const void *, const void *));
 35202 void *calloc(size_t, size_t);
 35203 div_t div(int, int);
 35204 EX double drand48(void);
 35205 char *ecvt (double, int, int *, int *);
 35206 double erand48(unsigned short int[3]);
 35207 void exit(int);
 35208 EX char *fcvt (double, int, int *, int *);
 35209 void free(void *);

```

35210 EX char      *gcvt (double, int, char *);
35211 char      *getenv(const char *);
35212 EX int       getsuopt(char **, char *const *, char **);
35213 int       grantpt(int);
35214 char      *initstate(unsigned int, char *, size_t);
35215 long int   jrand48 (unsigned short int[3]);
35216 char      *l64a(long);
35217 long int   labs(long int);
35218 EX void     lcong48(unsigned short int[7]);
35219 ldiv_t     ldiv(long int, long int);
35220 EX long int  lrand48 (void);
35221 void       *malloc(size_t);
35222 int        mblen (const char *, size_t);
35223 size_t     mbstowcs (wchar_t *, const char *, size_t);
35224 int        mbtowc (wchar_t *, const char *, size_t);
35225 EX char     *mktemp(char *);
35226 int        mkstemp(char *);
35227 long int   mrand48 (void);
35228 long int   nrand48 (unsigned short int [3]);
35229 char       *ptsname(int);
35230 int        putenv(const char *);
35231 void       qsort(void *, size_t, size_t, int (*)(const void *,
35232 const void *));
35233 int        rand(void);
35234 int        rand_r(unsigned int *);
35235 EX long     random(void);
35236 void       realloc(void *, size_t);
35237 EX char     realpath(const char *, char *);
35238 unsigned   short int  seed48 (unsigned short int[3]);
35239 void       setkey(const char *);
35240 char       *setstate(const char *);
35241 void       srand(unsigned int);
35242 EX void     srand48(long int);
35243 void       srandom(unsigned);
35244 double     strtod(const char *, char **);
35245 long int   strtol(const char *, char **, int);
35246 unsigned long int
35247 strtoul(const char *, char **, int);
35248 int        system(const char *);
35249 EX int      ttyslot(void); (LEGACY)
35250 int        unlockpt(int);
35251 void       *valloc(size_t); (LEGACY)
35252 size_t     wcstombs(char *, const wchar_t *, size_t);
35253 int        wctomb(char *, wchar_t);

```

35254 EX Inclusion of the <stdlib.h> header may also make visible all symbols from <stddef.h>,
35255 <limits.h>, <math.h> and <sys/wait.h>.

35256 APPLICATION USAGE

35257 None.

35258 FUTURE DIRECTIONS

35259 None.

35260 SEE ALSO

35261 *a64l()*, *abort()*, *abs()*, *atexit()*, *atof()*, *atoi()*, *atol()*, *bsearch()*, *calloc()*, *div()*, *drand48()*, *ecvt()*,
 35262 *erand48()*, *exit()*, *fcvt()*, *free()*, *gcvt()*, *getenv()*, *getsubopt()*, *grantpt()*, *initstate()*, *jrand48()*, *l64a()*,
 35263 *labs()*, *lcong48()*, *ldiv()*, *lrand48()*, *malloc()*, *mblen()*, *mbstowcs()*, *mbtowc()*, *mktemp()*, *mkstemp()*,
 35264 *mrand48()*, *nrand48()*, *ptsname()*, *putenv()*, *qsort()*, *rand()*, *rand_r()*, *realloc()*, *realpath()*, *setstate()*,
 35265 *srand()*, *srand48()*, *srandom()*, *strtod()*, *strtol()*, *strtoul()*, *unlockpt()*, *wcstombs()*, *wctomb()*,
 35266 <sys/types.h>.

35267 CHANGE HISTORY

35268 First released in Issue 3.

35269 Issue 4

35270 The following changes are incorporated for alignment with the ISO C standard:

- 35271 • The function declarations in this header are expanded to full ISO C prototypes.
- 35272 • The maximum value of `RAND_MAX` is defined.
- 35273 • The name `MB_CUR_MAX` is added to the list of macro names defined in this header, while
 35274 `div_t` and `ldiv_t` are added to the list of defined types.
- 35275 • The names *atexit()*, *div()*, *labs()*, *ldiv()*, *mblen()*, *mbstowcs()*, *mbtowc()*, *strtoul()*, *wcstombs()*
 35276 and *wctomb()* are added to the list of functions declared in this header.

35277 Other changes are incorporated as follows:

- 35278 • A reference is added to <stddef.h> and <wchar.h> for the definition of `size_t`.
- 35279 • A reference is added to <sys/wait.h> for definitions of the symbolic names and macros
 35280 defined for decoding the return value from the *system()* function. This reference and the
 35281 symbolic names and macros are marked as an extension.
- 35282 • The names *drand48()*, *erand48()*, *jrand48()*, *lcong48()*, *lrand48()*, *mrand48()*, *nrand48()*,
 35283 *putenv()*, *seed48()*, *setkey()* and *srand48()* are added to the list of functions declared in this
 35284 header and marked as extensions.
- 35285 • A warning is added indicating that inclusion of <stdlib.h> may also make visible all symbols
 35286 from <stddef.h>, <limits.h>, <math.h> and <sys/wait.h>.
- 35287 • The APPLICATION USAGE section is removed.

35288 Issue 4, Version 2

35289 For X/OPEN UNIX conformance, the *a64l()*, *ecvt()*, *fcvt()*, *gcvt()*, *getsubopt()*, *grantpt()*,
 35290 *initstate()*, *l64a()*, *mktemp()*, *mkstemp()*, *ptsname()*, *random()*, *realpath()*, *setstate()*, *srandom()*,
 35291 *ttyslot()*, *unlockpt()* and *valloc()* functions are added to the list of functions declared in this
 35292 header.

35293 Issue 5

35294 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

35295 The *ttyslot()* and *valloc()* functions are marked LEGACY.

35296 The type of the third argument to *initstate()* is changed from `int` to `size_t`. The type of the return
 35297 value from *setstate()* is changed from `char` to `char*`, and the type of the first argument is changed
 35298 from `char*` to `const char*`.

35299 NAME

35300 string.h — string operations

35301 SYNOPSIS

35302 #include <string.h>

35303 DESCRIPTION

35304 The <string.h> header defines the following:

35305 NULL Null pointer constant.

35306 size_t As described in <stddef.h>.

35307 The following are declared as functions and may also be defined as macros. Function prototypes
 35308 must be provided for use with an ISO C compiler.

```

35309 EX void *memccpy(void *, const void *, int, size_t);
35310 void *memchr(const void *, int, size_t);
35311 int memcmp(const void *, const void *, size_t);
35312 void *memcpy(void *, const void *, size_t);
35313 void *memmove(void *, const void *, size_t);
35314 void *memset(void *, int, size_t);
35315 char *strcat(char *, const char *);
35316 char *strchr(const char *, int);
35317 int strcmp(const char *, const char *);
35318 int strcoll(const char *, const char *);
35319 char *strcpy(char *, const char *);
35320 size_t strcspn(const char *, const char *);
35321 EX char *strdup(const char *);
35322 char *strerror(int);
35323 size_t strlen(const char *);
35324 char *strncat(char *, const char *, size_t);
35325 int strncmp(const char *, const char *, size_t);
35326 char *strncpy(char *, const char *, size_t);
35327 char *strpbrk(const char *, const char *);
35328 char *strrchr(const char *, int);
35329 size_t strspn(const char *, const char *);
35330 char *strstr(const char *, const char *);
35331 char *strtok(char *, const char *);
35332 char *strtok_r(char *, const char *, char **);
35333 size_t strxfrm(char *, const char *, size_t);

```

35334 EX Inclusion of the <string.h> header may also make visible all symbols from <stddef.h>.

35335 APPLICATION USAGE

35336 None.

35337 FUTURE DIRECTIONS

35338 None.

35339 SEE ALSO

35340 *memccpy()*, *memchr()*, *memcmp()*, *memcpy()*, *memmove()*, *memset()*, *strcat()*, *strchr()*, *strcmp()*,
 35341 *strcoll()*, *strcpy()*, *strcspn()*, *strdup()*, *strerror()*, *strlen()*, *strncat()*, *strncmp()*, *strncpy()*, *strpbrk()*,
 35342 *strrchr()*, *strspn()*, *strstr()*, *strtok()*, *strxfrm()*, <sys/types.h>.

35343 CHANGE HISTORY

35344 First released in Issue 1.

35345 Derived from Issue 1 of the SVID.

35346 **Issue 4**

35347 The following changes are incorporated for alignment with the ISO C standard:

- 35348 • The function declarations in this header are expanded to full ISO C prototypes.
- 35349 • The name *memmove()* is added to the list of functions declared in this header.

35350 Other changes are incorporated as follows:

- 35351 • A reference is added to <stddef.h> for the definition of **size_t**.
- 35352 • The *memcpy()* function is marked as an extension.
- 35353 • A warning is added indicating that inclusion of <string.h> may also make visible all symbols
- 35354 from <stddef.h>.
- 35355 • The APPLICATION USAGE section is removed.

35356 **Issue 4, Version 2**

35357 For X/OPEN UNIX conformance, the *strdup()* function is added to the list of functions declared
35358 in this header.

35359 **Issue 5**

35360 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

35361 **NAME**

35362 strings — string operations

35363 **SYNOPSIS**35364 EX `#include <strings.h>`

35365

35366 **DESCRIPTION**

35367 The following are declared as functions and may also be defined as macros. Function prototypes
35368 must be provided for use with an ISO C compiler.

```
35369     int    bcmp(const void *, const void *, size_t);
35370     void    bcopy(const void *, void *, size_t);
35371     void    bzero(void *, size_t);
35372     int     ffs(int);
35373     char    *index(const char *);
35374     char    *rindex(const char *, int);
35375     int     strcasecmp(const char *, const char *);
35376     int     strncasecmp(const char *, const char *, size_t);
```

35377 The `size_t` type is defined through **typedef** as described in `<stddef.h>`.

35378 **APPLICATION USAGE**

35379 None.

35380 **FUTURE DIRECTIONS**

35381 None.

35382 **SEE ALSO**35383 *bcmp()*, *bcopy()*, *bzero()*, *ffs()*, *index()*, *rindex()*, *strcasecmp()*.35384 **CHANGE HISTORY**

35385 First released in Issue 4, Version 2.

35386 NAME

35387 stropts.h — STREAMS interface

35388 SYNOPSIS

35389 EX #include <stropts.h>

35390

35391 DESCRIPTION

35392 The <stropts.h> header defines the **bandinfo** structure that includes at least the following
35393 members:

35394 unsigned char bi_pri
35395 int bi_flag

35396 The <stropts.h> header defines the **strpeek** structure that includes at least the following
35397 members:

35398 struct strbuf ctlbuf
35399 struct strbuf databuf
35400 t_uscalar_t flags

35401 The <stropts.h> header defines the **strbuf** structure that includes at least the following members:

35402 int maxlen maximum buffer length
35403 int len length of data
35404 char *buf ptr to buffer

35405 The <stropts.h> header defines the **strfdinsert** structure that includes at least the following
35406 members:

35407 struct strbuf ctlbuf
35408 struct strbuf databuf
35409 t_uscalar_t flags
35410 int fildes
35411 int offset

35412 The <stropts.h> header defines the **striocli** structure that includes at least the following
35413 members:

35414 int ic_cmd
35415 int ic_timeout
35416 int ic_len
35417 char *ic_dp

35418 The <stropts.h> header defines the **strrecvfd** structure that includes at least the following
35419 members:

35420 int fd
35421 uid_t uid
35422 gid_t gid

35423 The **uid_t** and **gid_t** types are defined through **typedef** as described in <sys/types.h>.

35424 The **t_uscalar_t** type is defined as described in <xti.h> in the referenced **Networking Services,**
35425 **Issue 5** specification.

35426 The <stropts.h> header defines the **str_list** structure that includes at least the following
35427 members:

35428 int sl_nmods
35429 struct str_mlist *sl_modlist

35430 The <stropts.h> header defines the **str_mlist** structure that includes at least the following
 35431 member:

35432 char l_name[FMNAMESZ+1]

35433 At least the following macros are defined for use as the *request* argument to *ioctl()*:

35434 I_PUSH Push STREAMS module onto the top of the current STREAM, just below the
 35435 STREAM head.

35436 I_POP Remove STREAMS module from just below the STREAM head.

35437 I_LOOK Retrieve the name of the module just below the STREAM head and place it in
 35438 a character string. At least the following macros are defined for use as the *arg*
 35439 argument:

35440 FMNAMESZ The minimum size in bytes of the buffer referred to by the
 35441 *arg* argument.

35442 I_FLUSH This request flushes all input and/or output queues, depending on the value
 35443 of the *arg* argument. At least the following macros are defined for use as the
 35444 *arg* argument:

35445 FLUSHR Flush read queues.

35446 FLUSHW Flush write queues.

35447 FLUSHRW Flush read and write queues.

35448 I_FLUSHBAND Flush only band specified.

35449 I_SETSIG Informs the STREAM head that the process wants the SIGPOLL signal issued
 35450 (see *signal()* and *sigset()*) when a particular event has occurred on the
 35451 STREAM.

35452 The header <stropts.h> defines these possible values for *arg* when I_SETSIG is
 35453 specified:

35454 S_RDNORM A normal (priority band set to 0) message has arrived at the
 35455 head of a STREAM head read queue.

35456 S_RDBAND A message with a non-zero priority band has arrived at the
 35457 head of a STREAM head read queue.

35458 S_INPUT A message, other than a high-priority message, has arrived
 35459 at the head of a STREAM head read queue.

35460 S_HIPRI A high-priority message is present on a STREAM head read
 35461 queue.

35462 S_OUTPUT The write queue for normal data (priority band 0) just
 35463 below the STREAM head is no longer full. This notifies the
 35464 process that there is room on the queue for sending (or
 35465 writing) normal data downstream.

35466 S_WRNORM Same as S_OUTPUT.

35467 S_WRBAND The write queue for a non-zero priority band just below the
 35468 STREAM head is no longer full.

35469 S_MSG A STREAMS signal message that contains the SIGPOLL
 35470 signal reaches the front of the STREAM head read queue.

35471 S_ERROR Notification of an error condition reaches the STREAM
 35472 head.

35473 S_HANGUP Notification of a hangup reaches the STREAM head.

35474 S_BANDURG When used in conjunction with S_RDBAND, SIGURG is
 35475 generated instead of SIGPOLL when a priority message
 35476 reaches the front of the STREAM head read queue.

35477	I_GETSIG	Returns the events for which the calling process is currently registered to be sent a SIGPOLL signal.
35478		
35479	I_FIND	Compares the names of all modules currently present in the STREAM to the name pointed to by <i>arg</i> .
35480		
35481	I_PEEK	Allows a process to retrieve the information in the first message on the STREAM head read queue without taking the message off the queue. At least the following macros are defined for use as the <i>arg</i> argument:
35482		
35483		
35484		RS_HIPRI Only look for high-priority messages.
35485	I_SRDOPT	Sets the read mode. At least the following macros are defined for use as the <i>arg</i> argument:
35486		
35487		RNORM Byte-STREAM mode, the default.
35488		RMSGD Message-discard mode.
35489		RMSGN Message-nondiscard mode.
35490		RPROTNORM Fail <i>read()</i> with [EBADMSG] if a message containing a control part is at the front of the STREAM head read queue.
35491		
35492		RPROTDAT Deliver the control part of a message as data when a process issues a <i>read()</i> .
35493		
35494		RPROTDIS Discard the control part of a message, delivering any data part, when a process issues a <i>read()</i> .
35495		
35496	I_GRDOPT	Returns the current read mode setting.
35497	I_NREAD	Counts the number of data bytes in data blocks in the first message on the STREAM head read queue.
35498		
35499	I_FDINSERT	Creates a message from the specified buffer(s), adds information about another STREAM, and sends the message downstream.
35500		
35501	I_STR	Constructs an internal STREAMS <i>ioctl()</i> message and sends that message downstream.
35502		
35503	I_SWROPT	Sets the write mode. At least the following macros are defined for use as the <i>arg</i> argument:
35504		
35505		SNDZERO Send a zero-length message downstream when a <i>write()</i> of 0 bytes occurs.
35506		
35507	I_GWROPT	Returns the current write mode setting.
35508	I_SENDFD	Requests the STREAM associated with <i>fildev</i> to send a message, containing a file pointer, to the STREAM head at the other end of a STREAMS pipe.
35509		
35510	I_RECVFD	Retrieves the file descriptor associated with the message sent by an I_SENDFD <i>ioctl()</i> over a STREAMS pipe.
35511		
35512	I_LIST	This request allows the process to list all the module names on the STREAM, up to and including the topmost driver name.
35513		
35514	I_ATMARK	This request allows the process to see if the current message on the STREAM head read queue is "marked" by some module downstream. At least the following macros are defined for use as the <i>arg</i> argument:
35515		
35516		
35517		ANYMARK Check if the message is marked.
35518		LASTMARK Check if the message is the last one marked on the queue.
35519	I_CKBAND	Check if the message of a given priority band exists on the STREAM head read queue.
35520		
35521	I_GETBAND	Return the priority band of the first message on the STREAM head read queue.
35522		
35523	I_CANPUT	Check if a certain band is writable.
35524	I_SETCLTIME	Allows the process to set the time the STREAM head will delay when a STREAM is closing and there is data on the write queues.
35525		

35526 I_GETCLTIME Returns the close time delay.
 35527 I_LINK Connects two STREAMs.
 35528 I_UNLINK Disconnects the two STREAMs. The header defines at least the following
 35529 value for *arg*:

35530 MUXID_ALL Unlink all STREAMs linked to the STREAM associated with
 35531 *files*.

35532 I_PLINK Connects two STREAMs with a persistent link.
 35533 I_PUNLINK Disconnects the two STREAMs that were connected with a persistent link.

35534 The following macros are defined for *getmsg()*, *getpmsg()*, *putmsg()* and *putpmsg()*:

35535 MSG_ANY Receive any message.
 35536 MSG_BAND Receive message from specified band.
 35537 MSG_HIPRI Send/Receive high priority message.
 35538 MORECTL More control information is left in message.
 35539 MOREDATA More data is left in message.

35540 The header <stropts.h> may make visible all of the symbols from <unistd.h>.

35541 The following are declared as functions in the <stropts.h> header and may also be defined as
 35542 macros. Function prototypes must be provided for use with an ISO C compiler.

35543 int isastream(int);
 35544 int getmsg(int, struct strbuf *, struct strbuf *, int *);
 35545 int getpmsg(int, struct strbuf *, struct strbuf *, int *, int *);
 35546 int ioctl(int, int, ...);
 35547 int putmsg(int, const struct strbuf *, const struct strbuf *, int);
 35548 int putpmsg(int, const struct strbuf *, const struct strbuf *, int,
 35549 int);
 35550 int fattach(int, const char *);
 35551 int fdetach(const char *);

35552 **APPLICATION USAGE**
 35553 None.

35554 **FUTURE DIRECTIONS**
 35555 None.

35556 **SEE ALSO**
 35557 *close()*, *fcntl()*, *getmsg()*, *ioctl()*, *open()*, *pipe()*, *read()*, *poll()*, *putmsg()*, *signal()*, *sigset()*, *write()*,
 35558 <xti.h>.

35559 **CHANGE HISTORY**
 35560 First released in Issue 4, Version 2.

35561 **Issue 5**
 35562 The flags member of the **strpeek** and **strfdinsert** structures are changed from type **long** to
 35563 **t_uscalar_t**.

35564 **NAME**

35565 syslog — definitions for system error logging

35566 **SYNOPSIS**

35567 EX #include <syslog.h>

35568

35569 **DESCRIPTION**

35570 The <syslog.h> header defines the following symbolic constants, zero or more of which may be
35571 OR-ed together to form the *logopt* option of *openlog()*:

35572	LOG_PID	Log the process ID with each message.
35573	LOG_CONS	Log to the system console on error.
35574	LOG_NDELAY	Connect to syslog daemon immediately.
35575	LOG_ODELAY	Delay open until <i>syslog()</i> is called.
35576	LOG_NOWAIT	Don't wait for child processes.

35577 The following symbolic constants are defined as possible values of the *facility* argument to
35578 *openlog()*:

35579	LOG_KERN	Reserved for message generated by the system.
35580	LOG_USER	Message generated by a process.
35581	LOG_MAIL	Reserved for message generated by mail system.
35582	LOG_NEWS	Reserved for message generated by news system.
35583	LOG_UUCP	Reserved for message generated by UUCP system.
35584	LOG_DAEMON	Reserved for message generated by system daemon.
35585	LOG_AUTH	Reserved for message generated by authorisation daemon.
35586	LOG_CRON	Reserved for message generated by the clock daemon.
35587	LOG_LPR	Reserved for message generated by printer system.
35588	LOG_LOCAL0	Reserved for local use.
35589	LOG_LOCAL1	Reserved for local use.
35590	LOG_LOCAL2	Reserved for local use.
35591	LOG_LOCAL3	Reserved for local use.
35592	LOG_LOCAL4	Reserved for local use.
35593	LOG_LOCAL5	Reserved for local use.
35594	LOG_LOCAL6	Reserved for local use.
35595	LOG_LOCAL7	Reserved for local use.

35596 The following are declared as macros for constructing the *maskpri* argument to *setlogmask()*. The
35597 following macros expand to an expression of type **int** when the argument *pri* is an expression of
35598 type **int**:

35599	LOG_MASK(<i>pri</i>)	A mask for priority <i>pri</i> .
-------	------------------------	----------------------------------

35600 The following constants are defined as possible values for the *priority* argument of *syslog()*:

35601	LOG_EMERG	A panic condition was reported to all processes.
35602	LOG_ALERT	A condition that should be corrected immediately.
35603	LOG_CRIT	A critical condition.
35604	LOG_ERR	An error message.
35605	LOG_WARNING	A warning message.
35606	LOG_NOTICE	A condition requiring special handling.
35607	LOG_INFO	A general information message.
35608	LOG_DEBUG	A message useful for debugging programs.

35609 The following are declared as functions and may also be defined as macros. Function prototypes
35610 must be provided for use with an ISO C compiler.

35611 void closelog(void);
35612 void openlog(const char *, int, int);
35613 int setlogmask(int);
35614 void syslog(int, const char *, ...);

35615 **APPLICATION USAGE**

35616 None.

35617 **FUTURE DIRECTIONS**

35618 None.

35619 **SEE ALSO**

35620 *closelog()*.

35621 **CHANGE HISTORY**

35622 First released in Issue 4, Version 2.

35623 **Issue 5**

35624 Moved to X/Open UNIX to Base.

35625 NAME

35626 sys/ipc.h — interprocess communication access structure

35627 SYNOPSIS

35628 EX `#include <sys/ipc.h>`

35629

35630 DESCRIPTION

35631 The <sys/ipc.h> header is used by three mechanisms for interprocess communication (IPC):
 35632 messages, semaphores and shared memory. All use a common structure type, **ipc_perm** to pass
 35633 information used in determining permission to perform an IPC operation.

35634 The structure **ipc_perm** contains the following members:

35635	uid_t	uid	owner's user ID
35636	gid_t	gid	owner's group ID
35637	uid_t	cuid	creator's user ID
35638	gid_t	cgid	creator's group ID
35639	mode_t	mode	read/write permission

35640 The **uid_t**, **gid_t**, **mode_t** and **key_t** types are defined as described in <sys/types.h>.

35641 Definitions are given for the following constants:

35642 Mode bits:

35643	IPC_CREAT	Create entry if key does not exist.
35644	IPC_EXCL	Fail if key exists.
35645	IPC_NOWAIT	Error if request must wait.

35646 Keys:

35647	IPC_PRIVATE	Private key.
-------	-------------	--------------

35648 Control commands:

35649	IPC_RMID	Remove identifier.
35650	IPC_SET	Set options.
35651	IPC_STAT	Get options.

35652 The following is declared as a function and may also be defined as a macro. Function prototypes
 35653 must be provided for use with an ISO C compiler.

35654 `key_t ftok(const char *, int);`

35655 APPLICATION USAGE

35656 None.

35657 FUTURE DIRECTIONS

35658 None.

35659 SEE ALSO

35660 `ftok()`, <sys/types.h>.

35661 CHANGE HISTORY

35662 First released in Issue 2.

35663 Derived from System V Release 2.0.

35664 Issue 4

35665 The following changes are incorporated in this issue:

- 35666 • The DESCRIPTION is corrected to say that the header “is used by three mechanisms...”. |
- 35667 • Reference to the header **<sys/types.h>** is added for the definitions of **uid_t**, **gid_t** and
35668 **mode_t**.

35669 Issue 4, Version 2

35670 For X/OPEN UNIX conformance, the *ftok()* function is added to the list of functions declared in |
35671 this header.

35672 **NAME**

35673 sys/mman.h — memory management declarations

35674 **SYNOPSIS**35675 EX `#include <sys/mman.h>`

35676

35677 **DESCRIPTION**

35678 The following protection options are defined:

35679	PROT_READ	Page can be read.
35680	PROT_WRITE	Page can be written.
35681	PROT_EXEC	Page can be executed.
35682	PROT_NONE	Page can not be accessed.

35683 The following *flag* options are defined:

35684	MAP_SHARED	Share changes.
35685	MAP_PRIVATE	Changes are private.
35686	MAP_FIXED	Interpret addr exactly.

35687 The following flags are defined for *msync()*:

35688	MS_ASYNC	Perform asynchronous writes.
35689	MS_SYNC	Perform synchronous writes.
35690	MS_INVALIDATE	Invalidate mappings.

35691 RT The following symbolic constants are defined for the *mlockall()* function:

35692	MCL_CURRENT	Lock currently mapped pages.
35693	MCL_FUTURE	Lock pages that become mapped.

35694 The symbolic constant MAP_FAILED is defined to indicate a failure from the *mmap()* function.35695 The **size_t** and **off_t** types are defined as described in <sys/types.h>.

35696 The following are declared in <sys/mman.h> as functions and may also be defined as macros.
 35697 Function prototypes must be provided for use with an ISO C compiler.

35698 RT	int	mlock(const void *, size_t);
35699	int	mlockall(int);
35700	void	*mmap(void *, size_t, int, int, int, off_t);
35701	int	mprotect(void *, size_t, int);
35702	int	msync(void *, size_t, int);
35703 RT	int	munlock(const void *, size_t);
35704	int	munlockall(void);
35705	int	munmap(void *, size_t);
35706 RT	int	shm_open(const char *, int, mode_t);
35707	int	shm_unlink(const char *);

35708

35709 **APPLICATION USAGE**

35710 None.

35711 **FUTURE DIRECTIONS**

35712 None.

35713 **SEE ALSO**35714 *mlock()*, *mlockall()*, *mmap()*, *mprotect()*, *msync()*, *munmap()*, *shm_open()*, *shm_unlink()*.

35715 **CHANGE HISTORY**

35716 First released in Issue 4, Version 2.

35717 **Issue 5**

35718 Updated for alignment with the POSIX Realtime Extension.

35719 NAME

35720 sys/msg.h — message queue structures

35721 SYNOPSIS

35722 EX `#include <sys/msg.h>`

35723

35724 DESCRIPTION

35725 The <sys/msg.h> header defines the following constant and members of the structure **msqid_ds**.

35726 The following data types are defined through **typedef**:

35727 **msgqnum_t** Used for the number of messages in the message queue.

35728 **msglen_t** Used for the number of bytes allowed in a message queue.

35729 These types are unsigned integer types that are able to store values at least as large as a type
35730 **unsigned short**.

35731 Message operation flag:

35732 **MSG_NOERROR** No error if big message.

35733 The structure **msqid_ds** contains the following members:

35734	<code>struct ipc_perm</code>	<code>msg_perm</code>	operation permission structure
35735	<code>msgqnum_t</code>	<code>msg_qnum</code>	number of messages currently on queue
35736	<code>msglen_t</code>	<code>msg_qbytes</code>	maximum number of bytes allowed on queue
35737	<code>pid_t</code>	<code>msg_lspid</code>	process ID of last <i>msgsnd()</i>
35738	<code>pid_t</code>	<code>msg_lrpid</code>	process ID of last <i>msgrcv()</i>
35739	<code>time_t</code>	<code>msg_stime</code>	time of last <i>msgsnd()</i>
35740	<code>time_t</code>	<code>msg_rtime</code>	time of last <i>msgrcv()</i>
35741	<code>time_t</code>	<code>msg_ctime</code>	time of last change

35742 The **pid_t**, **time_t**, **key_t** and **size_t** types are defined as described in <sys/types.h>.

35743 The following are declared as functions and may also be defined as macros. Function prototypes
35744 must be provided for use with an ISO C compiler.

35745	<code>int</code>	<code>msgctl(int, int, struct msqid_ds *)</code> ;
35746	<code>int</code>	<code>msgget(key_t, int)</code> ;
35747	<code>ssize_t</code>	<code>msgrcv(int, void *, size_t, long int, int)</code> ;
35748	<code>int</code>	<code>msgsnd(int, const void *, size_t, int)</code> ;

35749 In addition, all of the symbols from <sys/ipc.h> will be defined when this header is included.

35750 APPLICATION USAGE

35751 None.

35752 FUTURE DIRECTIONS

35753 None.

35754 SEE ALSO

35755 *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*, <sys/types.h>.

35756 CHANGE HISTORY

35757 First released in Issue 2.

35758 Derived from System V Release 2.0.

35759 **Issue 4**

35760 The following changes are incorporated in this issue:

- 35761 • The function declarations in this header are expanded to full ISO C prototypes.
- 35762 • Reference to the header **<sys/types.h>** is added for the definitions of **pid_t**, **time_t**, **key_t** and
- 35763 **size_t**.
- 35764 • A statement is added indicating that all symbols in **<sys/ipc.h>** will be defined when this
- 35765 header is included.

35766 **NAME**

35767 sys/resource.h — definitions for XSI resource operations

35768 **SYNOPSIS**

35769 EX `#include <sys/resource.h>`

35770

35771 **DESCRIPTION**

35772 The <sys/resource.h> header defines the following symbolic constants as possible values of the
35773 *which* argument of *getpriority()* and *setpriority()*:

35774	PRIO_PROCESS	Identifies <i>who</i> argument as a process ID.
35775	PRIO_PGRP	Identifies <i>who</i> argument as a process group ID.
35776	PRIO_USER	Identifies <i>who</i> argument as a user ID.

35777 The following type is defined through **typedef**:

35778 **rlim_t** Unsigned integral type used for limit values.

35779 The following symbolic constants are defined:

35780	RLIM_INFINITY	A value of rlim_t indicating no limit.
35781	RLIM_SAVED_MAX	A value of type rlim_t indicating an unrepresentable saved hard limit.
35782		
35783	RLIM_SAVED_CUR	A value of type rlim_t indicating an unrepresentable saved soft limit.

35784 On implementations where all resource limits are representable in an object of type **rlim_t**,
35785 RLIM_SAVED_MAX and RLIM_SAVED_CUR need not be distinct from RLIM_INFINITY.

35786 The following symbolic constants are defined as possible values of the *who* parameter of
35787 *getrusage()*:

35788	RUSAGE_SELF	Returns information about the current process.
35789	RUSAGE_CHILDREN	Returns information about children of the current process.

35790 The <sys/resource.h> header defines the **rlimit** structure that includes at least the following
35791 members:

35792	<code>rlim_t rlim_cur</code>	the current (soft) limit
35793	<code>rlim_t rlim_max</code>	the hard limit

35794 The <sys/resource.h> header defines the **rusage** structure that includes at least the following
35795 members:

35796	<code>struct timeval ru_utime</code>	user time used
35797	<code>struct timeval ru_stime</code>	system time used

35798 The **timeval** structure is defined as described in <sys/time.h>.

35799 The following symbolic constants are defined as possible values for the *resource* argument of
35800 *getrlimit()* and *setrlimit()*:

35801	RLIMIT_CORE	Limit on size of core dump file.
35802	RLIMIT_CPU	Limit on CPU time per process.
35803	RLIMIT_DATA	Limit on data segment size.
35804	RLIMIT_FSIZE	Limit on file size.
35805	RLIMIT_NOFILE	Limit on number of open files.
35806	RLIMIT_STACK	Limit on stack size.
35807	RLIMIT_AS	Limit on address space size.

35808 The following are declared as functions and may also be defined as macros. Function prototypes
35809 must be provided for use with an ISO C compiler.

35810 int getpriority(int, id_t);
35811 int getrlimit(int, struct rlimit *);
35812 int getrusage(int, struct rusage *);
35813 int setpriority(int, id_t, int);
35814 int setrlimit(int, const struct rlimit *);

35815 The **id_t** type is defined through **typedef** as described in <sys/types.h>.

35816 Inclusion of the <sys/resource.h> header may also make visible all symbols from <sys/time.h>.

35817 **APPLICATION USAGE**

35818 None.

35819 **FUTURE DIRECTIONS**

35820 None.

35821 **SEE ALSO**

35822 *getpriority()*, *getrusage()*, *getrlimit()*.

35823 **CHANGE HISTORY**

35824 First released in Issue 4, Version 2.

35825 **Issue 5**

35826 Large File System extensions added.

35827 NAME

35828 sys/sem.h — semaphore facility

35829 SYNOPSIS

35830 EX `#include <sys/sem.h>`

35831

35832 DESCRIPTION

35833 The <sys/sem.h> header defines the following constants and structures.

35834 Semaphore operation flags:

35835 SEM_UNDO Set up adjust on exit entry.

35836 Command definitions for the function *semctl()*:

35837 GETNCNT Get **semncnt**.

35838 GETPID Get **sempid**.

35839 GETVAL Get **semval**.

35840 GETALL Get all cases of **semval**.

35841 GETZCNT Get **semzcnt**.

35842 SETVAL Set **semval**.

35843 SETALL Set all cases of **semval**.

35844 The structure **semid_ds** contains the following members:

35845 struct ipc_perm sem_perm operation permission structure

35846 unsigned short int sem_nsems number of semaphores in set

35847 time_t sem_otime last *semop*() time

35848 time_t sem_ctime last time changed by *semctl()*

35849 The **pid_t**, **time_t**, **key_t** and **size_t** types are defined as described in <sys/types.h>.

35850 A semaphore is represented by an anonymous structure containing the following members:

35851 unsigned short int semval semaphore value

35852 pid_t sempid process ID of last operation

35853 unsigned short int semncnt number of processes waiting for **semval**

35854 to become greater than current value

35855 unsigned short int semzcnt number of processes waiting for **semval**

35856 to become 0

35857 The structure **sembuf** contains the following members:

35858 unsigned short int sem_num semaphore number

35859 short int sem_op semaphore operation

35860 short int sem_flg operation flags

35861 The following are declared as functions and may also be defined as macros. Function prototypes
35862 must be provided for use with an ISO C compiler.

35863 int semctl(int, int, int, ...);

35864 int semget(key_t, int, int);

35865 int semop(int, struct sembuf *, size_t);

35866 In addition, all of the symbols from <sys/ipc.h> will be defined when this header is included.

35867 APPLICATION USAGE

35868 None.

35869 FUTURE DIRECTIONS

35870 None.

35871 SEE ALSO

35872 *semctl()*, *semget()*, *semop()*, **<sys/types.h>**.

35873 CHANGE HISTORY

35874 First released in Issue 2.

35875 Derived from System V Release 2.0.

35876 Issue 4

35877 The following changes are incorporated in this issue:

- 35878 • The function declarations in this header are expanded to full ISO C prototypes.
- 35879 • Reference to the header **<sys/types.h>** is added for the definitions of **pid_t**, **time_t**, **key_t** and
- 35880 **size_t**.
- 35881 • A statement is added indicating that all symbols in **<sys/ipc.h>** will be defined when this
- 35882 header is included.

35883 **NAME**

35884 sys/shm.h — shared memory facility

35885 **SYNOPSIS**

35886 EX #include <sys/shm.h>

35887

35888 **DESCRIPTION**

35889 The <sys/shm.h> header defines the following symbolic constants and structure:

35890 Symbolic constants:

35891 SHM_RDONLY Attach read-only (else read-write).

35892 SHMLBA Segment low boundary address multiple.

35893 SHM_RND Round attach address to SHMLBA.

35894 The following data types are defined through **typedef**:

35895 **shmatt_t** Unsigned integer used for the number of current attaches that must be able to
 35896 store values at least as large as a type **unsigned short**.

35897 The structure **shmid_ds** contains the following members:

35898	struct ipc_perm	shm_perm	operation permission structure
35899	size_t	shm_segsz	size of segment in bytes
35900	pid_t	shm_lpid	process ID of last shared memory operation
35901	pid_t	shm_cpid	process ID of creator
35902	shmatt_t	shm_nattch	number of current attaches
35903	time_t	shm_atime	time of last <i>shmat()</i>
35904	time_t	shm_dtime	time of last <i>shmdt()</i>
35905	time_t	shm_ctime	time of last change by <i>shmctl()</i>

35906 The **pid_t**, **time_t**, **key_t** and **size_t** types are defined as described in <sys/types.h>. The
 35907 following are declared as functions and may also be defined as macros. Function prototypes
 35908 must be provided for use with an ISO C compiler.

```
35909 void *shmat(int, const void *, int);
35910 int shmctl(int, int, struct shmid_ds *);
35911 int shmdt(const void *);
35912 int shmget(key_t, size_t, int);
```

35913 In addition, all of the symbols from <sys/ipc.h> will be defined when this header is included.

35914 **APPLICATION USAGE**

35915 None.

35916 **FUTURE DIRECTIONS**

35917 None.

35918 **SEE ALSO**35919 *shmat()*, *shmctl()*, *shmdt()*, *shmget()*, <sys/types.h>.35920 **CHANGE HISTORY**

35921 First released in Issue 2.

35922 Derived from System V Release 2.0.

35923 Issue 4

35924 The following changes are incorporated in this issue:

- 35925 • The function declarations in this header are expanded to full ISO C prototypes.
- 35926 • Reference to the header **<sys/types.h>** is added for the definitions of **pid_t**, **time_t**, **key_t** and
- 35927 **size_t**.
- 35928 • A statement is added indicating that all symbols in **<sys/ipc.h>** will be defined when this
- 35929 header is included.

35930 Issue 5

35931 The type of *shm_segsz* is changed from **int** to **size_t**.

35932 NAME

35933 sys/stat.h — data returned by the *stat()* function

35934 SYNOPSIS

35935 #include <sys/stat.h>

35936 DESCRIPTION

35937 EX The <sys/stat.h> header defines the structure of the data returned by the functions *fstat()*, *lstat()*,
35938 and *stat()*.

35939 The structure **stat** contains at least the following members:

35940	dev_t	st_dev	ID of device containing file
35941	ino_t	st_ino	file serial number
35942	mode_t	st_mode	mode of file (see below)
35943	nlink_t	st_nlink	number of links to the file
35944	uid_t	st_uid	user ID of file
35945	gid_t	st_gid	group ID of file
35946 EX	dev_t	st_rdev	device ID (if file is character or block special)
35947	off_t	st_size	file size in bytes (if file is a regular file)
35948	time_t	st_atime	time of last access
35949	time_t	st_mtime	time of last data modification
35950	time_t	st_ctime	time of last status change
35951 EX	blksize_t	st_blksize	a filesystem-specific preferred I/O block size for
35952			this object. In some filesystem types, this may
35953			vary from file to file
35954	blkcnt_t	st_blocks	number of blocks allocated for this object
35955			

35956 EX File serial number and device ID taken together uniquely identify the file within the system. The
35957 **dev_t**, **ino_t**, **mode_t**, **nlink_t**, **uid_t**, **gid_t**, **off_t** and **time_t** types are defined as described in
35958 <sys/types.h>. Times are given in seconds since the Epoch.

35959 The following symbolic names for the values of **st_mode** are also defined:

35960 File type:

35961 EX	S_IFMT	type of file
35962	S_IFBLK	block special
35963	S_IFCHR	character special
35964	S_IFIFO	FIFO special
35965	S_IFREG	regular
35966	S_IFDIR	directory
35967	S_IFLNK	symbolic link

35968 File mode bits:

35969	S_IRWXU	read, write, execute/search by owner
35970	S_IRUSR	read permission, owner
35971	S_IWUSR	write permission, owner
35972	S_IXUSR	execute/search permission, owner
35973	S_IRWXG	read, write, execute/search by group
35974	S_IRGRP	read permission, group
35975	S_IWGRP	write permission, group
35976	S_IXGRP	execute/search permission, group
35977	S_IRWXO	read, write, execute/search by others
35978	S_IROTH	read permission, others

35979	S_IWOTH	write permission, others
35980	S_IXOTH	execute/search permission, others
35981	S_ISUID	set-user-ID on execution
35982	S_ISGID	set-group-ID on execution
35983 EX	S_ISVTX	on directories, restricted deletion flag

35984 The bits defined by S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH,
35985 EX S_IWOTH, S_IXOTH, S_ISUID, S_ISGID and S_ISVTX are unique.

35986 S_IRWXU is the bitwise OR of S_IRUSR, S_IWUSR and S_IXUSR.

35987 S_IRWXG is the bitwise OR of S_IRGRP, S_IWGRP and S_IXGRP.

35988 S_IRWXO is the bitwise OR of S_IROTH, S_IWOTH and S_IXOTH.

35989 Implementations may OR other implementation-dependent bits into S_IRWXU, S_IRWXG and
35990 S_IRWXO, but they will not overlap any of the other bits defined in this document. The *file*
35991 *permission bits* are defined to be those corresponding to the bitwise inclusive OR of S_IRWXU,
35992 S_IRWXG and S_IRWXO.

35993 The following macros will test whether a file is of the specified type. The value *m* supplied to
35994 the macros is the value of **st_mode** from a **stat** structure. The macro evaluates to a non-zero
35995 value if the test is true, 0 if the test is false.

35996	S_ISBLK (<i>m</i>)	Test for a block special file.
35997	S_ISCHR (<i>m</i>)	Test for a character special file.
35998	S_ISDIR (<i>m</i>)	Test for a directory.
35999	S_ISFIFO (<i>m</i>)	Test for a pipe or FIFO special file.
36000	S_ISREG (<i>m</i>)	Test for a regular file.
36001 EX	S_ISLNK (<i>m</i>)	Test for a symbolic link.

36002 RT The implementation may implement message queues, semaphores, or shared memory objects as
36003 distinct file types. The following macros test whether a file is of the specified type. The value of
36004 the *buf* argument supplied to the macros is a pointer to a **stat** structure. The macro evaluates to a
36005 non-zero value if the specified object is implemented as a distinct file type and the specified file
36006 type is contained in the **stat** structure referenced by *buf*. Otherwise, the macro evaluates to zero.

36007	S_TYPEISMQ (<i>buf</i>)	Test for a message queue
36008	S_TYPEISSEM (<i>buf</i>)	Test for a semaphore
36009	S_TYPEISSHM (<i>buf</i>)	Test for a shared memory object
36010		

36011 The following are declared as functions and may also be defined as macros. Function prototypes
36012 must be provided for use with an ISO C compiler.

36013	int	chmod(const char *, mode_t);
36014 EX	int	fchmod(int, mode_t);
36015	int	fstat(int, struct stat *);
36016 EX	int	lstat(const char *, struct stat *);
36017	int	mkdir(const char *, mode_t);
36018	int	mkfifo(const char *, mode_t);
36019 EX	int	mknod(const char *, mode_t, dev_t);
36020	int	stat(const char *, struct stat *);
36021	mode_t	umask(mode_t);

36022 **APPLICATION USAGE**

36023 Use of the macros is recommended for determining the type of a file.

36024 **FUTURE DIRECTIONS**

36025 None.

36026 **SEE ALSO**36027 *chmod()*, *fchmod()*, *fstat()*, *lstat()*, *mkdir()*, *mkfifo()*, *mknod()*, *stat()*, *umask()*, <sys/types.h>.36028 **CHANGE HISTORY**

36029 First released in Issue 1.

36030 Derived from Issue 1 of the SVID.

36031 **Issue 4**

36032 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- 36033 • The function declarations in this header are expanded to full ISO C prototypes.
- 36034 • The DESCRIPTION is expanded to indicate (a) how files are uniquely identified within the
- 36035 system, (b) that times are given in units of seconds since the Epoch, (c) rules governing the
- 36036 definition and use of the file mode bits, and (d) usage of the file type test macros.

36037 Other changes are incorporated as follows:

- 36038 • Reference to the <sys/types.h> header is added for the definitions of **dev_t**, **ino_t**, **mode_t**,
- 36039 **nlink_t**, **uid_t**, **gid_t**, **off_t** and **time_t**. This has been marked as an extension.
- 36040 • References to the S_IREAD, S_IWRITE, S_IEXEC file and S_ISVTX modes are removed.
- 36041 • The descriptions of the members of the **stat** structure in the DESCRIPTION are corrected.

36042 **Issue 4, Version 2**

36043 The following changes are incorporated for X/OPEN UNIX conformance:

- 36044 • The **st_blksize** and **st_blocks** members are added to the **stat** structure.
- 36045 • The S_IFLINK value of S_IFMT is defined.
- 36046 • The S_ISVTX file mode bit and the S_ISLNK file type test macro is defined.
- 36047 • The *fchmod()*, *lstat()* and *mknod()* functions are added to the list of functions declared in this
- 36048 header.

36049 **Issue 5**

36050 The DESCRIPTION is updated for alignment with POSIX Realtime Extension.

36051 The type of *st_blksize* is changed from **long** to **blksize_t**; the type of *st_blocks* is changed from **long**

36052 to **blkcnt_t**.

36053 NAME

36054 sys/statvfs.h — VFS Filesystem information structure

36055 SYNOPSIS

36056 EX #include <sys/statvfs.h>

36057

36058 DESCRIPTION

36059 The <sys/statvfs.h> header defines the **statvfs** structure that includes at least the following
36060 members:

36061	unsigned long	f_bsize	file system block size
36062	unsigned long	f_frsize	fundamental filesystem block size
36063	fsblkcnt_t	f_blocks	total number of blocks on file system in units of f_frsize
36064	fsblkcnt_t	f_bfree	total number of free blocks
36065	fsblkcnt_t	f_bavail	number of free blocks available to
36066			non-privileged process
36067	fsfilcnt_t	f_files	total number of file serial numbers
36068	fsfilcnt_t	f_ffree	total number of free file serial numbers
36069	fsfilcnt_t	f_favail	number of file serial numbers available to
36070			non-privileged process
36071	unsigned long	f_fsid	file system id
36072	unsigned long	f_flag	bit mask of f_flag values
36073	unsigned long	f_namemax	maximum filename length

36074 The following flags for the **f_flag** member are defined:

36075	ST_RDONLY	read-only file system
36076	ST_NOSUID	does not support setuid/setgid semantics

36077 The header <sys/statvfs.h> declares the following functions which may also be defined as
36078 macros. Function prototypes must be provided for use with an ISO C compiler.

```
36079 int statvfs(const char *, struct statvfs *);
36080 int fstatvfs(int, struct statvfs *);
```

36081 APPLICATION USAGE

36082 None.

36083 FUTURE DIRECTIONS

36084 None.

36085 SEE ALSO

36086 *fstatvfs()*, *statvfs()*.

36087 CHANGE HISTORY

36088 First released in Issue 4, Version 2.

36089 Issue 5

36090 The type of *f_blocks*, *f_bfree* and *f_bavail* is changed from **unsigned long** to **fsblkcnt_t**; the type of
36091 *f_files*, *f_ffree* and *f_favail* is changed from **unsigned long** to **fsfilcnt_t**.

36092 NAME

36093 sys/time.h — time types

36094 SYNOPSIS

36095 EX `#include <sys/time.h>`

36096

36097 DESCRIPTION

36098 The <sys/time.h> header defines the **timeval** structure that includes at least the following
36099 members:

36100	time_t	tv_sec	seconds
36101	suseconds_t	tv_usec	microseconds

36102 The <sys/time.h> header defines the **itimerval** structure that includes at least the following
36103 members:

36104	struct timeval it_interval	timer interval
36105	struct timeval it_value	current value

36106 The **time_t** and **suseconds_t** types are defined as described in <sys/types.h>.

36107 The <sys/time.h> header defines the **fd_set** type as a structure that includes at least the
36108 following member:

36109 long fds_bits[] bit mask for open file descriptions

36110 The <sys/time.h> header defines the following values for the *which* argument of *getitimer()* and
36111 *setitimer()*:

36112	ITIMER_REAL	Decrements in real time.
36113	ITIMER_VIRTUAL	Decrements in process virtual time.
36114	ITIMER_PROF	Decrements both in process virtual time and when the system is running
36115		on behalf of the process.

36116 Each of the following may be declared as a function, or defined as a macro, or both:

36117 void FD_CLR(int fd, fd_set *fdset)
36118 Clears the bit for the file descriptor *fd* in the file descriptor set *fdset*.

36119 int FD_ISSET(int fd, fd_set *fdset)
36120 Returns a non-zero value if the bit for the file descriptor *fd* is set in the file descriptor set by
36121 *fdset*, and 0 otherwise.

36122 void FD_SET(int fd, fd_set *fdset)
36123 Sets the bit for the file descriptor *fd* in the file descriptor set *fdset*.

36124 void FD_ZERO(fd_set *fdset)
36125 Initialises the file descriptor set *fdset* to have zero bits for all file descriptors.

36126 FD_SETSIZE
36127 Maximum number of file descriptors in an **fd_set** structure.

36128 If implemented as macros, these may evaluate their arguments more than once, so that
36129 arguments must never be expressions with side effects.

36130 The following are declared as functions and may also be defined as macros. Function prototypes
36131 must be provided for use with an ISO C compiler.

36132	int	getitimer(int, struct itimerval *);
36133	int	setitimer(int, const struct itimerval *, struct itimerval *);
36134	int	gettimeofday(struct timeval *, void *);

36135 int select(int, fd_set *, fd_set *, fd_set *, struct timeval *);

36136 int utimes(const char *, const struct timeval [2]);

36137 **APPLICATION USAGE**

36138 None.

36139 **FUTURE DIRECTIONS**

36140 None.

36141 **SEE ALSO**

36142 *getitimer()*, *gettimeofday()*, *select()*, *setitimer()*, *utimes()*.

36143 **CHANGE HISTORY**

36144 First released in Issue 4, Version 2.

36145 **Issue 5**

36146 The type of *tv_usec* is changed from **long** to **suseconds_t**.

36147 NAME

36148 sys/timeb.h — additional definitions for date and time

36149 SYNOPSIS

36150 EX `#include <sys/timeb.h>`

36151

36152 DESCRIPTION

36153 The <sys/timeb.h> header defines the **timeb** structure that includes at least the following
36154 members:

36155	time_t	time	the seconds portion of the current time
36156	unsigned short	millitm	the milliseconds portion of the current time
36157	short	timezone	the local timezone in minutes west of Greenwich
36158	short	dstflag	TRUE if Daylight Savings Time is in effect

36159 The **time_t** type is defined as described in <sys/types.h>.

36160 The header <sys/timeb.h> declares the following as a function which may also be defined as a
36161 macro. Function prototypes must be provided for use with an ISO C compiler.

36162 `int ftime(struct timeb *);`

36163 APPLICATION USAGE

36164 None.

36165 FUTURE DIRECTIONS

36166 None.

36167 SEE ALSO

36168 `ftime()`, <time.h>.

36169 CHANGE HISTORY

36170 First released in Issue 4, Version 2.

36171 **NAME**

36172 sys/times.h — file access and modification times structure

36173 **SYNOPSIS**

36174 #include <sys/times.h>

36175 **DESCRIPTION**36176 The <sys/times.h> header defines the structure **tms**, which is returned by *times()* and includes at
36177 least the following members:

36178 clock_t tms_utime user CPU time

36179 clock_t tms_stime system CPU time

36180 clock_t tms_cutime user CPU time of terminated child processes

36181 clock_t tms_cstime system CPU time of terminated child processes

36182 The **clock_t** type is defined as described in <sys/types.h>.36183 The following is declared as a function and may also be defined as a macro. Function prototypes
36184 must be provided for use with an ISO C compiler.

36185 clock_t times(struct tms *);

36186 **APPLICATION USAGE**

36187 None.

36188 **FUTURE DIRECTIONS**

36189 None.

36190 **SEE ALSO**36191 *times()*, <sys/types.h>.36192 **CHANGE HISTORY**

36193 First released in Issue 1.

36194 Derived from Issue 1 of the SVID.

36195 **Issue 4**

36196 The following change is incorporated for alignment with the ISO POSIX-1 standard:

36197 • The function declarations in this header are expanded to full ISO C prototypes.

36198 Other changes are incorporated as follows:

36199 • Reference to the <sys/types.h> header is added for the definitions of **clock_t**.36200 • This issue states that the *times()* function can also be defined as a macro.

36201 NAME

36202 sys/types.h — data types

36203 SYNOPSIS

36204 #include <sys/types.h>

36205 DESCRIPTION

36206 The <sys/types.h> header includes definitions for at least the following types:

36207	blkcnt_t	Used for file block counts
36208	blksize_t	Used for block sizes
36209 EX	clock_t	Used for system times in clock ticks or CLOCKS_PER_SEC (see <time.h>).
36210		
36211 RT	clockid_t	Used for clock ID type in the clock and timer functions.
36212	dev_t	Used for device IDs.
36213 EX	fsblkcnt_t	Used for file system block counts
36214	fsfilcnt_t	Used for file system file counts
36215	gid_t	Used for group IDs.
36216 EX	id_t	Used as a general identifier; can be used to contain at least a pid_t , uid_t or a gid_t .
36217		
36218	ino_t	Used for file serial numbers.
36219 EX	key_t	Used for interprocess communication.
36220	mode_t	Used for some file attributes.
36221	nlink_t	Used for link counts.
36222	off_t	Used for file sizes.
36223	pid_t	Used for process IDs and process group IDs.
36224	pthread_attr_t	Used to identify a thread attribute object.
36225	pthread_cond_t	Used for condition variables.
36226	pthread_condattr_t	Used to identify a condition attribute object.
36227	pthread_key_t	Used for thread-specific data keys.
36228	pthread_mutex_t	Used for mutexes.
36229	pthread_mutexattr_t	Used to identify a mutex attribute object.
36230	pthread_once_t	Used for dynamic package initialisation.
36231 EX	pthread_rwlock_t	Used for read-write locks.
36232	pthread_rwlockattr_t	Used for read-write lock attributes.
36233	pthread_t	Used to identify a thread.
36234	size_t	Used for sizes of objects.
36235	ssize_t	Used for a count of bytes or an error indication.
36236 EX	suseconds_t	Used for time in microseconds
36237	time_t	Used for time in seconds.
36238 RT	timer_t	Used for timer ID returned by <i>timer_create()</i> .
36239	uid_t	Used for user IDs.
36240 EX	useconds_t	Used for time in microseconds.

36241 All of the types are defined as arithmetic types of an appropriate length, with the following
 36242 EX exceptions: **key_t**, **pthread_attr_t**, **pthread_cond_t**, **pthread_condattr_t**, **pthread_key_t**,
 36243 EX **pthread_mutex_t**, **pthread_mutexattr_t**, **pthread_once_t**, **pthread_rwlock_t** and
 36244 EX **pthread_rwlockattr_t**. Additionally, **blkcnt_t** and **off_t** are extended signed integral types,
 36245 EX **fsblkcnt_t**, **fsfilcnt_t** and **ino_t** are defined as extended unsigned integral types, **size_t** is an
 36246 unsigned integral type, and **blksize_t**, **pid_t** and **ssize_t** are signed integral types. The type
 36247 EX **ssize_t** is capable of storing values at least in the range [−1, SSIZE_MAX]. The type **useconds_t**
 36248 is an unsigned integral type capable of storing values at least in the range [0, 1,000,000]. The
 36249 type **suseconds_t** is a signed integral type capable of storing values at least in the range [−1,
 36250 1,000,000].

36251 There are no defined comparison or assignment operators for the types **pthread_attr_t**,
 36252 EX **pthread_cond_t**, **pthread_condattr_t**, **pthread_mutex_t**, **pthread_mutexattr_t**, **pthread_rwlock_t**
 36253 and **pthread_rwlockattr_t**.

36254 APPLICATION USAGE

36255 None.

36256 FUTURE DIRECTIONS

36257 None.

36258 SEE ALSO

36259 None.

36260 CHANGE HISTORY

36261 First released in Issue 1.

36262 Derived from Issue 1 of the SVID.

36263 Issue 4

36264 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- 36265 • The data type **ssize_t** is added.
- 36266 • The DESCRIPTION is expanded to indicate the required arithmetic types.

36267 Other changes are incorporated as follows:

- 36268 • The **clock_t** type is marked as an extension.
- 36269 • In the last paragraph of the DESCRIPTION, only the reference to type **key_t** is now marked
 36270 as an extension.

36271 Issue 4, Version 2

36272 The **id_t** and **useconds_t** types are defined for X/OPEN UNIX conformance. The capability of
 36273 the **useconds_t** type is described.

36274 Issue 5

36275 The **clockid_t** and **timer_t** types are defined for alignment with the POSIX Realtime Extension.

36276 Added the types **blkcnt_t**, **blksize_t**, **fsblkcnt_t**, **fsfilcnt_t** and **suseconds_t**.

36277 Large File System extensions added.

36278 Updated for alignment with the POSIX Threads Extension.

36279 **NAME**

36280 sys/uio.h — definitions for vector I/O operations

36281 **SYNOPSIS**

36282 EX `#include <sys/uio.h>`

36283

36284 **DESCRIPTION**

36285 The <sys/uio.h> header defines the **iovec** structure that includes at least the following members:

36286 void *iov_base base address of a memory region for input or output

36287 size_t iov_len the size of the memory pointed to by *iov_base*

36288 The following are declared as functions and may also be defined as macros. Function prototypes
36289 must be provided for use with an ISO C compiler.

36290 ssize_t readv(int, const struct iovec *, int);

36291 ssize_t writev(int, const struct iovec *, int);

36292 **APPLICATION USAGE**

36293 None.

36294 **FUTURE DIRECTIONS**

36295 None.

36296 **SEE ALSO**

36297 *read()*, *write()*.

36298 **CHANGE HISTORY**

36299 First released in Issue 4, Version 2.

36300 **NAME**

36301 sys/utsname.h — system name structure

36302 **SYNOPSIS**

36303 #include <sys/utsname.h>

36304 **DESCRIPTION**36305 The <sys/utsname.h> header defines structure **utsname**, which includes at least the following
36306 members:

36307 char sysname[] name of this implementation of the operating system
 36308 char nodename[] name of this node within an implementation-dependent
 36309 communications network
 36310 char release[] current release level of this implementation
 36311 char version[] current version level of this release
 36312 char machine[] name of the hardware type on which the system is running

36313 The character arrays are of unspecified size, but the data stored in them is terminated by a null
36314 byte.

36315 The following is declared as a function and may also be defined as a macro.

36316 int uname (struct utsname *);

36317 **APPLICATION USAGE**

36318 None.

36319 **FUTURE DIRECTIONS**

36320 None.

36321 **SEE ALSO**36322 *uname()*.36323 **CHANGE HISTORY**

36324 First released in Issue 1.

36325 Derived from Issue 1 of the SVID.

36326 **Issue 4**

36327 The following change is incorporated for alignment with the ISO C standard:

- 36328 • The function declarations in this header are expanded to full ISO C prototypes.

36329 Other changes are incorporated as follows:

- 36330 • The word “character” is replaced with the word “byte” in the DESCRIPTION.
- 36331 • The function in this header can now also be defined as a macro.

36332 NAME

36333 sys/wait.h — declarations for waiting

36334 SYNOPSIS

36335 #include <sys/wait.h>

36336 DESCRIPTION

36337 The <sys/wait.h> header defines the following symbolic constants for use with *waitpid()*:

36338	WNOHANG	Do not hang if no status is available, return immediately.
36339	WUNTRACED	Report status of stopped child process.

36340 and the following macros for analysis of process status values:

36341	WEXITSTATUS ()	Return exit status.
36342 EX	WIFCONTINUED ()	True if child has been continued
36343	WIFEXITED ()	True if child exited normally.
36344	WIFSIGNALED ()	True if child exited due to uncaught signal.
36345	WIFSTOPPED ()	True if child is currently stopped.
36346	WSTOPSIG ()	Return signal number that caused process to stop.
36347	WTERMSIG ()	Return signal number that caused process to terminate.

36348 EX The following symbolic constants are defined as possible values for the *options* argument to
36349 *waitid()*:

36350	WEXITED	Wait for processes that have exited.
36351	WSTOPPED	Status will be returned for any child that has stopped upon receipt of a signal.
36352		
36353	WCONTINUED	Status will be returned for any child that was stopped and has been continued.
36354		
36355	WNOHANG	Return immediately if there are no children to wait for.
36356	WNOWAIT	Keep the process whose status is returned in <i>infop</i> in a waitable state.

36357 The type **idtype_t** is defined as an enumeration type whose possible values include at least the
36358 following:

36359	P_ALL
36360	P_PID
36361	P_PGID

36362 The **id_t** type is defined as described in <sys/types.h>.

36363 The **siginfo_t** type is defined as described in <signal.h>.

36364 The **rusage** structure is defined as described in <sys/resource.h>.

36365 The **pid_t** type is defined as described in <sys/types.h>.

36366 Inclusion of the <sys/wait.h> header may also make visible all symbols from <signal.h> and
36367 <sys/resource.h>.

36368 The following are declared as functions and may also be defined as macros. Function prototypes
36369 must be provided for use with an ISO C compiler.

36370	pid_t	wait(int *);
36371 EX	pid_t	wait3(int *, int, struct rusage *);
36372	int	waitid(idtype_t, id_t, siginfo_t *, int);
36373	pid_t	waitpid(pid_t, int *, int);

36374 **APPLICATION USAGE**

36375 None.

36376 **FUTURE DIRECTIONS**

36377 None.

36378 **SEE ALSO**36379 *wait()*, *waitid()*. <sys/resource.h>, <sys/types.h>.36380 **CHANGE HISTORY**

36381 First released in Issue 3.

36382 Entry included for alignment with the POSIX.1-1988 standard.

36383 **Issue 4**

36384 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 36385
- The function declarations in this header are expanded to full ISO C prototypes.

36386 Another change is incorporated as follows:

- 36387
- Reference to the <sys/types.h> header is added for the definition of **pid_t** and marked as an
-
- 36388 extension.

36389 **Issue 4, Version 2**

36390 The following changes are incorporated for X/OPEN UNIX conformance:

- 36391
- The WIFCONTINUED macro, the list of symbolic constants for the *options* argument to
-
- 36392
- waitid()*
- , and the description of the
- idtype_t**
- enumeration type are added.

- 36393
- A statement is added indicated that inclusion of this header may also make visible constants
-
- 36394 from <signal.h> and <sys/resource.h>.

- 36395
- The *wait3()* and *waitid()* functions are added to the list of functions declared in this header.

36396 NAME

36397 tar.h — extended tar definitions

36398 SYNOPSIS

36399 #include <tar.h>

36400 DESCRIPTION

36401 Header block definitions are:

36402 General definitions:

36403

36404	Name	Description	Value
36405	TMAGIC	"ustar"	ustar plus null byte.
36406	TMAGLEN	6	Length of the above.
36407	TVERSION	"00"	00 without a null byte.
36408	TVERSLEN	2	Length of the above.

36409 *Typeflag* field definitions:

36410

36411	Name	Description	Value
36412	REGTYPE	'0'	Regular file.
36413	AREGTYPE	' \0'	Regular file.
36414	LNKTYPE	'1'	Link.
36415 EX	SYMTYPE	'2'	Symbolic link.
36416	CHRTYPE	'3'	Character special.
36417	BLKTYPE	'4'	Block special.
36418	DIRTYPE	'5'	Directory.
36419	FIFOTYPE	'6'	FIFO special.
36420	CONTTYPE	'7'	Reserved.

36421 *Mode* field bit definitions (octal):

36422

36423	Name	Description	Value
36424	TSUID	04000	Set UID on execution.
36425	TSGID	02000	Set GID on execution.
36426 EX	TSVTX	01000	On directories, restricted deletion flag.
36427	TUREAD	00400	Read by owner.
36428	TUWRITE	00200	Write by owner special.
36429	TUEXEC	00100	Execute/search by owner.
36430	TGREAD	00040	Read by group.
36431	TGWRITE	00020	Write by group.
36432	TGEXEC	00010	Execute/search by group.
36433	TOREAD	00004	Read by other.
36434	TOWRITE	00002	Write by other.
36435	TOEXEC	00001	Execute/search by other.

36436 APPLICATION USAGE

36437 None.

36438 FUTURE DIRECTIONS

36439 None.

36440 **SEE ALSO**36441 The **XCU** specification, *tar*.36442 **CHANGE HISTORY**

36443 First released in Issue 3.

36444 Derived from the entry in the POSIX.1-1988 standard.

36445 **Issue 4**36446 This entry is moved from the referenced **Headers** specification.36447 **Issue 4, Version 2**

36448 The following changes are incorporated for X/OPEN UNIX conformance:

- 36449 • The significance of SYMTYPE as the value of the *typeflag* field is explained.
- 36450 • The value of TSVTX as the value of the *mode* field is explained.

36451 NAME

36452 termios.h — define values for termios

36453 SYNOPSIS

36454 #include <termios.h>

36455 DESCRIPTION

36456 The <termios.h> header contains the definitions used by the terminal I/O interfaces (see the
36457 XBD specification, **Chapter 9, General Terminal Interface** for the structures and names
36458 defined).

36459 The termios Structure

36460 The following data types are defined through **typedef**:

36461 **cc_t** Used for terminal special characters.

36462 **speed_t** Used for terminal baud rates.

36463 **tcflag_t** Used for terminal modes.

36464 The above types are all unsigned integral types.

36465 The **termios** structure is defined, and includes at least the following members:

36466	tcflag_t	c_iflag	input modes
36467	tcflag_t	c_oflag	output modes
36468	tcflag_t	c_cflag	control modes
36469	tcflag_t	c_lflag	local modes
36470	cc_t	c_cc[NCCS]	control chars

36471 A definition is given for:

36472 NCCS Size of the array **c_cc** for control characters.

36473 The following subscript names for the array **c_cc** are defined:

36474

36475

36476

36477

36478

36479

36480

36481

36482

36483

36484

36485

36486

36487

Subscript Usage		Description
Canonical Mode	Non-canonical Mode	
VEOF	VINTR	EOF character
VEOL		EOL character
VERASE		ERASE character
VINTR		INTR character
VKILL		KILL character
	VMIN	MIN value
VQUIT	VQUIT	QUIT character
VSTART	VSTART	START character
VSTOP	VSTOP	STOP character
VSUSP	VSUSP	SUSP character
	VTIME	TIME value

36488 The subscript values are unique, except that the VMIN and VTIME subscripts may have the
36489 same values as the VEOF and VEOL subscripts, respectively.

36490 **Input Modes**36491 The **c_iflag** field describes the basic terminal input control:

36492	BRKINT	Signal interrupt on break.
36493	ICRNL	Map CR to NL on input.
36494	IGNBRK	Ignore break condition.
36495	IGNCR	Ignore CR
36496	IGNPAR	Ignore characters with parity errors.
36497	INLCR	Map NL to CR on input.
36498	INPCK	Enable input parity check.
36499	ISTRIP	Strip character
36500 EX	IUCLC	Map upper-case to lower-case on input (LEGACY).
36501	IXANY	Enable any character to restart output.
36502	IXOFF	Enable start/stop input control.
36503	IXON	Enable start/stop output control.
36504	PARMRK	Mark parity errors.

36505 **Output Modes**36506 The **c_oflag** field specifies the system treatment of output:

36507	OPOST	Post-process output
36508 EX	OLCUC	Map lower-case to upper-case on output (LEGACY).
36509	ONLCR	Map NL to CR-NL on output.
36510	OCRNL	Map CR to NL on output.
36511	ONOCR	No CR output at column 0.
36512	ONLRET	NL performs CR function.
36513	OFILL	Use fill characters for delay.
36514	NLDLY	Select newline delays:
36515	NL0	Newline character type 0.
36516	NL1	Newline character type 1.
36517	CRDLY	Select carriage-return delays:
36518	CR0	Carriage-return delay type 0.
36519	CR1	Carriage-return delay type 1.
36520	CR2	Carriage-return delay type 2.
36521	CR3	Carriage-return delay type 3.
36522	TABDLY	Select horizontal-tab delays:
36523	TAB0	Horizontal-tab delay type 0.
36524	TAB1	Horizontal-tab delay type 1.
36525	TAB2	Horizontal-tab delay type 2.
36526	TAB3	Expand tabs to spaces.
36527	BSDLY	Select backspace delays:
36528	BS0	Backspace-delay type 0.
36529	BS1	Backspace-delay type 1.
36530	VTDLY	Select vertical-tab delays:
36531	VT0	Vertical-tab delay type 0.
36532	VT1	Vertical-tab delay type 1.

36533	FFDLY	Select form-feed delays:
36534	FF0	Form-feed delay type 0.
36535	FF1	Form-feed delay type 1.
36536	Baud Rate Selection	
36537	The input and output baud rates are stored in the termios structure. These are the valid values	
36538	for objects of type speed_t . The following values are defined, but not all baud rates need be	
36539	supported by the underlying hardware.	
36540	B0	Hang up
36541	B50	50 baud
36542	B75	75 baud
36543	B110	110 baud
36544	B134	134.5 baud
36545	B150	150 baud
36546	B200	200 baud
36547	B300	300 baud
36548	B600	600 baud
36549	B1200	1200 baud
36550	B1800	1800 baud
36551	B2400	2400 baud
36552	B4800	4800 baud
36553	B9600	9600 baud
36554	B19200	19200 baud
36555	B38400	38400 baud
36556	Control Modes	
36557	The c_cflag field describes the hardware control of the terminal; not all values specified are	
36558	required to be supported by the underlying hardware:	
36559	CSIZE	Character size:
36560	CS5	5 bits.
36561	CS6	6 bits.
36562	CS7	7 bits.
36563	CS8	8 bits.
36564	CSTOPB	Send two stop bits, else one.
36565	CREAD	Enable receiver.
36566	PARENB	Parity enable.
36567	PARODD	Odd parity, else even.
36568	HUPCL	Hang up on last close.
36569	CLOCAL	Ignore modem status lines.
36570	Local Modes	
36571	The c_lflag field of the argument structure is used to control various terminal functions:	
36572	ECHO	Enable echo.
36573	ECHOE	Echo erase character as error-correcting backspace.
36574	ECHOK	Echo KILL.
36575	ECHONL	Echo NL.
36576	ICANON	Canonical input (erase and kill processing).
36577	IEXTEN	Enable extended input character processing.

36578	ISIG	Enable signals.
36579	NOFLSH	Disable flush after interrupt or quit.
36580	TOSTOP	Send SIGTTOU for background output.
36581 EX	XCASE	Canonical upper/lower presentation (LEGACY).

36582 Attribute Selection

36583 The following symbolic constants for use with *tcsetattr()* are defined:

36584	TCSANOW	Change attributes immediately.
36585	TCSADRAIN	Change attributes when output has drained.
36586	TCSAFLUSH	Change attributes when output has drained; also flush pending input.

36587 Line Control

36588 The following symbolic constants for use with *tcflush()* are defined:

36589	TCIFLUSH	Flush pending input. Flush untransmitted output.
36590	TCIOFLUSH	Flush both pending input and untransmitted output.

36591 The following symbolic constants for use with *tcflow()* are defined:

36592	TCIOFF	Transmit a STOP character, intended to suspend input data.
36593	TCION	Transmit a START character, intended to restart input data.
36594	TCOOFF	Suspend output.
36595	TCOON	Restart output.

36596 The following are declared as functions and may also be defined as macros. Function prototypes
36597 must be provided for use with an ISO C compiler.

```

36598 speed_t cfgetispeed(const struct termios *);
36599 speed_t cfgetospeed(const struct termios *);
36600 int      cfsetispeed(struct termios *, speed_t);
36601 int      cfsetospeed(struct termios *, speed_t);
36602 int      tcdrain(int);
36603 int      tcflow(int, int);
36604 int      tcflush(int, int);
36605 int      tcgetattr(int, struct termios *);
36606 EX pid_t tcgetsid(int);
36607 int      tcsendbreak(int, int);
36608 int      tcsetattr(int, int, struct termios *);

```

36609 APPLICATION USAGE

36610 The following names are commonly used as extensions to the above, therefore portable
36611 applications must not use them:

36612	CBAUD	EXTB	VDSUSP
36613	DEFECHO	FLUSHO	VLNEXT
36614	ECHOCTL	LOBLK	VREPRINT
36615	ECHOKE	PENDIN	VSTATUS
36616	ECHOPRT	SWTCH	VWERASE
36617	EXTA	VDISCARD	

36618 FUTURE DIRECTIONS

36619 None.

36620 **SEE ALSO**

36621 *cfgetispeed(), cfgetospeed(), cfsetispeed(), cfsetospeed(), tcdrain(), tcflow(), tcflush(), tcgetattr(),*
 36622 *tcgetsid(), tcsendbreak(), tcsetattr(),* the **XBD** specification, **Chapter 9, General Terminal**
 36623 **Interface**.

36624 **CHANGE HISTORY**

36625 First released in Issue 3.

36626 Entry included for alignment with the ISO POSIX-1 standard.

36627 **Issue 4**

36628 The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- 36629 • The function declarations in this header are expanded to full ISO C prototypes.
- 36630 • Some minor rewording of the DESCRIPTION is done to align the text more exactly with the |
 36631 ISO POSIX-1 standard. No functional differences are implied by these changes.
- 36632 • The list of mask name symbols for the *c_oflag* field have all been marked as extensions, with
 36633 the exception of OPOST.

36634 Other changes are incorporated as follows:

- 36635 • The following words are removed from the description of the **c_cc** array:
 36636 “Implementations that do not support the job control option, may ignore the SUSP character
 36637 value in the **c_cc** array indexed by the VSUSP subscript.”
 36638 This is because job control is defined as mandatory for Issue 4 conforming implementations.
- 36639 • The mask name symbols IUCLC and OLCUC are marked **LEGACY**. |

36640 **Issue 4, Version 2**

36641 For X/OPEN UNIX conformance, the *tcgetsid()* function is added to the list of functions declared |
 36642 in this header.

36643 NAME

36644 time.h — time types

36645 SYNOPSIS

36646 #include <time.h>

36647 DESCRIPTION

36648 The <time.h> header declares the structure **tm**, which includes at least the following members:

36649	int	tm_sec	seconds [0,61]
36650	int	tm_min	minutes [0,59]
36651	int	tm_hour	hour [0,23]
36652	int	tm_mday	day of month [1,31]
36653	int	tm_mon	month of year [0,11]
36654	int	tm_year	years since 1900
36655	int	tm_wday	day of week [0,6] (Sunday = 0)
36656	int	tm_yday	day of year [0,365]
36657	int	tm_isdst	daylight savings flag

36658 The value of **tm_isdst** is positive if Daylight Saving Time is in effect, 0 if Daylight Saving Time is
 36659 not in effect, and negative if the information is not available.

36660 This header defines the following symbolic names:

36661	NULL	Null pointer constant.
36662	CLK_TCK	Number of clock ticks per second returned by the <i>times()</i> function (LEGACY).
36663		
36664	CLOCKS_PER_SEC	A number used to convert the value returned by the <i>clock()</i> function into 36665 seconds.

36666 RT The <time.h> header declares the structure **timespec**, which has at least the following members:

36667	time_t	tv_sec	seconds
36668	long	tv_nsec	nanoseconds

36669 This header also declares the **itimerspec** structure, which has at least the following members:

36670	struct timespec	it_interval	timer period
36671	struct timespec	it_value	timer expiration

36672 The following manifest constants are defined:

36673	CLOCK_REALTIME	The identifier of the systemwide realtime clock.
36674	TIMER_ABSTIME	Flag indicating time is absolute with respect to the clock associated with a 36675 timer.
36676		

36677 The **clock_t**, **size_t** and **time_t** types are defined as described in <sys/types.h>.

36678 EX Although the value of CLOCKS_PER_SEC is required to be 1 million on all XSI-conformant
 36679 systems, it may be variable on other systems and it should not be assumed that
 36680 CLOCKS_PER_SEC is a compile-time constant.

36681 The value of CLK_TCK is currently the same as the value of *sysconf*(SC_CLK_TCK); however,
 36682 new applications should call *sysconf*() because the CLK_TCK macro may be withdrawn in a
 36683 future issue.

36684 EX The <time.h> header provides a declaration for *getdate_err*.

36685 The following are declared as functions and may also be defined as macros. Function prototypes
 36686 must be provided for use with an ISO C compiler.

```

36687      char      *asctime(const struct tm *);
36688      char      *asctime_r(const struct tm *, char *);
36689      clock_t    clock(void);
36690 RT      int      clock_getres(clockid_t, struct timespec *);
36691      int      clock_gettime(clockid_t, struct timespec *);
36692      int      clock_settime(clockid_t, const struct timespec *);
36693      char      *ctime(const time_t *);
36694      char      *ctime_r(const time_t *, char *);
36695      double     difftime(time_t, time_t);
36696 EX      struct tm *getdate(const char *);
36697      struct tm *gmtime(const time_t *);
36698      struct tm *gmtime_r(const time_t *, struct tm *);
36699      struct tm *localtime(const time_t *);
36700      struct tm *localtime_r(const time_t *, struct tm *);
36701      time_t     mktime(struct tm *);
36702 RT      int      nanosleep(const struct timespec *, struct timespec *);
36703      size_t     strftime(char *, size_t, const char *, const struct tm *);
36704 EX      char      *strptime(const char *, const char *, struct tm *);
36705      time_t     time(time_t *);
36706 RT      int      timer_create(clockid_t, struct sigevent *, timer_t *);
36707      int      timer_delete(timer_t);
36708      int      timer_gettime(timer_t, struct itimerspec *);
36709      int      timer_getoverrun(timer_t);
36710      int      timer_settime(timer_t, int, const struct itimerspec *,
36711                          struct itimerspec *);
36712      void      tzset(void);

```

36713 The following are declared as variables:

```

36714 EX      extern int      daylight;
36715      extern long int     timezone;
36716      extern char      *tzname[ ];

```

36717 APPLICATION USAGE

36718 The range [0,61] for **tm_sec** allows for the occasional leap second or double leap second.

36719 **tm_year** is a signed value, therefore years before 1900 may be represented.

36720 FUTURE DIRECTIONS

36721 None.

36722 SEE ALSO

36723 *asctime()*, *asctime_r()*, *clock()*, *clock_settime()*, *ctime()*, *ctime_r()*, *daylight*, *difftime()*, *getdate()*,
36724 *gmtime()*, *gmtime_r()*, *localtime()*, *localtime_r()*, *mktime()*, *nanosleep()*, *strftime()*, *strptime()*,
36725 *sysconf()*, *time()*, *timer_create()*, *timer_delete()*, *timer_settime()*, *timezone*, *tzname()*, *tzset()*, *utime()*.

36726 CHANGE HISTORY

36727 First released in Issue 1.

36728 Derived from Issue 1 of the SVID.

36729 Issue 4

36730 The following changes are incorporated for alignment with the ISO C standard:

- 36731 • The function declarations in this header are expanded to full ISO C prototypes.
- 36732 • The range of **tm_min** is changed from [0,61] to [0,59].

- 36733 • Possible settings of **tm_isdst** and their meanings are added.
- 36734 • The functions *clock()* and *difftime()* are added to the list of functions declared in this header.
- 36735 Other changes are incorporated as follows:
- 36736 • The symbolic name CLK_TCK is marked as an extension and **LEGACY**. Warnings about its
36737 use are also added to the DESCRIPTION.
- 36738 • Reference to the header <sys/types.h> is added for the definitions of **clock_t**, **size_t** and
36739 **time_t**.
- 36740 • References to CLK_TCK are changed to CLOCKS_PER_SEC in part of the DESCRIPTION.
36741 The fact that CLOCKS_PER_SEC is always one millionth of a second on XSI-conformant
36742 systems is also marked as an extension.
- 36743 • External declarations for *daylight*, *timezone* and *tzname* are added. The first two are marked as
36744 extensions.
- 36745 • The function *strptime()* is added to the list of functions declared in this header.
- 36746 • A note about the settings of **tm_sec** is added to the APPLICATION USAGE section.
- 36747 **Issue 4, Version 2**
- 36748 The following changes are incorporated for X/OPEN UNIX conformance:
- 36749 • The <time.h> header provides a declaration for *getdate_err*.
- 36750 • The *getdate()* function is added to the list of functions declared in this header.
- 36751 **Issue 5**
- 36752 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
36753 Threads Extension.

36754 **NAME**

36755 ucontext — user context

36756 **SYNOPSIS**

36757 EX #include <ucontext.h>

36758

36759 **DESCRIPTION**

36760 The <ucontext.h> header defines the **mcontext_t** type through **typedef**.

36761 The <ucontext.h> header defines the **ucontext_t** type as a structure that includes at least the
36762 following members:

36763	ucontext_t	*uc_link	pointer to the context that will be resumed
36764			when this context returns
36765	sigset_t	uc_sigmask	the set of signals that are blocked when this
36766			context is active
36767	stack_t	uc_stack	the stack used by this context
36768	mcontext_t	uc_mcontext	a machine-specific representation of the saved
36769			context

36770 The types **sigset_t** and **stack_t** are defined as in <signal.h>.

36771 The following are declared as functions and may also be defined as macros, Function prototypes
36772 must be provided for use with an ISO C compiler.

```
36773       int  getcontext(ucontext_t *);
36774       int  setcontext(const ucontext_t *);
36775       void makecontext(ucontext_t *, (void *)(), int, ...);
36776       int  swapcontext(ucontext_t *, const ucontext_t *);
```

36777 **APPLICATION USAGE**

36778 None.

36779 **FUTURE DIRECTIONS**

36780 None.

36781 **SEE ALSO**

36782 *getcontext()*, *makecontext()*, *sigaction()*, *sigprocmask()*, *sigaltstack()*, <signal.h>.

36783 **CHANGE HISTORY**

36784 First released in Issue 4, Version 2.

36785 NAME

36786 ulimit.h — ulimit commands

36787 SYNOPSIS

36788 EX #include <ulimit.h>

36789

36790 DESCRIPTION

36791 The **<ulimit.h>** header defines the symbolic constants used in the *ulimit()* function.

36792 Symbolic constants:

36793 UL_GETFSIZE Get maximum file size.

36794 UL_SETFSIZE Set maximum file size.

36795 The following is declared as a function and may also be defined as a macro. Function prototypes
36796 must be provided for use with an ISO C compiler.

36797 long int ulimit (int, ...);

36798 APPLICATION USAGE

36799 None.

36800 FUTURE DIRECTIONS

36801 None.

36802 SEE ALSO

36803 *ulimit()*.

36804 CHANGE HISTORY

36805 First released in Issue 3.

36806 Issue 4

36807 The following change is incorporated in this issue:

- 36808
 - The function declarations in this header are expanded to full ISO C prototypes.

36809 NAME

36810 unistd.h — standard symbolic constants and types

36811 SYNOPSIS

36812 #include <unistd.h>

36813 DESCRIPTION

36814 The <unistd.h> header defines miscellaneous symbolic constants and types, and declares
36815 miscellaneous functions. The contents of this header are shown below.

36816 Version Test Macros

36817 The following symbolic constants are defined:

36818 _POSIX_VERSION

36819 Integer value indicating version of the ISO POSIX-1 standard (C language binding).

36820 _POSIX2_VERSION

36821 Integer value indicating version of the ISO POSIX-2 standard (Commands).

36822 _POSIX2_C_VERSION

36823 Integer value indicating version of the ISO POSIX-2 standard (C language binding).

36824 EX _XOPEN_VERSION

36825 Integer value indicating version of the X/Open Portability Guide to which the
36826 implementation conforms.

36827 _POSIX_VERSION is defined in the ISO POSIX-1 standard. It changes with each new version of
36828 the ISO POSIX-1 standard.

36829 _POSIX2_VERSION is defined in the ISO POSIX-2 standard. It changes with each new version
36830 of the ISO POSIX-2 standard.

36831 _POSIX2_C_VERSION is defined in the ISO POSIX-2 standard. It changes with each new
36832 EX version of the ISO POSIX-2 standard. When the C language binding option of the ISO POSIX-2
36833 standard and therefore the X/Open POSIX2 C-language Binding Feature Group is not
36834 supported, _POSIX2_C_VERSION will be set to -1.

36835 _XOPEN_VERSION is defined as an integer value equal to 500.

36836 _XOPEN_XCU_VERSION is defined as an integer value indicating the version of the XCU
36837 specification to which the implementation conforms. If the value is -1, no commands and
36838 utilities are provided on the implementation. If the value is greater than or equal to 4, the
36839 functionality associated with the following symbols is also supported (see **Mandatory Symbolic**
36840 **Constants** on page 1196 and **Constants for Options and Feature Groups** on page 1197):

36841 _POSIX2_C_BIND

36842 _POSIX2_C_VERSION

36843 _POSIX2_CHAR_TERM

36844 _POSIX2_LOCALEDEF

36845 _POSIX2_UPE

36846 _POSIX2_VERSION

36847 If this constant is not defined use the *sysconf()* function to determine which features are
36848 supported.

Each of the following symbolic constants is defined only if the implementation supports the indicated issue of the X/Open Portability Guide:

_XOPEN_XPG2
X/Open Portability Guide, Volume 2, January 1987, XVS System Calls and Libraries (ISBN: 0-444-70175-3).

_XOPEN_XPG3
X/Open Specification, February 1992, System Interfaces and Headers, Issue 3 (ISBN: 1-872630-37-5, C212); this specification was formerly X/Open Portability Guide, Issue 3, Volume 2, January 1989, XSI System Interface and Headers (ISBN: 0-13-685843-0, XO/XPG/89/003).

_XOPEN_XPG4
X/Open CAE Specification, July 1992, System Interfaces and Headers, Issue 4 (ISBN: 1-872630-47-2, C202).

_XOPEN_UNIX
X/Open CAE Specification, January 1997, System Interfaces and Headers, Issue 5 (ISBN: 1-85912-181-0, C606).

36865 **Mandatory Symbolic Constants**

Although all implementations conforming to this specification support all of the FIPS features described below, there may be system-dependent or file-system-dependent configuration procedures that can remove or modify any or all of these features. Such configurations should not be made if strict FIPS compliance is required.

The following symbolic constants are either undefined or defined with a value other than -1. If a constant is undefined, an application should use the *sysconf()*, *pathconf()* or *fpathconf()* functions to determine which features are present on the system at that time or for the particular pathname in question.

_POSIX_CHOWN_RESTRICTED
The use of *chown()* is restricted to a process with appropriate privileges, and to changing the group ID of a file only to the effective group ID of the process or to one of its supplementary group IDs.

_POSIX_NO_TRUNC
Pathname components longer than {NAME_MAX} generate an error.

_POSIX_VDISABLE
Terminal special characters defined in <termios.h> can be disabled using this character value.

_POSIX_SAVED_IDS
Each process has a saved set-user-ID and a saved set-group-ID.

_POSIX_JOB_CONTROL
Implementation supports job control.

_POSIX_CHOWN_RESTRICTED, **_POSIX_NO_TRUNC** and **_POSIX_VDISABLE** will have values other than -1.

36889	The following symbolic constants are always defined to unspecified values to indicate that this	
36890	functionality from the POSIX Threads Extension is always present on XSI-conformant systems:	
36891	<code>_POSIX_THREADS</code>	
36892	The implementation supports the threads option.	
36893	<code>_POSIX_THREAD_ATTR_STACKADDR</code>	
36894	The implementation supports the thread stack address attribute option.	
36895	<code>_POSIX_THREAD_ATTR_STACKSIZE</code>	
36896	The implementation supports the thread stack size attribute option.	
36897	<code>_POSIX_THREAD_PROCESS_SHARED</code>	
36898	The implementation supports the process-shared synchronisation option.	
36899	<code>_POSIX_THREAD_SAFE_FUNCTIONS</code>	
36900	The implementation supports the thread-safe functions option.	
36901	Constants for Options and Feature Groups	
36902	The following symbolic constants are defined to have the value <code>-1</code> if the implementation will	
36903	never provide the feature, and to have a value other than <code>-1</code> if the implementation always	
36904	provides the feature. If these are undefined, the <code>sysconf()</code> function can be used to determine	
36905	whether the feature is provided for a particular invocation of the application.	
36906	<code>_POSIX2_C_BIND</code>	
36907	Implementation supports the C Language Binding option. This will always have a value	
36908	other than <code>-1</code> .	
36909	<code>_POSIX2_C_DEV</code>	
36910	Implementation supports the C Language Development Utilities option.	
36911	<code>_POSIX2_CHAR_TERM</code>	
36912	Implementation supports at least one terminal type.	
36913	<code>_POSIX2_FORT_DEV</code>	
36914	Implementation supports the FORTRAN Development Utilities option.	
36915	<code>_POSIX2_FORT_RUN</code>	
36916	Implementation supports the FORTRAN Run-time Utilities option.	
36917 EX	<code>_POSIX2_LOCALEDEF</code>	
36918	Implementation supports the creation of locales by the <i>localedef</i> utility.	
36919	<code>_POSIX2_SW_DEV</code>	
36920	Implementation supports the Software Development Utilities option.	
36921	<code>_POSIX2_UPE</code>	
36922	The implementation supports the User Portability Utilities option.	
36923 EX	<code>_XOPEN_CRYPT</code>	
36924	The implementation supports the X/Open Encryption Feature Group.	
36925	<code>_XOPEN_ENH_I18N</code>	
36926	The implementation supports the Issue 4, Version 2 Enhanced Internationalisation Feature	
36927	Group. This is always set to a value other than <code>-1</code> .	
36928	<code>_XOPEN_LEGACY</code>	
36929	The implementation supports the Legacy Feature Group.	

36930	_XOPEN_REALTIME
36931	The implementation supports the X/Open Realtime Feature Group.
36932	_XOPEN_REALTIME_THREADS
36933	The implementation supports the X/Open Realtime Threads Feature Group.
36934	_XOPEN_SHM
36935	The implementation supports the Issue 4, Version 2 Shared Memory Feature Group. This is
36936	always set to a value other than -1.
36937	_XBS5_ILP32_OFF32
36938	Implementation provides a C-language compilation environment with 32-bit int , long ,
36939	pointer and off_t types.
36940	_XBS5_ILP32_OFFBIG
36941	Implementation provides a C-language compilation environment with 32-bit int , long and
36942	pointer types and an off_t type using at least 64 bits.
36943	_XBS5_LP64_OFF64
36944	Implementation provides a C-language compilation environment with 32-bit int and 64-bit
36945	long , pointer and off_t types.
36946	_XBS5_LPBIG_OFFBIG
36947	Implementation provides a C-language compilation environment with an int type using at
36948	least 32 bits and long , pointer and off_t types using at least 64 bits.
36949 RT	If _XOPEN_REALTIME is defined to have a value other than -1, then the following symbolic
36950	constants will be defined to an unspecified value to indicate that the features are supported.
36951	_POSIX_ASYNCHRONOUS_IO
36952	Implementation supports the Asynchronous Input and Output option.
36953	_POSIX_MEMLOCK
36954	Implementation supports the Process Memory Locking option.
36955	_POSIX_MEMLOCK_RANGE
36956	Implementation supports the Range Memory Locking option.
36957	_POSIX_MESSAGE_PASSING
36958	Implementation supports the Message Passing option.
36959	_POSIX_PRIORITY_SCHEDULING
36960	Implementation supports the Process Scheduling option.
36961	_POSIX_REALTIME_SIGNALS
36962	Implementation supports the Realtime Signals Extension option.
36963	_POSIX_SEMAPHORES
36964	Implementation supports the Semaphores option.
36965	_POSIX_SHARED_MEMORY_OBJECTS
36966	Implementation supports the Shared Memory Objects option.
36967	_POSIX_SYNCHRONIZED_IO
36968	Implementation supports the Synchronised Input and Output option.
36969	_POSIX_TIMERS
36970	Implementation supports the Timers option.
36971	

36972	The following symbolic constants are always defined to unspecified values to indicate that the	
36973	functionality is always present on XSI-conformant systems.	
36974	<code>_POSIX_FSYNC</code>	
36975	Implementation supports the File Synchronisation option.	
36976	<code>_POSIX_MAPPED_FILES</code>	
36977	Implementation supports the Memory Mapped Files option.	
36978	<code>_POSIX_MEMORY_PROTECTION</code>	
36979	Implementation supports the Memory Protection option.	
36980	The following symbolic constant will be defined if the option is supported; otherwise, it will be	
36981	undefined:	
36982	<code>_POSIX_PRIORITIZED_IO</code>	
36983	Implementation supports the Prioritized Input and Output option.	
36984 RTT	If <code>_XOPEN_REALTIME_THREADS</code> is defined to have a value other than <code>-1</code> , then the following	
36985	symbolic constants will be defined to an unspecified value to indicate that the features are	
36986	supported:	
36987	<code>_POSIX_THREAD_PRIORITY_SCHEDULING</code>	
36988	The implementation supports the thread execution scheduling option.	
36989	<code>_POSIX_THREAD_PRIO_INHERIT</code>	
36990	The implementation supports the priority inheritance option.	
36991	<code>_POSIX_THREAD_PRIO_PROTECT</code>	
36992	The implementation supports the priority protection option.	
36993		
36994	Execution-time Symbolic Constants	
36995 RT	If any of the following constants are not defined in the header <code><unistd.h></code> , the value varies	
36996	depending on the file to which it is applied.	
36997	If any of the following constants are defined to have value <code>-1</code> in the header <code><unistd.h></code> , the	
36998	implementation will not provide the option on any file; if any are defined to have a value other	
36999	than <code>-1</code> in the header <code><unistd.h></code> , the implementation will provide the option on all applicable	
37000	files.	
37001	All of the following constants, whether defined in <code><unistd.h></code> or not, may be queried with	
37002	respect to a specific file using the <code>pathconf()</code> or <code>fpathconf()</code> functions.	
37003	<code>_POSIX_ASYNC_IO</code>	
37004	Asynchronous input or output operations may be performed for the associated file.	
37005	<code>_POSIX_PRIO_IO</code>	
37006	Prioritized input or output operations may be performed for the associated file.	
37007	<code>_POSIX_SYNC_IO</code>	
37008	Synchronised input or output operations may be performed for the associated file.	
37009		

37010 **Constants for Functions**

37011 The following symbolic constant is defined:

37012 NULL Null pointer

37013 The following symbolic constants are defined for the *access()* function:

37014 R_OK Test for read permission.

37015 W_OK Test for write permission.

37016 X_OK Test for execute (search) permission.

37017 F_OK Test for existence of file.

37018 The constants F_OK, R_OK, W_OK and X_OK and the expressions *R_OK / W_OK*,
37019 *R_OK / X_OK* and *R_OK / W_OK / X_OK* all have distinct values.37020 The following symbolic constants are defined for the *confstr()* function:

37021 _CS_PATH

37022 If the ISO POSIX-2 standard is supported, this is the value for the *PATH* environment
37023 variable that finds all standard utilities. Otherwise the meaning of this value is unspecified.

37024 EX _CS_XBS5_ILP32_OFF32_CFLAGS

37025 If *sysconf(_SC_XBS5_ILP32_OFF32)* returns -1, the meaning of this value is unspecified.
37026 Otherwise, this value is the set of initial options to be given to the *cc* and *c89* utilities to
37027 build an application using a programming model with 32-bit int, long, pointer, and off_t
37028 types.

37029 _CS_XBS5_ILP32_OFF32_LDFLAGS

37030 If *sysconf(_SC_XBS5_ILP32_OFF32)* returns -1, the meaning of this value is unspecified.
37031 Otherwise, this value is the set of final options to be given to the *cc* and *c89* utilities to build
37032 an application using a programming model with 32-bit int, long, pointer, and off_t types.

37033 _CS_XBS5_ILP32_OFF32_LIBS

37034 If *sysconf(_SC_XBS5_ILP32_OFF32)* returns -1, the meaning of this value is unspecified.
37035 Otherwise, this value is the set of libraries to be given to the *cc* and *c89* utilities to build an
37036 application using a programming model with 32-bit int, long, pointer, and off_t types.

37037 _CS_XBS5_ILP32_OFF32_LINTFLAGS

37038 If *sysconf(_SC_XBS5_ILP32_OFF32)* returns -1, the meaning of this value is unspecified.
37039 Otherwise, this value is the set of options to be given to the *lint* utility to check application
37040 source using a programming model with 32-bit int, long, pointer, and off_t types.

37041 _CS_XBS5_ILP32_OFFBIG_CFLAGS

37042 If *sysconf(_SC_XBS5_ILP32_OFFBIG)* returns -1, the meaning of this value is unspecified.
37043 Otherwise, this value is the set of initial options to be given to the *cc* and *c89* utilities to
37044 build an application using a programming model with 32-bit int, long, and pointer types,
37045 and an off_t type using at least 64 bits.

37046 _CS_XBS5_ILP32_OFFBIG_LDFLAGS

37047 If *sysconf(_SC_XBS5_ILP32_OFFBIG)* returns -1, the meaning of this value is unspecified.
37048 Otherwise, this value is the set of final options to be given to the *cc* and *c89* utilities to build
37049 an application using a programming model with 32-bit int, long, and pointer types, and an
37050 off_t type using at least 64 bits.

37051 _CS_XBS5_ILP32_OFFBIG_LIBS

37052 If *sysconf(_SC_XBS5_ILP32_OFFBIG)* returns -1, the meaning of this value is unspecified.
37053 Otherwise, this value is the set of libraries to be given to the *cc* and *c89* utilities to build an
37054 application using a programming model with 32-bit int, long, and pointer types, and an

37055	off_t type using at least 64 bits.
37056	_CS_XBS5_ILP32_OFFBIG_LINTFLAGS
37057	If <i>sysconf</i> (_SC_XBS5_ILP32_OFFBIG) returns -1, the meaning of this value is unspecified.
37058	Otherwise, this value is the set of options to be given to the <i>lint</i> utility to check an
37059	application using a programming model with 32-bit int, long, and pointer types, and an
37060	off_t type using at least 64 bits.
37061	_CS_XBS5_LP64_OFF64_CFLAGS
37062	If <i>sysconf</i> (_SC_XBS5_LP64_OFF64) returns -1, the meaning of this value is unspecified.
37063	Otherwise, this value is the set of initial options to be given to the <i>cc</i> and <i>c89</i> utilities to
37064	build an application using a programming model with 64-bit int, long, pointer, and off_t
37065	types.
37066	_CS_XBS5_LP64_OFF64_LDFLAGS
37067	If <i>sysconf</i> (_SC_XBS5_LP64_OFF64) returns -1, the meaning of this value is unspecified.
37068	Otherwise, this value is the set of final options to be given to the <i>cc</i> and <i>c89</i> utilities to build
37069	an application using a programming model with 64-bit int, long, pointer, and off_t types.
37070	_CS_XBS5_LP64_OFF64_LIBS
37071	If <i>sysconf</i> (_SC_XBS5_LP64_OFF64) returns -1, the meaning of this value is unspecified.
37072	Otherwise, this value is the set of libraries to be given to the <i>cc</i> and <i>c89</i> utilities to build an
37073	application using a programming model with 64-bit int, long, pointer, and off_t types.
37074	_CS_XBS5_LP64_OFF64_LINTFLAGS
37075	If <i>sysconf</i> (_SC_XBS5_LP64_OFF64) returns -1, the meaning of this value is unspecified.
37076	Otherwise, this value is the set of options to be given to the <i>lint</i> utility to check application
37077	source using a programming model with 64-bit int, long, pointer, and off_t types.
37078	_CS_XBS5_LPBIG_OFFBIG_CFLAGS
37079	If <i>sysconf</i> (_SC_XBS5_LPBIG_OFFBIG) returns -1, the meaning of this value is unspecified.
37080	Otherwise, this value is the set of initial options to be given to the <i>cc</i> and <i>c89</i> utilities to
37081	build an application using a programming model with an int type using at least 32 bits and
37082	long, pointer, and off_t types using at least 64 bits.
37083	_CS_XBS5_LPBIG_OFFBIG_LDFLAGS
37084	If <i>sysconf</i> (_SC_XBS5_LPBIG_OFFBIG) returns -1, the meaning of this value is unspecified.
37085	Otherwise, this value is the set of final options to be given to the <i>cc</i> and <i>c89</i> utilities to build
37086	an application using a programming model with an int type using at least 32 bits and long,
37087	pointer, and off_t types using at least 64 bits.
37088	_CS_XBS5_LPBIG_OFFBIG_LIBS
37089	If <i>sysconf</i> (_SC_XBS5_LPBIG_OFFBIG) returns -1, the meaning of this value is unspecified.
37090	Otherwise, this value is the set of libraries to be given to the <i>cc</i> and <i>c89</i> utilities to build an
37091	application using a programming model with an int type using at least 32 bits and long,
37092	pointer, and off_t types using at least 64 bits.
37093	_CS_XBS5_LPBIG_OFFBIG_LINTFLAGS
37094	If <i>sysconf</i> (_SC_XBS5_LPBIG_OFFBIG) returns -1, the meaning of this value is unspecified.
37095	Otherwise, this value is the set of options to be given to the <i>lint</i> utility to check application
37096	source using a programming model with an int type using at least 32 bits and long, pointer,
37097	and off_t types using at least 64 bits.
37098	The following symbolic constants are defined for the <i>lseek</i> () and <i>fcntl</i> () functions (they have
37099	distinct values):
37100	SEEK_SET Set file offset to <i>offset</i> .
37101	SEEK_CUR Set file offset to current plus <i>offset</i> .

37102 SEEK_END Set file offset to EOF plus *offset*.

37103 The following symbolic constants are defined for *sysconf()*:

37104 _SC_2_C_BIND

37105 _SC_2_C_DEV

37106 _SC_2_C_VERSION

37107 _SC_2_FORT_DEV

37108 _SC_2_FORT_RUN

37109 _SC_2_LOCALEDEF

37110 _SC_2_SW_DEV

37111 _SC_2_UPE

37112 _SC_2_VERSION

37113 _SC_ARG_MAX

37114 RT _SC_AIO_LISTIO_MAX

37115 _SC_AIO_MAX

37116 _SC_AIO_PRIO_DELTA_MAX

37117 _SC_ASYNCHRONOUS_IO

37118 EX _SC_ATEXIT_MAX

37119 _SC_BC_BASE_MAX

37120 _SC_BC_DIM_MAX

37121 _SC_BC_SCALE_MAX

37122 _SC_BC_STRING_MAX

37123 _SC_CHILD_MAX

37124 _SC_CLK_TCK

37125 _SC_COLL_WEIGHTS_MAX

37126 RT _SC_DELAYTIMER_MAX

37127 _SC_EXPR_NEST_MAX

37128 _SC_FSYNC

37129 _SC_GETGR_R_SIZE_MAX

37130 _SC_GETPW_R_SIZE_MAX

37131 EX _SC_IOV_MAX

37132 _SC_JOB_CONTROL

37133 _SC_LINE_MAX

37134 _SC_LOGIN_NAME_MAX

37135 _SC_MAPPED_FILES

37136 RT _SC_MEMLOCK

37137 _SC_MEMLOCK_RANGE

37138 _SC_MEMORY_PROTECTION

37139 RT _SC_MESSAGE_PASSING

37140 _SC_MQ_OPEN_MAX

37141 _SC_MQ_PRIO_MAX

37142 _SC_NGROUPS_MAX

37143 _SC_OPEN_MAX

37144 EX _SC_PAGESIZE

37145 _SC_PAGE_SIZE

37146 _SC_PASS_MAX (LEGACY)

37147 RT _SC_PRIORITIZED_IO

37148 _SC_PRIORITY_SCHEDULING

37149 _SC_RE_DUP_MAX

37150 RT _SC_REALTIME_SIGNALS

37151 _SC_RTSIG_MAX

37152 _SC_SAVED_IDS

37153	RT	_SC_SEMAPHORES
37154		_SC_SEM_NSEMS_MAX
37155		_SC_SEM_VALUE_MAX
37156		_SC_SHARED_MEMORY_OBJECTS
37157		_SC_SIGQUEUE_MAX
37158		_SC_STREAM_MAX
37159	RT	_SC_SYNCHRONIZED_IO
37160		_SC_THREADS
37161		_SC_THREAD_ATTR_STACKADDR
37162		_SC_THREAD_ATTR_STACKSIZE
37163		_SC_THREAD_DESTRUCTOR_ITERATIONS
37164		_SC_THREAD_KEYS_MAX
37165	RTT	_SC_THREAD_PRIORITY_SCHEDULING
37166		_SC_THREAD_PRIO_INHERIT
37167		_SC_THREAD_PRIO_PROTECT
37168		_SC_THREAD_PROCESS_SHARED
37169		_SC_THREAD_SAFE_FUNCTIONS
37170		_SC_THREAD_STACK_MIN
37171		_SC_THREAD_THREADS_MAX
37172	RT	_SC_TIMERS
37173		_SC_TIMER_MAX
37174		_SC_TTY_NAME_MAX
37175		_SC_TZNAME_MAX
37176		_SC_VERSION
37177	EX	_SC_XOPEN_VERSION
37178		_SC_XOPEN_CRYPT
37179		_SC_XOPEN_ENH_I18N
37180		_SC_XOPEN_SHM
37181		_SC_XOPEN_UNIX
37182		_SC_XOPEN_XCU_VERSION
37183		_SC_XBS5_ILP32_OFF32
37184		_SC_XBS5_ILP32_OFFBIG
37185		_SC_XBS5_LP64_OFF64
37186		_SC_XBS5_LPBIG_OFFBIG
37187		
37188		The two constants <code>_SC_PAGESIZE</code> and <code>_SC_PAGE_SIZE</code> may be defined to have the same
37189		value.
37190	EX	The following symbolic constants are defined as possible values for the <i>function</i> argument to the
37191		<i>lockf()</i> function:
37192		<code>F_LOCK</code> Lock a section for exclusive use.
37193		<code>F_ULOCK</code> Unlock locked sections.
37194		<code>F_TEST</code> Test section for locks by other processes.
37195		<code>F_TLOCK</code> Test and lock a section for exclusive use.
37196		
37197		The following symbolic constants are defined for <i>pathconf()</i> :
37198	RT	_PC_ASYNC_IO
37199		_PC_CHOWN_RESTRICTED
37200	EX	_PC_FILESIZEBITS
37201		_PC_LINK_MAX
37202		_PC_MAX_CANON

```

37203     _PC_MAX_INPUT
37204     _PC_NAME_MAX
37205     _PC_NO_TRUNC
37206     _PC_PATH_MAX
37207     _PC_PIPE_BUF
37208 RT   _PC_PRIO_IO
37209     _PC_SYNC_IO
37210     _PC_VDISABLE

```

37211 The following symbolic constants are defined for file streams:

```

37212     STDIN_FILENO      File number of stdin. It is 0.
37213     STDOUT_FILENO     File number of stdout. It is 1.
37214     STDERR_FILENO     File number of stderr. It is 2.

```

37215 Type Definitions

37216 EX The **size_t**, **ssize_t**, **uid_t**, **gid_t**, **off_t** and **pid_t** types are defined as described in <sys/types.h>.

37217 EX The **useconds_t** type is defined as described in <sys/types.h>.

37218 The **intptr_t** type is defined as described in <inttypes.h>.

37219 Declarations

37220 The following are declared as functions and may also be defined as macros. Function prototypes
37221 must be provided for use with an ISO C compiler.

```

37222     int      access(const char *, int);
37223     unsigned int alarm(unsigned int);
37224 EX   int      brk(void *);
37225     int      chdir(const char *);
37226 EX   int      chroot(const char *); (LEGACY)
37227     int      chown(const char *, uid_t, gid_t);
37228     int      close(int);
37229     size_t    confstr (int, char *, size_t);
37230 EX   char     *crypt(const char *, const char *);
37231     char     *ctermid(char *);
37232 EX   char     *cuserid(char *s); (LEGACY)
37233     int      dup(int);
37234     int      dup2(int, int);
37235 EX   void     encrypt(char[64], int);
37236     int      execl(const char *, const char *, ...);
37237     int      execle(const char *, const char *, ...);
37238     int      execlp(const char *, const char *, ...);
37239     int      execv(const char *, char *const []);
37240     int      execve(const char *, char *const [], char *const []);
37241     int      execvp(const char *, char *const []);
37242     void     _exit(int);
37243 EX   int      fchown(int, uid_t, gid_t);
37244     int      fchdir(int);
37245     pid_t    fork(void);
37246     long int  fpathconf(int, int);
37247     int      fsync(int);
37248     int      ftruncate(int, off_t);
37249     char     *getcwd(char *, size_t);

```

37250 EX	int	getdtablesize(void); (LEGACY)
37251	gid_t	getegid(void);
37252	uid_t	geteuid(void);
37253	gid_t	getgid(void);
37254	int	getgroups(int, gid_t []);
37255 EX	long	gethostid(void);
37256	char	*getlogin(void);
37257	int	getlogin_r(char *, size_t);
37258	int	getopt(int, char * const [], const char *);
37259 EX	int	getpagesize(void); (LEGACY)
37260	char	*getpass(const char *); (LEGACY)
37261	pid_t	getpgid(pid_t);
37262	pid_t	getpgrp(void);
37263	pid_t	getpid(void);
37264	pid_t	getppid(void);
37265 EX	pid_t	getsid(pid_t);
37266	uid_t	getuid(void);
37267 EX	char	*getwd(char *);
37268	int	isatty(int);
37269 EX	int	lchown(const char *, uid_t, gid_t);
37270	int	link(const char *, const char *);
37271 EX	int	lockf(int, int, off_t);
37272	off_t	lseek(int, off_t, int);
37273 EX	int	nice(int);
37274	long int	pathconf(const char *, int);
37275	int	pause(void);
37276	int	pipe(int [2]);
37277 EX	ssize_t	pread(int, void *, size_t, off_t);
37278	int	pthread_atfork(void (*)(void), void (*)(void),
37279		void (*)(void));
37280 EX	ssize_t	pwrite(int, const void *, size_t, off_t);
37281	ssize_t	read(int, void *, size_t);
37282 EX	int	readlink(const char *, char *, size_t);
37283	int	rmdir(const char *);
37284 EX	void	*sbrk(intptr_t);
37285	int	setgid(gid_t);
37286	int	setpgid(pid_t, pid_t);
37287 EX	pid_t	setpgrp(void);
37288	int	setregid(gid_t, gid_t);
37289	int	setreuid(uid_t, uid_t);
37290	pid_t	setsid(void);
37291	int	setuid(uid_t);
37292	unsigned int	sleep(unsigned int);
37293 EX	void	swab(const void *, void *, ssize_t);
37294 EX	int	symlink(const char *, const char *);
37295	void	sync(void);
37296	long int	sysconf(int);
37297	pid_t	tcgetpgrp(int);
37298	int	tcsetpgrp(int, pid_t);
37299 EX	int	truncate(const char *, off_t);
37300	char	*ttyname(int);
37301	int	ttyname_r(int, char *, size_t);

```

37302 EX    useconds_t    ualarm(useconds_t, useconds_t);
37303        int            unlink(const char *);
37304 EX    int            usleep(useconds_t);
37305        pid_t          vfork(void);
37306        ssize_t        write(int, const void *, size_t);

```

37307 The following external variables are declared:

```

37308        extern char    *optarg;
37309        extern int      optind, opterr, optopt;

```

37310 APPLICATION USAGE

37311 None.

37312 FUTURE DIRECTIONS

37313 None.

37314 SEE ALSO

37315 *access()*, *alarm()*, *chdir()*, *chown()*, *close()*, *crypt()*, *ctermid()*, *dup()*, *encrypt()*, *environ()*, *exec*,
37316 *exit()*, *fchdir()*, *fchown()*, *fcntl()*, *fork()*, *fpathconf()*, *fsync()*, *ftruncate()*, *getcwd()*, *getegid()*,
37317 *geteuid()*, *getgid()*, *getgroups()*, *gethostid()*, *getlogin()*, *getpgid()*, *getpgrp()*, *getpid()*, *getppid()*,
37318 *getsid()*, *getuid()*, *getwd()*, *isatty()*, *lchown()*, *link()*, *lockf()*, *lseek()*, *nice()*, *pathconf()*, *pause()*,
37319 *pipe()*, *read()*, *readlink()*, *rmdir()*, *setgid()*, *setpgid()*, *setpgrp()*, *setregid()*, *setreuid()*, *setsid()*,
37320 *setuid()*, *sleep()*, *swab()*, *symlink()*, *sync()*, *sysconf()*, *tcgetpgrp()*, *tcsetpgrp()*, *truncate()*, *ttyname()*,
37321 *ualarm()*, *unlink()*, *usleep()*, *vfork()*, *write()*, <limits.h>, <sys/types.h>, <termios.h>, Section 1.2
37322 on page 1.

37323 CHANGE HISTORY

37324 First released in Issue 1.

37325 Derived from Issue 1 of the SVID.

37326 Issue 4

37327 The following changes are incorporated for alignment with the ISO POSIX-1 standard and the
37328 ISO POSIX-2 standard:

- 37329 • The function declarations in this header are expanded to full ISO C prototypes.
- 37330 • A large number of new constants are defined for the *sysconf()* function, including all those
- 37331 with prefixes *_SC_2* and *_SC_BC*, plus:
 - 37332 *_SC_COLL_WEIGHTS_MAX*
 - 37333 *_SC_EXPR_NEST_MAX*
 - 37334 *_SC_LINE_MAX*
 - 37335 *_SC_RE_DUP_MAX*
 - 37336 *_SC_STREAM_MAX*
 - 37337 *_SC_TZNAME_MAX*

- 37338 • The *confstr()* function is added to the list of functions declared in this header, complete with
- 37339 a new set of constants for alignment with the ISO POSIX-2 standard.

37340 The following change is incorporated for alignment with the FIPS requirements:

- 37341 • The following symbolic constants are always defined:

37342 _POSIX_CHOWN_RESTRICTED
 37343 _POSIX_NO_TRUNC
 37344 _POSIX_VDISABLE
 37345 _POSIX_SAVED_IDS
 37346 _POSIX_JOB_CONTROL

37347 In Issue 3, they are only defined if the associated option is present.

37348 Other changes are incorporated as follows:

- 37349 • The symbolic constants F_ULOCK, F_LOCK, F_TLOCK, F_TEST, GF_PATH, IF_PATH and
 37350 PF_PATH are withdrawn.
- 37351 • The required value of _XOPEN_VERSION is defined and the constant is marked as an
 37352 extension.
- 37353 • The constants _XOPEN_XPG2, _XOPEN_XPG3 and _XOPEN_XPG4 are added.
- 37354 • The constants _POSIX2_* are added.
- 37355 • Reference to the header <sys/types.h> is added for the definitions of **size_t**, **ssize_t**, **uid_t**,
 37356 **gid_t**, **off_t** and **pid_t**. These are marked as extensions.
- 37357 • The names *chroot()*, *crypt()*, *encrypt()*, *fsync()*, *getopt()*, *getpass()*, *nice()* and *swab()* are added
 37358 to the list of functions declared in this header. With the exception of *getopt()*, these are all
 37359 marked as extensions.
- 37360 • The APPLICATION USAGE section is removed.

37361 **Issue 4, Version 2**

37362 The following changes are incorporated for X/OPEN UNIX conformance:

- 37363 • The Feature Group constant _XOPEN_UNIX is defined.
- 37364 • The *sysconf()* symbolic constants _SC_ATEXIT_MAX, _SC_IOV_MAX, _SC_PAGESIZE and
 37365 _SC_PAGE_SIZE are defined.
- 37366 • The *brk()*, *fchown()*, *fchdir()*, *ftruncate()*, *gethostid()*, *getpagesize()*, *getpgid()*, *getsid()*, *getwd()*,
 37367 *lchown()*, *lockf()*, *readlink()*, *sbrk()*, *setpgrp()*, *setregid()*, *setreuid()*, *symlink()*, *sync()*, *truncate()*,
 37368 *ualarm()*, *usleep()* and *vfork()* functions are added to the list of functions declared in this
 37369 header.
- 37370 • The symbolic constants F_ULOCK, F_LOCK, F_TLOCK and F_TEST are added.

37371 **Issue 5**

37372 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
 37373 Threads Extension.

37374 The symbolic constants _XOPEN_REALTIME and _XOPEN_REALTIME_THREADS are added.
 37375 _POSIX2_C_BIND, _XOPEN_ENH_I18N and _XOPEN_SHM must now be set to a value other
 37376 than -1 by a conforming implementation.

37377 Large File System extensions added.

37378 The type of the argument to *sbrk()* is changed from **int** to **intptr_t**.

37379 _XBS_ constants are added to the list of Constants for Options and Feature Groups, to the list of
 37380 constants for the *confstr()* function, and to the list of constants to the *sysconf()* function. These
 37381 are all marked EX.

37382 **NAME**

37383 utime.h — access and modification times structure

37384 **SYNOPSIS**

37385 #include <utime.h>

37386 **DESCRIPTION**37387 The <utime.h> header declares the structure **utimbuf**, which includes the following members:

37388 time_t actime access time

37389 time_t modtime modification time

37390 The times are measured in seconds since the Epoch.

37391 EX The type **time_t** is defined as described in <sys/types.h>.37392 The following is declared as a function and may also be defined as a macro. Function prototypes
37393 must be provided for use with an ISO C compiler.

37394 int utime(const char *, const struct utimbuf *);

37395 **APPLICATION USAGE**

37396 None.

37397 **FUTURE DIRECTIONS**

37398 None.

37399 **SEE ALSO**

37400 utime(), <sys/types.h>.

37401 **CHANGE HISTORY**

37402 First released in Issue 3.

37403 **Issue 4**

37404 The following change is incorporated for alignment with the ISO POSIX-1 standard:

- 37405
- The function declarations in this header are expanded to full ISO C prototypes.

37406 Another change is incorporated as follows:

- 37407
- Reference to the <sys/types.h> header is added for the definition of **time_t**. This is marked
37408 as an extension.

37409 NAME

37410 utmpx.h — user accounting database definitions

37411 SYNOPSIS

37412 EX `#include <utmpx.h>`

37413

37414 DESCRIPTION

37415 The <utmpx.h> header defines the **utmpx** structure that includes at least the following members:

37416	char	ut_user[]	user login name
37417	char	ut_id[]	unspecified initialisation process identifier
37418	char	ut_line[]	device name
37419	pid_t	ut_pid	process id
37420	short int	ut_type	type of entry
37421	struct timeval	ut_tv	time entry was made

37422 The **pid_t** type is defined through **typedef** as described in <sys/types.h>.

37423 The **timeval** structure is defined as described in <sys/time.h>.

37424 Inclusion of the <utmpx.h> header may also make visible all symbols from <sys/time.h>.

37425 The following symbolic constants are defined as possible values for the **ut_type** member of the
37426 **utmpx** structure:

37427	EMPTY	No valid user accounting information.
37428	BOOT_TIME	Identifies time of system boot.
37429	OLD_TIME	Identifies time when system clock changed.
37430	NEW_TIME	Identifies time after system clock changed.
37431	USER_PROCESS	Identifies a process.
37432	INIT_PROCESS	Identifies a process spawned by the init process.
37433	LOGIN_PROCESS	Identifies the session leader of a logged in user.
37434	DEAD_PROCESS	Identifies a session leader who has exited.

37435 The following are declared as functions and may also be defined as macros. Function prototypes
37436 must be provided for use with an ISO C compiler.

37437	void	endutxent(void);
37438	struct utmpx	*getutxent(void);
37439	struct utmpx	*getutxid(const struct utmpx *);
37440	struct utmpx	*getutxline(const struct utmpx *);
37441	struct utmpx	*pututxline(const struct utmpx *);
37442	void	setutxent(void);

37443 APPLICATION USAGE

37444 None.

37445 FUTURE DIRECTIONS

37446 None.

37447 SEE ALSO

37448 *endutxent()*.

37449 CHANGE HISTORY

37450 First released in Issue 4, Version 2.

37451 NAME

37452 varargs.h — handle variable argument list (**LEGACY**)

37453 SYNOPSIS

37454 EX #include <varargs.h>

37455

37456 va_alist

37457 va_dcl

37458 void va_start(pvar)

37459 va_list pvar;

37460 type va_arg(pvar, type)

37461 va_list pvar;

37462 void va_end(pvar)

37463 va_list pvar;

37464 DESCRIPTION

37465 The <varargs.h> header contains a set of macros which allows portable procedures that accept
 37466 variable argument lists to be written. Routines that have variable argument lists (such as
 37467 *printf()*) but do not use <varargs.h> are inherently non-portable, as different machines use
 37468 different argument-passing conventions.

37469 **va_alist** Used as the parameter list in a function header.

37470 **va_dcl** A declaration for **va_alist**. No semicolon should follow **va_dcl**.

37471 **va_list** A type defined for the variable used to traverse the list.

37472 **va_start()** Called to initialise *pvar* to the beginning of the list.

37473 **va_arg()** Will return the next argument in the list pointed to by *pvar*. The argument
 37474 *type* is the type the argument is expected to be. Different types can be mixed,
 37475 but it is up to the routine to know what type of argument is expected, as it
 37476 cannot be determined at run time.

37477 **va_end()** Used to clean up.

37478 Multiple traversals, each bracketed by *va_start()* ... *va_end()*, are possible.

37479 EXAMPLES

37480 This example is a possible implementation of *execl()*.

37481 #include <varargs.h>

37482 #define MAXARGS 100

37483 /* execl is called by

37484 * execl(file, arg1, arg2, ..., (char *)0);

37485 */

37486 execl(va_alist)

37487 va_dcl

37488 {

37489 va_list ap;

37490 char *file;

37491 char *args[MAXARGS];

37492 int argno = 0;

37493 va_start(ap);

37494 file = va_arg(ap, char *);

37495 while ((args[argno++] = va_arg(ap, char *)) != (char *)0)

37496 ;

```

37497         va_end(ap);
37498         return execv(file, args);
37499     }

```

37500 APPLICATION USAGE

37501 It is up to the calling routine to specify how many arguments there are, since it is not always
 37502 possible to determine this from the stack frame. For example, *execl()* is passed a zero pointer to
 37503 signal the end of the list. The *printf()* function can tell how many arguments are there by the
 37504 format.

37505 It is non-portable to specify a second argument of **char**, **short** or **float** to *va_arg()*, since
 37506 arguments seen by the called function are not type **char**, **short** or **float**. C language converts
 37507 type **char** and **short** arguments to **int** and converts type **float** arguments to **double** before
 37508 passing them to a function.

37509 For backward compatibility with Issue 3, XSI-conformant systems support <varargs.h> as well
 37510 as <stdarg.h>. Use of <varargs.h> is not recommended.

37511 FUTURE DIRECTIONS

37512 None.

37513 SEE ALSO

37514 *exec*, *printf()*, <stdarg.h>.

37515 CHANGE HISTORY

37516 First released in Issue 1.

37517 Issue 4

37518 The following changes are incorporated in this issue:

- 37519 • The interface is marked TO BE WITHDRAWN.
- 37520 • The APPLICATION USAGE section is added, recommending use of <stdarg.h> in preference
 37521 to this header.
- 37522 • The FUTURE DIRECTIONS section is removed.

37523 Issue 5

37524 Marked LEGACY.

37525 NAME

37526 wchar.h — wide-character types

37527 SYNOPSIS

37528 #include <wchar.h>

37529 DESCRIPTION

37530 The <wchar.h> header defines the following data types through **typedef**:

37531	wchar_t	As described in <stddef.h>.
37532	wint_t	An integral type capable of storing any valid value of wchar_t , or WEOF .
37533	wctype_t	A scalar type of a data object that can hold values which represent locale-specific character classification.
37534		
37535	mbstate_t	An object type other than an array type that can hold the conversion state information necessary to convert between sequences of (possibly multibyte) characters and wide-characters. If a codeset is being used such that an
37536		mbstate_t needs to preserve more than 2 levels of reserved state, the results
37537 EX		are unspecified.
37538		
37539	FILE	As described in <stdio.h>.
37540 EX	size_t	As described in <stddef.h>.
37541		

37542 The <wchar.h> header declares the following as functions and may also define them as macros.
 37543 Function prototypes must be provided for use with an ISO C compiler.

37544	wint_t	btowc(int);
37545	int	fwprintf(FILE *, const wchar_t *, ...);
37546	int	fwscanf(FILE *, const wchar_t *, ...);
37547	int	iswalnum(wint_t);
37548	int	iswalpha(wint_t);
37549	int	iswcntrl(wint_t);
37550	int	iswdigit(wint_t);
37551	int	iswgraph(wint_t);
37552	int	iswlower(wint_t);
37553	int	iswprint(wint_t);
37554	int	iswpunct(wint_t);
37555	int	iswspace(wint_t);
37556	int	iswupper(wint_t);
37557	int	iswxdigit(wint_t);
37558	int	iswctype(wint_t, wctype_t);
37559	wint_t	fgetwc(FILE *);
37560	wchar_t	*fgetws(wchar_t *, int, FILE *);
37561	wint_t	fputwc(wchar_t, FILE *);
37562	int	fputws(const wchar_t *, FILE *);
37563	int	fwide(FILE *, int);
37564	wint_t	getwc(FILE *);
37565	wint_t	getwchar(void);
37566	size_t	mbsinit(const mbstate_t *);
37567	size_t	mbrlen(const char *, size_t, mbstate_t *);
37568	size_t	mbrtowc(wchar_t *, const char *, size_t,
37569		mbstate_t *);
37570	size_t	mbsrtowcs(wchar_t *, const char **, size_t,
37571		mbstate_t *);
37572	wint_t	putwc(wchar_t, FILE *);
37573	wint_t	putwchar(wchar_t);
37574	int	swprintf(wchar_t *, size_t, const wchar_t *, ...);

```

37575     int                swscanf(const wchar_t *, const wchar_t *, ...);
37576     wint_t            towlower(wint_t);
37577     wint_t            towupper(wint_t);
37578     wint_t            ungetwc(wint_t, FILE *);
37579     int               vfwprintf(FILE *, const wchar_t *, va_list);
37580     int               vwprintf(const wchar_t *, va_list);
37581     int               vswprintf(wchar_t *, size_t, const wchar_t *,
37582                                va_list);
37583     size_t            wcrctomb(char *, wchar_t, mbstate_t *);
37584     wchar_t           *wcscat(wchar_t *, const wchar_t *);
37585     wchar_t           *wcschr(const wchar_t *, wchar_t);
37586     int               wcscmp(const wchar_t *, const wchar_t *);
37587     int               wscoll(const wchar_t *, const wchar_t *);
37588     wchar_t           *wcscpy(wchar_t *, const wchar_t *);
37589     size_t            wcscspn(const wchar_t *, const wchar_t *);
37590     size_t            wcsftime(wchar_t *, size_t, const wchar_t *,
37591                                const struct tm *);
37592     size_t            wcslen(const wchar_t *);
37593     wchar_t           *wcsncat(wchar_t *, const wchar_t *, size_t);
37594     int               wcsncmp(const wchar_t *, const wchar_t *, size_t);
37595     wchar_t           *wcsncpy(wchar_t *, const wchar_t *, size_t);
37596     wchar_t           *wcpbrk(const wchar_t *, const wchar_t *);
37597     wchar_t           *wcsrchr(const wchar_t *, wchar_t);
37598     size_t            wcsrtombs(char *, const wchar_t **, size_t,
37599                                mbstate_t *);
37600     size_t            wcsspncpy(const wchar_t *, const wchar_t *);
37601     wchar_t           *wcsstr(const wchar_t *, const wchar_t *);
37602     double            wcstod(const wchar_t *, wchar_t **);
37603     wchar_t           *wcstok(wchar_t *, const wchar_t *, wchar_t **);
37604     long int          wcstol(const wchar_t *, wchar_t **, int);
37605     unsigned long int wcstoul(const wchar_t *, wchar_t **, int);
37606 EX  wchar_t           *wcswcs(const wchar_t *, const wchar_t *);
37607     int               wcswidth(const wchar_t *, size_t);
37608     size_t            wcsxfrm(wchar_t *, const wchar_t *, size_t);
37609     int               wctob(wint_t);
37610     wctype_t          wctype(const char *);
37611     int               wcwidth(wchar_t);
37612     wchar_t           *wmemchr(const wchar_t *, wchar_t, size_t);
37613     int               wmemcmp(const wchar_t *, const wchar_t *, size_t);
37614     wchar_t           *wmemcpy(wchar_t *, const wchar_t *, size_t);
37615     wchar_t           *wmemmove(wchar_t *, const wchar_t *, size_t);
37616     wchar_t           *wmemset(wchar_t *, wchar_t, size_t);
37617     int               wprintf(const wchar_t *, ...);
37618     int               wscanf(const wchar_t *, ...);

```

37619 **<wchar.h>** defines the following macro names:

37620	WCHAR_MAX	The maximum value representable by an object of type wchar_t .
37621	WCHAR_MIN	The minimum value representable by an object of type wchar_t .
37622	WEOF	Constant expression of type wint_t that is returned by several WP functions to indicate end-of-file.
37623		
37624	NULL	As described in <stddef.h> .

37625 The tag **tm** is declared as naming an incomplete structure type, the contents of which are
 37626 described in the header <**time.h**>.

37627 Inclusion of the <**wchar.h**> header may make visible all symbols from the headers <**ctype.h**>,
 37628 <**stdio.h**>, <**stdarg.h**>, <**stdlib.h**>, <**string.h**>, <**stddef.h**> and <**time.h**>.

37629 APPLICATION USAGE

37630 None.

37631 FUTURE DIRECTIONS

37632 None.

37633 SEE ALSO

37634 *btowc()*, *fwprintf()*, *fwscanf()*, *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswdigit()*, *iswgraph()*,
 37635 *iswlower()*, *iswprint()*, *iswpunct()*, *iswspace()*, *iswupper()*, *iswxdigit()*, *iswctype()*, *fgetwc()*,
 37636 *fgetws()*, *fputwc()*, *fputws()*, *fwide()*, *getwc()*, *getwchar()*, *getws()*, *mbsinit()*, *mbrlen()*, *mbrtowc()*,
 37637 *mbsrtowcs()*, *putwc()*, *putwchar()*, *putws()*, *swprintf()*, *swscanf()*, *towlower()*, *towupper()*,
 37638 *ungetwc()*, *vfwprintf()*, *vwprintf()*, *vswprintf()*, *wcrtomb()*, *wcrtombs()*, *wscat()*, *wcschr()*,
 37639 *wscmp()*, *wscoll()*, *wscpy()*, *wscspn()*, *wcsftime()*, *wcslen()*, *wcsncat()*, *wcsncmp()*, *wcsncpy()*,
 37640 *wcspbrk()*, *wcsrchr()*, *wcsspn()*, *wcsstr()*, *wctod()*, *wctok()*, *wctol()*, *wcstoul()*, *wcswcs()*,
 37641 *wcswidth()*, *wcsxfrm()*, *wctob()*, *wctype()*, *wcwidth()*, *wmemchr()*, *wmemcmp()*, *wmemcpy()*,
 37642 *wmemmove()*, *wmemset()*, *wprintf()*, *wscanf()*, <**ctype.h**>, <**stdio.h**>, <**stdarg.h**>, <**stdlib.h**>,
 37643 <**string.h**>, <**stddef.h**> and <**time.h**>.

37644 CHANGE HISTORY

37645 First released in Issue 4.

37646 Issue 5

37647 Aligned with the ISO/IEC 9899:1990/Amendment 1:1994 (E).

37648 NAME

37649 wctype.h — wide-character classification and mapping utilities

37650 SYNOPSIS

37651 #include <wctype.h>

37652 DESCRIPTION

37653 The <wctype.h> header defines the following data types through **typedef**:

37654	wint_t	As described in <wchar.h>.
37655	wctrans_t	A scalar type that can hold values which represent locale-specific character mappings.
37656		
37657	wctype_t	As described in <wchar.h>.

37658 The <wctype.h> header declares the following as functions and may also define them as macros.
37659 Function prototypes must be provided for use with an ISO C compiler.

```

37660 int      iswalnum(wint_t);
37661 int      iswalpha(wint_t);
37662 int      iswcntrl(wint_t);
37663 int      iswdigit(wint_t);
37664 int      iswgraph(wint_t);
37665 int      iswlower(wint_t);
37666 int      iswprint(wint_t);
37667 int      iswpunct(wint_t);
37668 int      iswspace(wint_t);
37669 int      iswupper(wint_t);
37670 int      iswxdigit(wint_t);
37671 int      iswctype(wint_t, wctype_t);
37672 wint_t   towctrans(wint_t, wctrans_t);
37673 wint_t   towlower(wint_t);
37674 wint_t   towupper(wint_t);
37675 wctrans_t wctrans(const char *);
37676 wctype_t wctype(const char *);

```

37677 <wctype.h> defines the following macro name:

37678	WEOF	Constant expression of type wint_t that is returned by several MSE functions to indicate end-of-file.
37679		

37680 For all functions described in this header that accept an argument of type **wint_t**, the value will
37681 be representable as a **wchar_t** or will equal the value of **WEOF**. If this argument has any other
37682 value, the behaviour is undefined.

37683 The behaviour of these functions is affected by the LC_CTYPE category of the current locale.

37684 Inclusion of the <wctype.h> header may make visible all symbols from the headers <ctype.h>,
37685 <stdio.h>, <stdarg.h>, <stdlib.h>, <string.h>, <stddef.h> <time.h>. and <wchar.h>.

37686 APPLICATION USAGE

37687 None.

37688 FUTURE DIRECTIONS

37689 None.

37690 SEE ALSO

37691 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*,
37692 *iswpunct()*, *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, *towctrans()*, *towlower()*, *towupper()*,
37693 *wctrans()*, *wctype()*, <locale.h>. <wchar.h>.

37694 **CHANGE HISTORY**

37695 First released in Issue 5.

37696 Derived from the ISO/IEC 9899:1990/Amendment 1:1994 (E).

37697 **NAME**

37698 wordexp.h — word-expansion types

37699 **SYNOPSIS**

37700 #include <wordexp.h>

37701 **DESCRIPTION**

37702 The <wordexp.h> header defines the structures and symbolic constants used by the *wordexp()*
37703 and *wordfree()* functions.

37704 The structure type **wordexp_t** contains at least the following members:

37705	size_t	we_wordc	count of words matched by <i>words</i>
37706	char	**we_wordv	pointer to list of expanded words
37707	size_t	we_offs	slots to reserve at the beginning of <i>we_wordv</i>

37708 The *flags* argument to the *wordexp()* function is the bitwise inclusive OR of the following flags:

37709	WRDE_APPEND	Append words to those previously generated.
37710	WRDE_DOOFFS	Number of null pointers to prepend to <i>we_wordv</i> .
37711	WRDE_NOCMD	Fail if command substitution is requested.
37712	WRDE_REUSE	The <i>pwordexp</i> argument was passed to a previous successful call to <i>wordexp()</i> , and has not been passed to <i>wordfree()</i> . The result will be the same as if the application had called <i>wordfree()</i> and then called <i>wordexp()</i> without WRDE_REUSE.
37713		
37714		
37715		
37716	WRDE_SHOWERR	Do not redirect <i>stderr</i> to <i>/dev/null</i> .
37717	WRDE_UNDEF	Report error on an attempt to expand an undefined shell variable.

37718 The following constants are defined as error return values:

37719	WRDE_BADCHAR	One of the unquoted characters:
37720		<newline> & ; < > () { }
37721		appears in <i>words</i> in an inappropriate context.
37722	WRDE_BADVAL	Reference to undefined shell variable when WRDE_UNDEF is set in <i>flags</i> .
37723	WRDE_CMDSUB	Command substitution requested when WRDE_NOCMD was set in <i>flags</i> .
37724	WRDE_NOSPACE	Attempt to allocate memory failed.
37725	WRDE_NOSYS	The implementation does not support the function.
37726	WRDE_SYNTAX	Shell syntax error, such as unbalanced parentheses or unterminated string.
37727		

37728 The following are declared as functions and may also be declared as macros. Function
37729 prototypes must be provided for use with an ISO C compiler.

37730	int	wordexp(const char *, wordexp_t *, int);
37731	void	wordfree(wordexp_t *);

37732 The implementation may define additional macros or constants using names beginning with
37733 WRDE_.

37734 **APPLICATION USAGE**

37735 None.

37736 **FUTURE DIRECTIONS**

37737 None.

37738 **SEE ALSO**

37739 *wordexp()*, the XCU specification.

37740 **CHANGE HISTORY**

37741 First released in Issue 4.

37742 Derived from the ISO POSIX-2 standard.